Read the Docs Template Documentation

Release v3.1.7

Read the Docs

May 19, 2020

Contents

1 Get Started **3**
1.1 Introduction . . . 3
1.2 What You Need . . . 3
1.3 Guides . . . 4
1.4 Setup Toolchain . . . 35
1.5 Get ESP-IDF . . . 44
1.6 Setup Path to ESP-IDF . . . 45
1.7 Start a Project . . . 45
1.8 Connect . . . 45
1.9 Configure . . . 45
1.10 Build and Flash . . . 46
1.11 Monitor . . . 47
1.12 Updating ESP-IDF . . . 48
1.13 Related Documents . . . 48

2 Get Started (CMake) **65**
2.1 Introduction . . . 65
2.2 What You Need . . . 66
2.3 Development Board Guides . . . 66
2.4 Setup Toolchain . . . 97
2.5 Get ESP-IDF . . . 107
2.6 Setup Environment Variables . . . 108
2.7 Start a Project . . . 108
2.8 Connect . . . 109
2.9 Configure . . . 109
2.10 Build The Project . . . 110
2.11 Flash To A Device . . . 111
2.12 Monitor . . . 112
2.13 Updating ESP-IDF . . . 113
2.14 Related Documents . . . 114

3 API Reference **125**
3.1 Wi-Fi API . . . 125
3.2 Mesh API . . . 169
3.3 Bluetooth API . . . 201
3.4 Ethernet API . . . 350
3.5 Peripherals API . . . 358

3.6 Protocols API . . . 573
3.7 Storage API . . . 598
3.8 System API . . . 666
3.9 Configuration Options . . . 842
3.10 Error Codes Reference . . . 941

4 ESP32 Hardware Reference **947**
4.1 ESP32 Modules and Boards . . . 947
4.2 Previous Versions of ESP32 Modules and Boards . . . 953

5 API Guides **959**
5.1 General Notes About ESP-IDF Programming . . . 959
5.2 Build System . . . 964
5.3 Build System (CMake) . . . 976
5.4 Error Handling . . . 997
5.5 Fatal Errors . . . 999
5.6 Deep Sleep Wake Stubs . . . 1005
5.7 ESP32 Core Dump . . . 1007
5.8 Flash Encryption . . . 1009
5.9 ESP-IDF FreeRTOS SMP Changes . . . 1019
5.10 Thread Local Storage . . . 1026
5.11 High-Level Interrupts . . . 1027
5.12 JTAG Debugging . . . 1028
5.13 Bootloader . . . 1084
5.14 Partition Tables . . . 1086
5.15 Secure Boot . . . 1089
5.16 ULP coprocessor programming . . . 1096
5.17 Unit Testing in ESP32 . . . 1120
5.18 Console . . . 1123
5.19 ESP32 ROM console . . . 1126
5.20 Wi-Fi Driver . . . 1129
5.21 ESP-MESH . . . 1162
5.22 BluFi . . . 1182
5.23 Support for external RAM . . . 1189

6 Contributions Guide **1193**
6.1 How to Contribute . . . 1193
6.2 Before Contributing . . . 1193
6.3 Pull Request Process . . . 1194
6.4 Legal Part . . . 1194
6.5 Related Documents . . . 1194

7 ESP-IDF Versions **1211**
7.1 Releases . . . 1211
7.2 Which Version Should I Start With? . . . 1212
7.3 Versioning Scheme . . . 1213
7.4 Checking The Current Version . . . 1213
7.5 Git Workflow . . . 1213
7.6 Updating ESP-IDF . . . 1214

8 Resources **1217**

9 Copyrights and Licenses **1219**
9.1 Software Copyrights . . . 1219
9.2 ROM Source Code Copyrights . . . 1220

9.3 Xtensa libhal MIT License 1220
9.4 TinyBasic Plus MIT License 1221
9.5 TJpgDec License 1221

10 About **1223**

11 Switch Between Languages/ **1225**

Index **1227**

[]

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

The documentation has different language versions (English, , *How to switch between languages?*). However, please refer to the English version if there is any discrepancy.

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

CHAPTER 1

Get Started

[]

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 stable version v3.1.7 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**

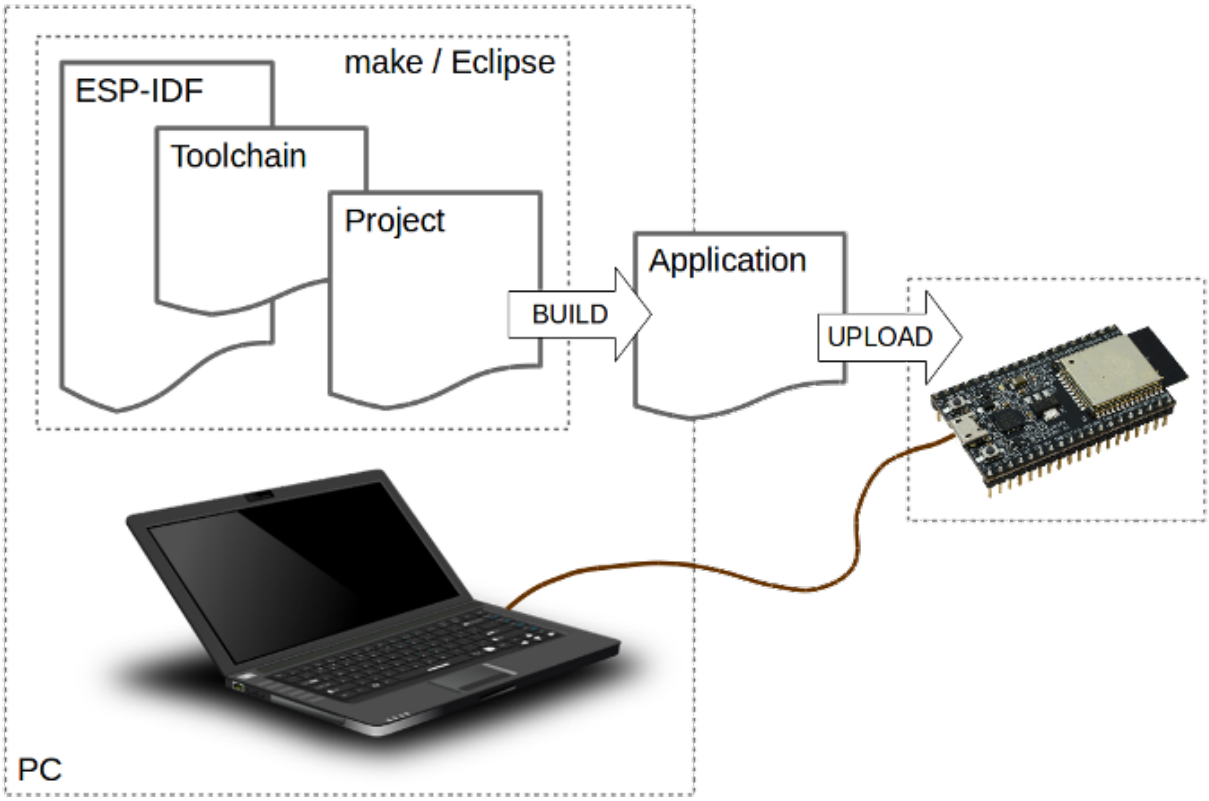


Fig. 1: Development of applications for ESP32

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 V4 Getting Started Guide

This user guide shows how to get started with ESP32-DevKitC V4 development board. For description of other versions of the ESP32-DevKitC check *ESP32 Hardware Reference*.

What You Need

- 1 × *ESP32-DevKitC V4 board*
- 1 × USB A / micro USB B cable
- 1 × PC loaded with Windows, Linux or Mac OS

Overview

ESP32-DevKitC V4 is a small-sized ESP32-based development board produced by Espressif. Most of the I/O pins are broken out to the pin headers on both sides for easy interfacing. Developers can connect these pins to peripherals as needed. Standard headers also make development easy and convenient when using a breadboard.

The board comes in two versions, either with *ESP-WROOM-32* or *ESP32-WROVER* module soldered.

Functional Description

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

ESP-WROOM-32 *ESP-WROOM-32* module soldered to the ESP32-DevKitC V4 board.

ESP32-WROVER Optionally *ESP32-WROVER* module may be soldered instead of the ESP-WROOM-32.

USB-UART Bridge A single chip USB-UART bridge provides up to 3 Mbps transfers rates.

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.

Micro USB Port USB interface. It functions as the power supply for the board and the communication interface between PC and the ESP module.

5V Power On LED This LED lights when the USB or an external 5V power supply is applied to the board. For details see schematic in *Related Documents*.

EN Reset button: pressing this button resets the system.

I/O Most of the pins on the ESP module 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.

> **Note:** Some of broken out pins are used internally be the ESP32 module to communicate with SPI memory. They are grouped on one side of the board besides the USB connector and labeled D0, D1, D2, D3, CMD and CLK. In general these pins should be left unconnected or access to the SPI flash memory / SPI RAM may be disturbed.

> **Note:** GPIO16 and 17 are used internally by the ESP32-WROVER module. They are broken out and avialable for use only for boards that have the ESP-WROOM-32 module installed.

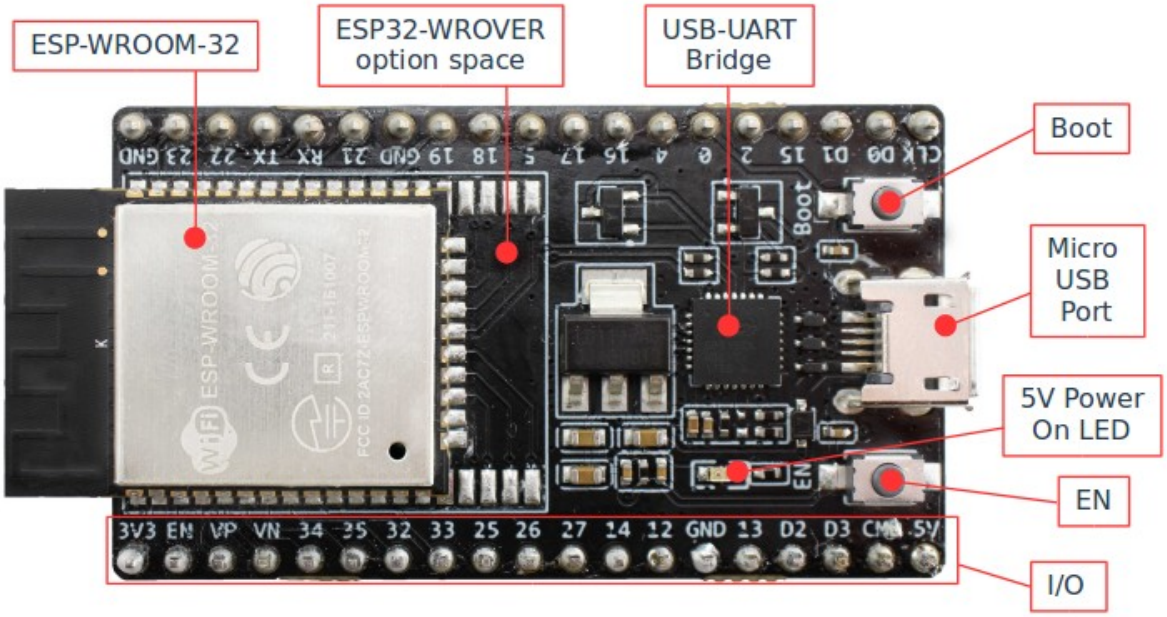


Fig. 2: ESP32-DevKitC V4 with ESP-WROOM-32 module soldered

Power Supply Options

There following options are available to provide power supply to this board:

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:

- *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

Board Dimensions

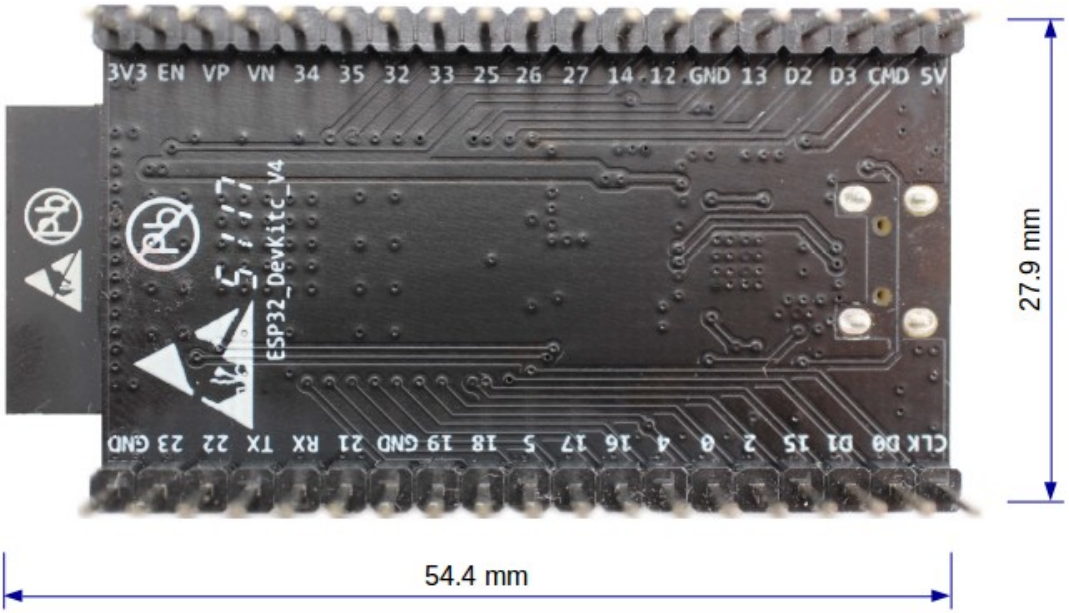


Fig. 3: ESP32 DevKitC board dimensions - back

Related Documents

- ESP32-DevKitC V4 schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP-WROOM-32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)

ESP32-DevKitC V2 Getting Started Guide

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

What You Need

- 1 × *ESP32-DevKitC V2 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.

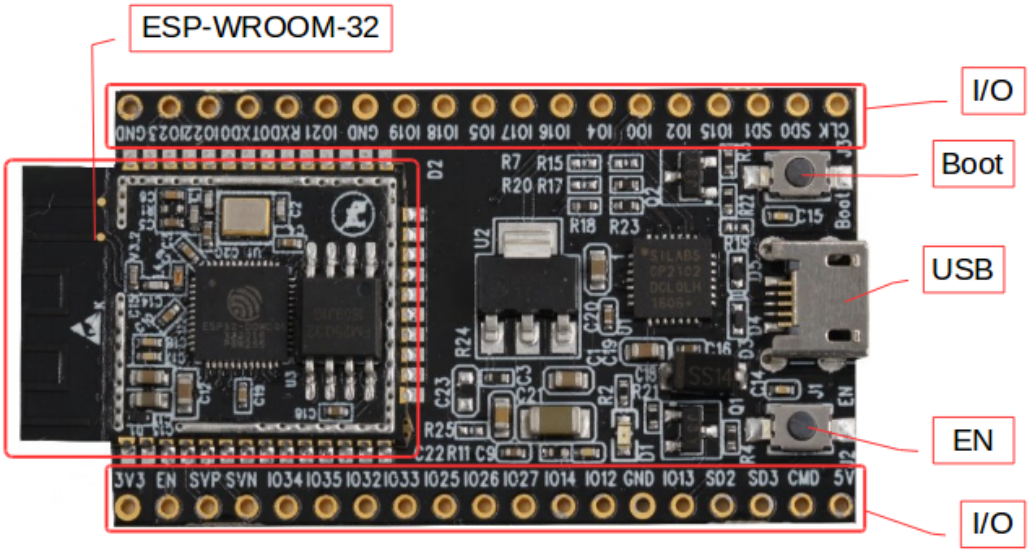


Fig. 4: ESP32-DevKitC V2 board layout

Power Supply Options

There following options are available to provide power supply to this board:

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:

- *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.

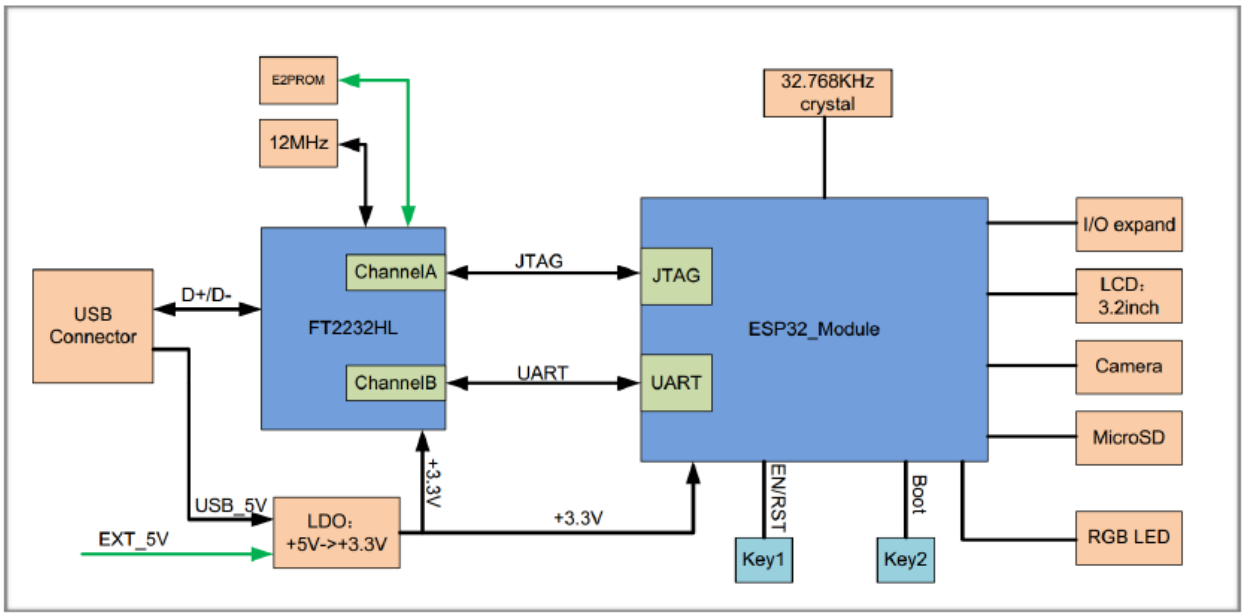


Fig. 5: ESP-WROVER-KIT block diagram

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*.

Setup Options

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

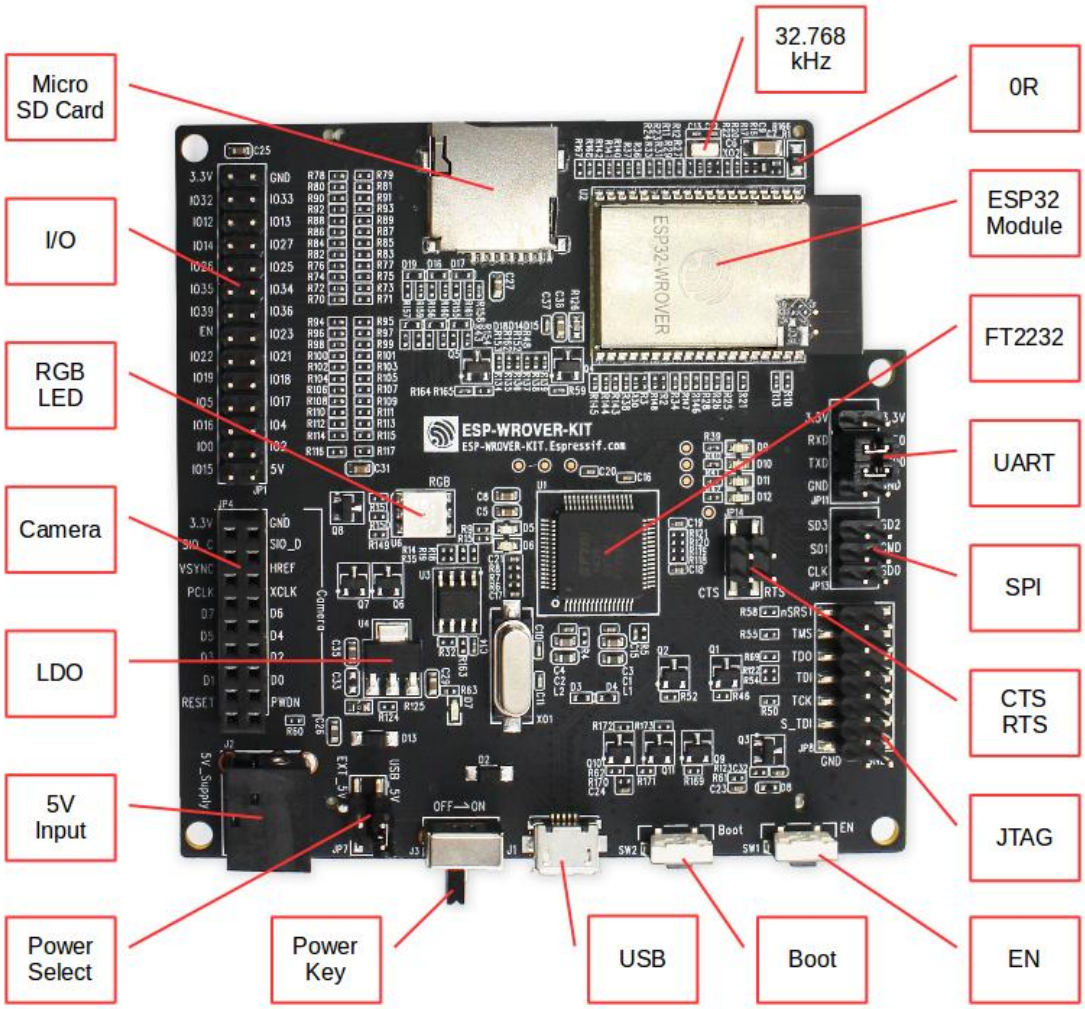


Fig. 6: ESP-WROVER-KIT board layout - front

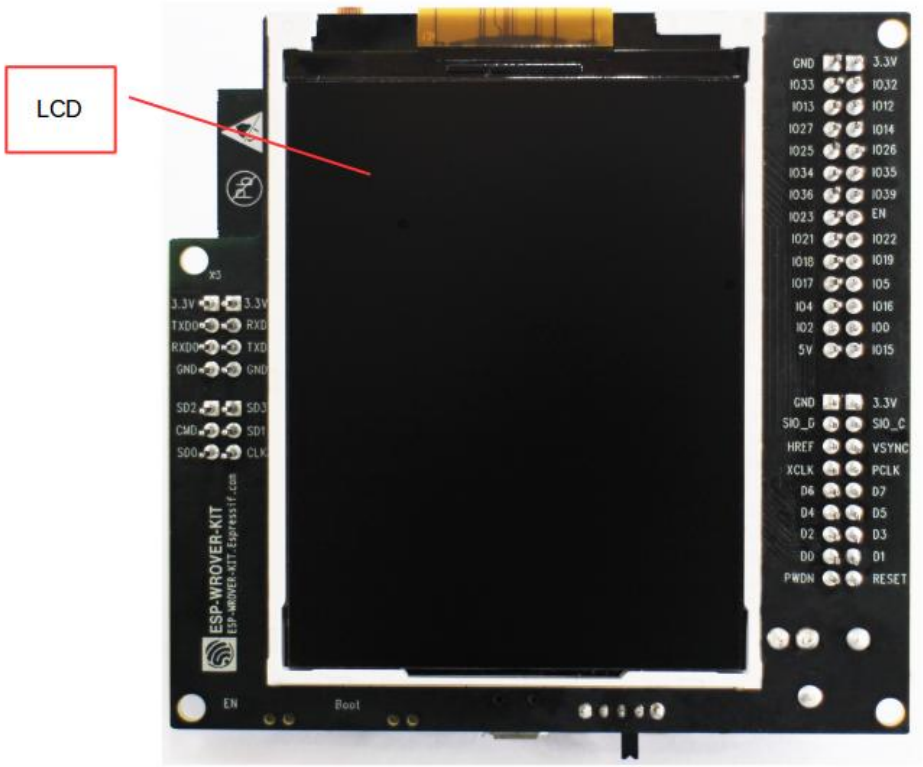
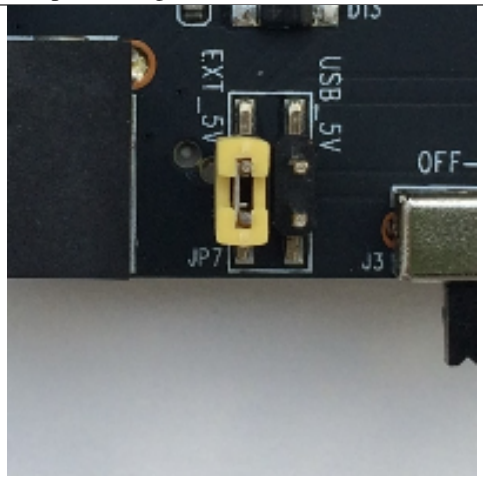
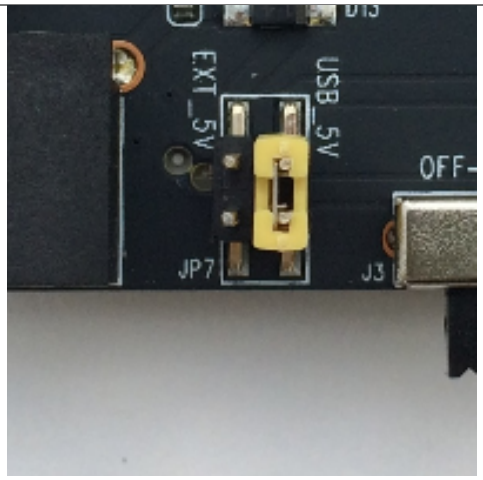
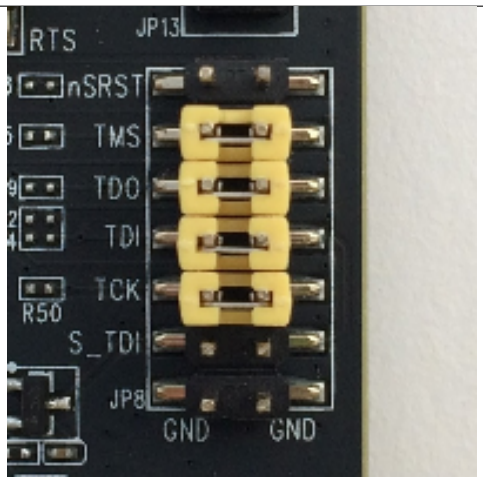
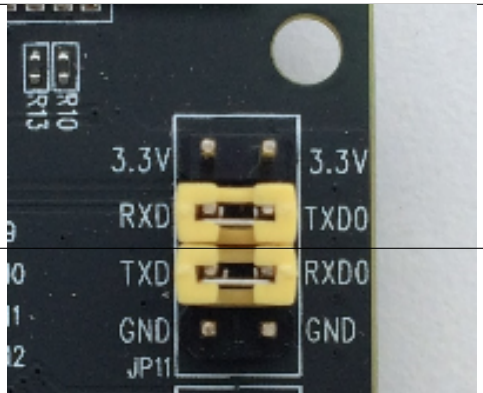


Fig. 7: ESP-WROVER-KIT board layout - back

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
		

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

Important: The module's flash bus is connected to the pin header JP13 through 0-Ohm resistors R140 ~ R145. If the flash frequency needs to operate at 80 MHz, to improve integrity of the bus signals, it is recommended to desolder resistors R140 ~ R145. At this point, the module's flash bus is disconnected with the pin header JP13.

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

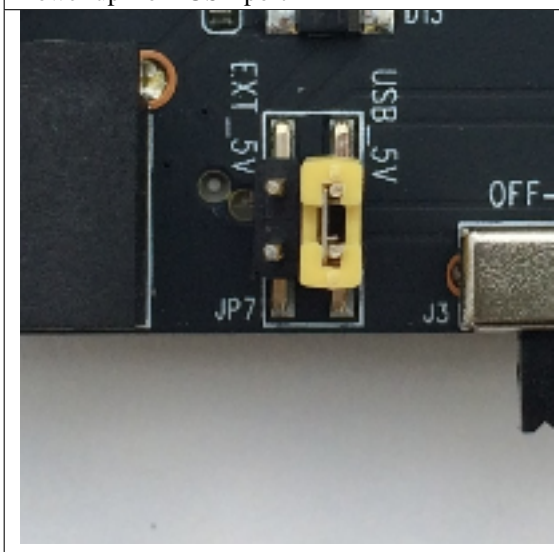
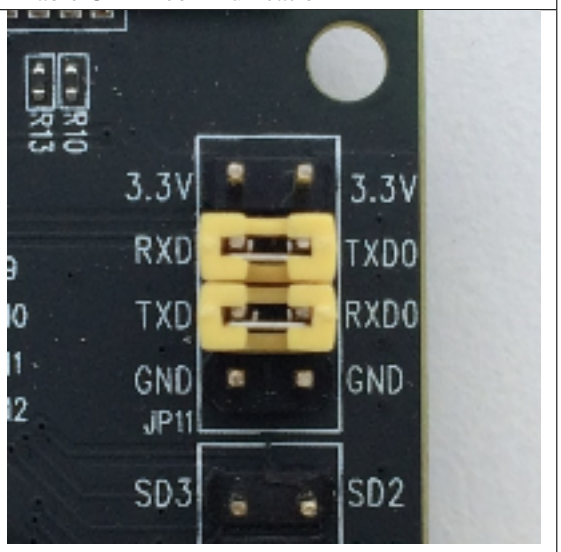
	ESP32 Pin	LCD Signal
1	GPIO18	RESET
2	GPIO19	SCL
3	GPIO21	D/C
4	GPIO22	CS
5	GPIO23	SDA
6	GPIO25	SDO
7	GPIO5	Backlight

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.

Power up from USB port	Enable UART communication
	

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.

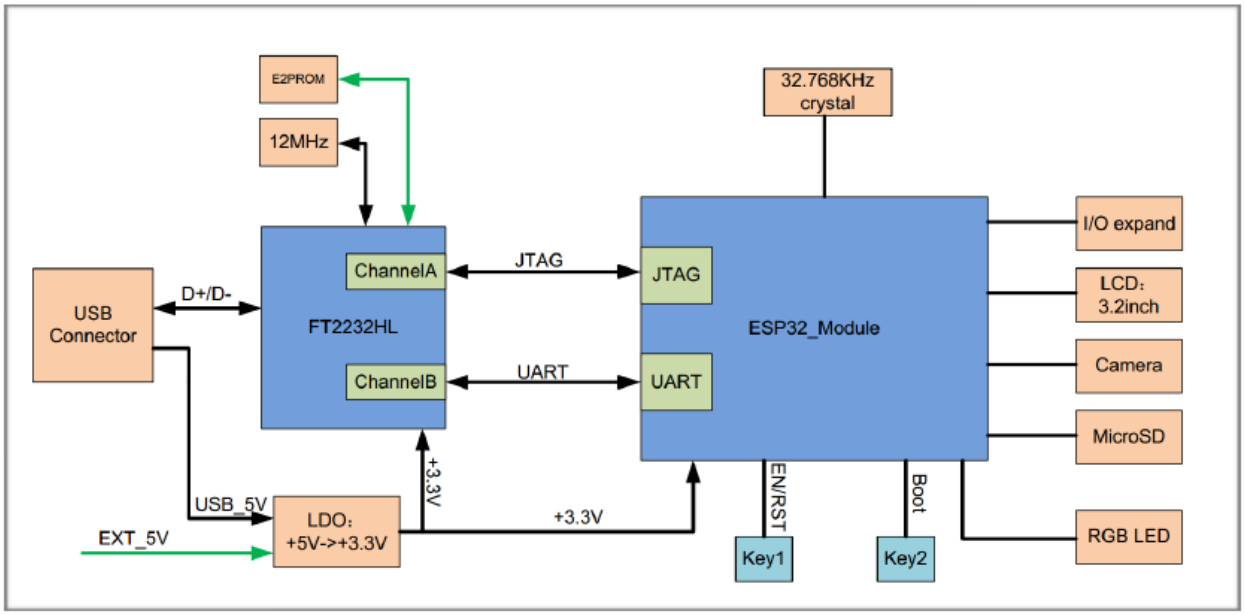


Fig. 8: ESP-WROVER-KIT block diagram

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, uses 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*.

Setup Options

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

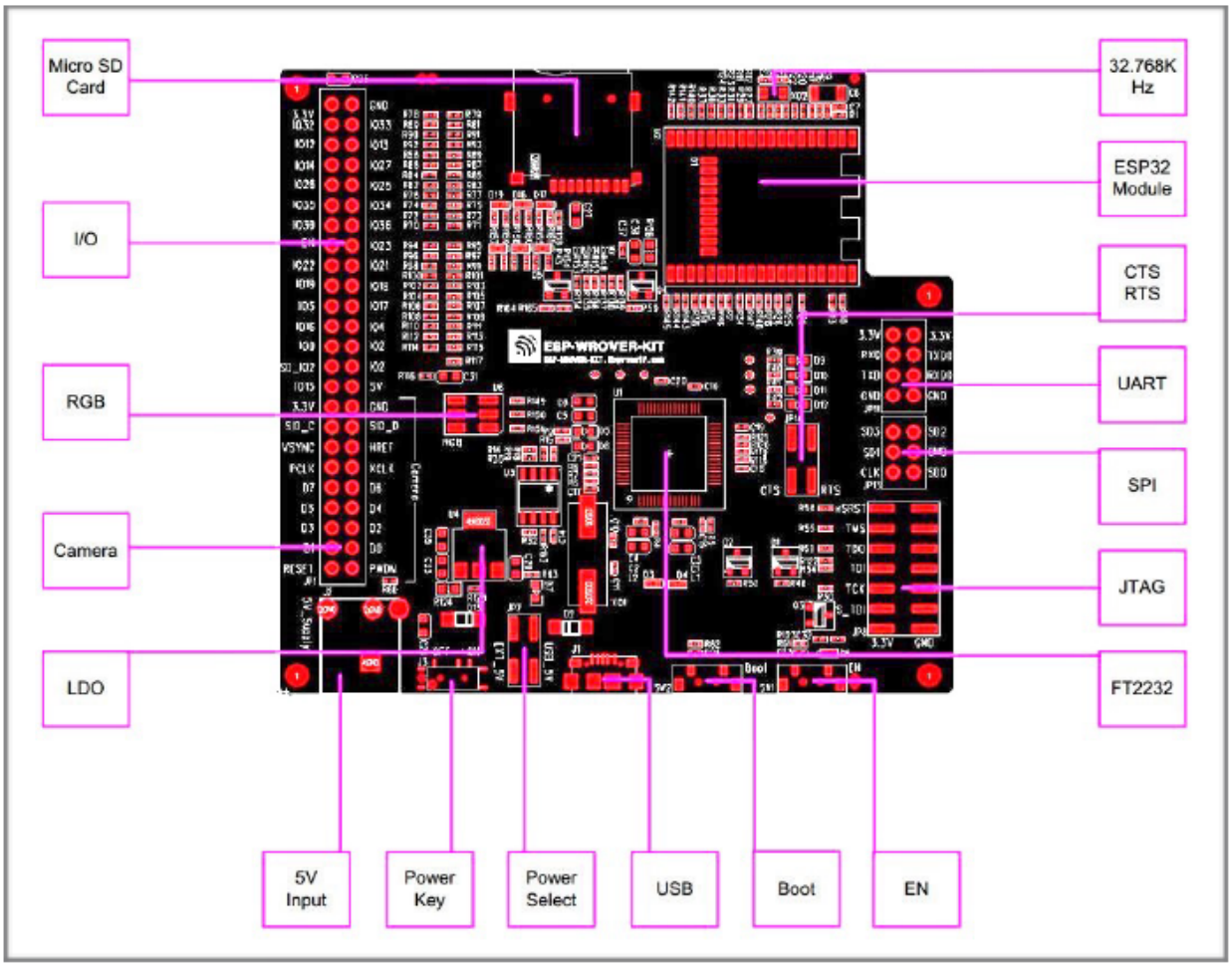


Fig. 9: ESP-WROVER-KIT board layout - front

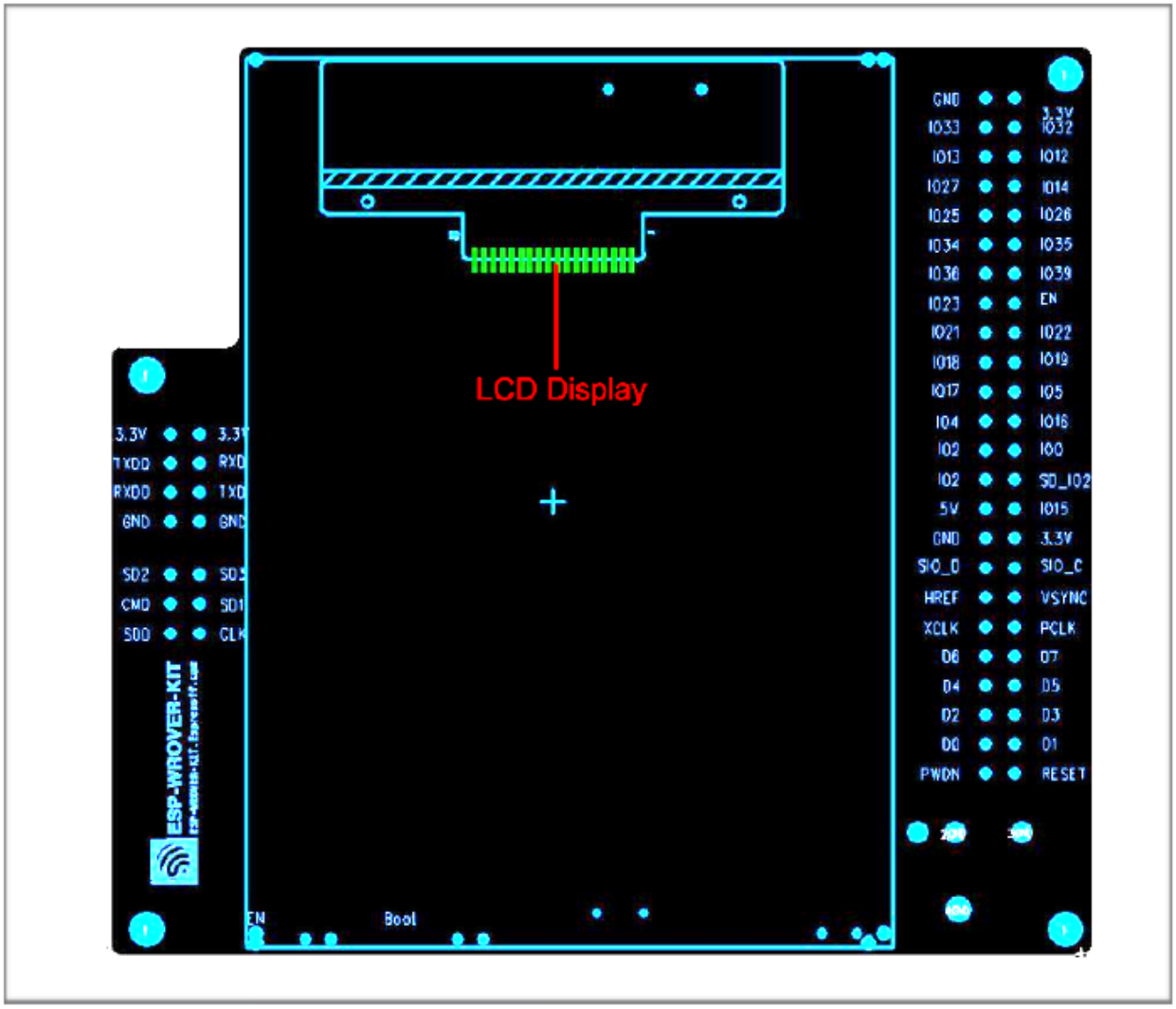
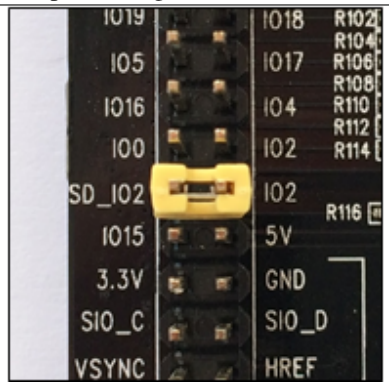
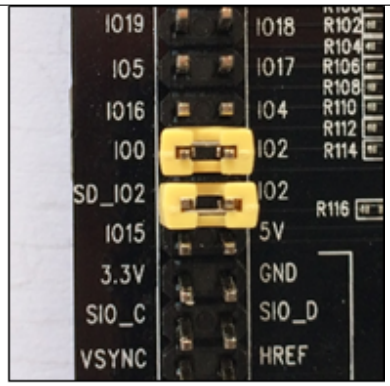
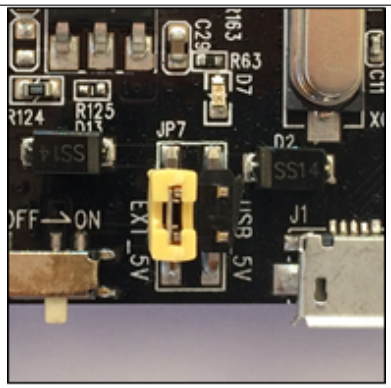
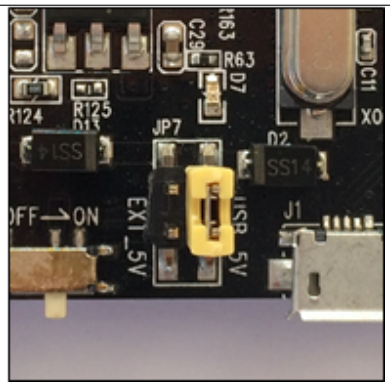
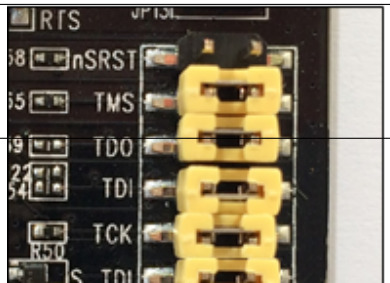


Fig. 10: ESP-WROVER-KIT board layout - back

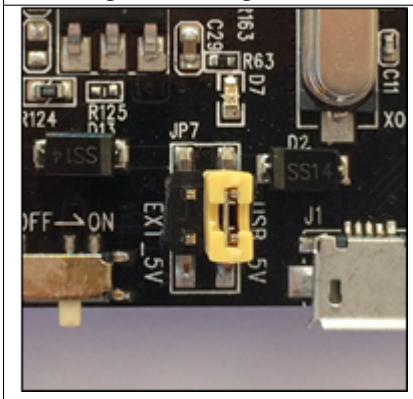
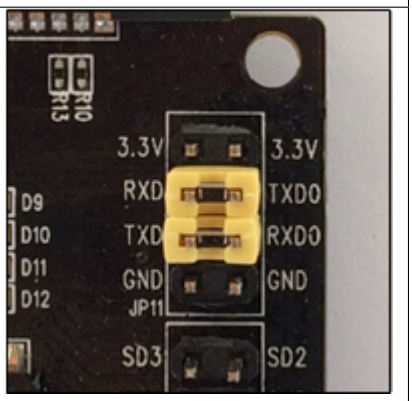
Header	Jumper Setting	Description of Functionality
JP1		Enable pull up for the Micro SD Card
JP1		Assert GPIO2 low during each download (by jumping it to GPIO0)
JP7		Power ESP-WROVER-KIT board from an external power supply
JP7		Power ESP-WROVER-KIT board from an USB port
		

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.

Power up from USB port	Enable UART communication
	

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.

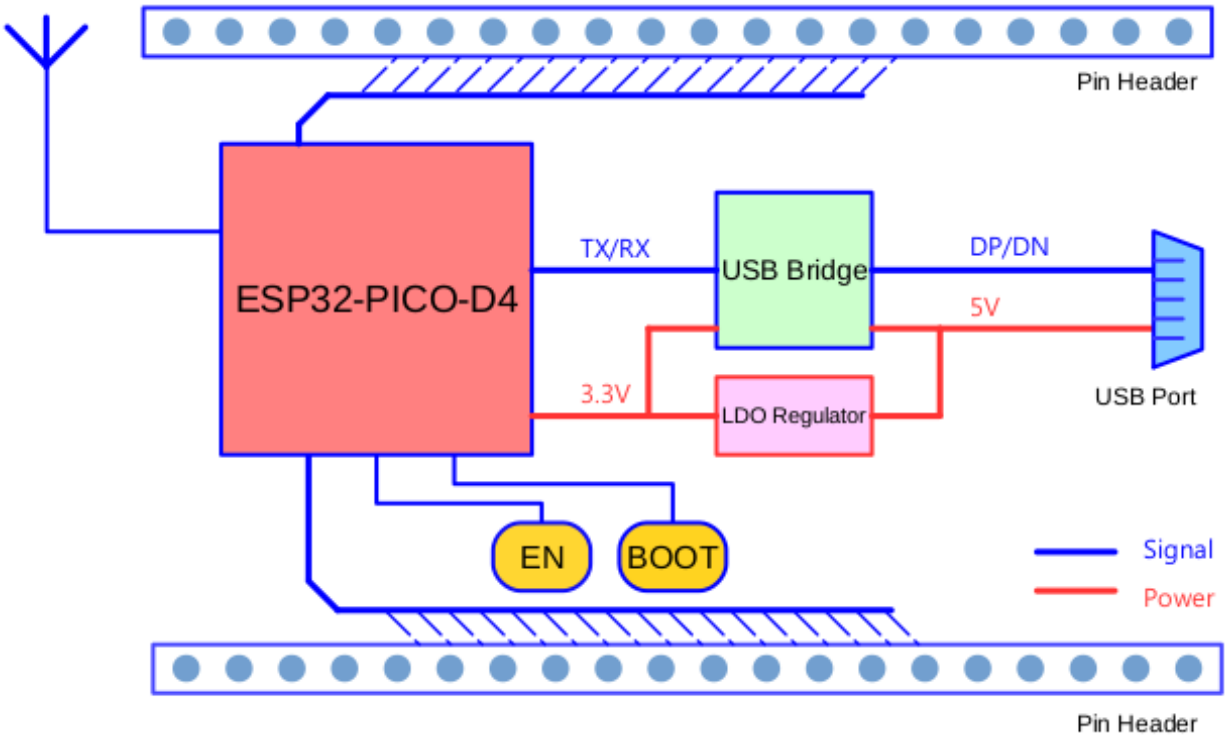


Fig. 11: ESP32-PICO-KIT V4 functional block diagram

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*.

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.

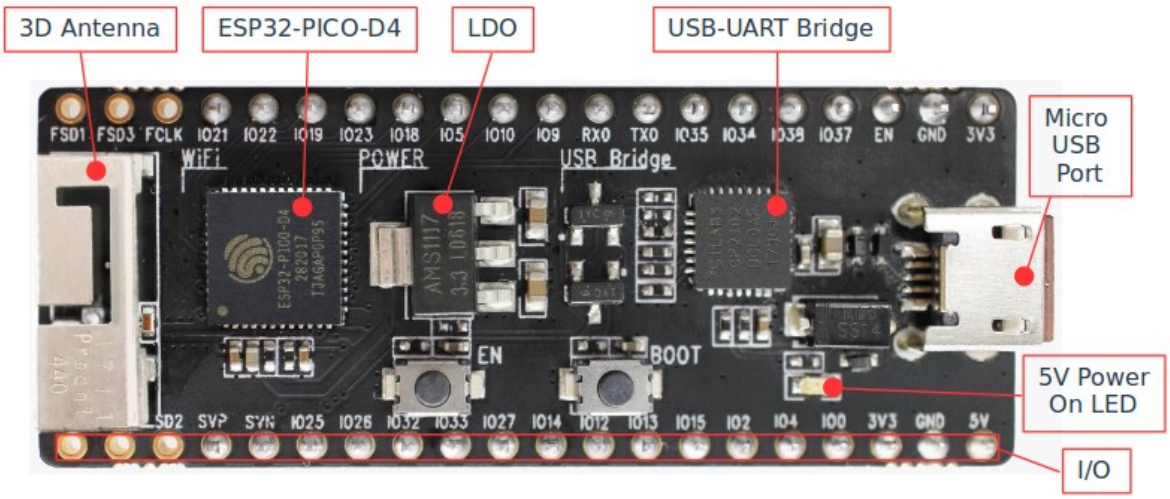


Fig. 12: ESP32-PICO-KIT V4 board layout

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.

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, SPID, HS1_DATA1 *(1)* , U2CTS
2	FLASH_SD3 (FSD3)	I/O	GPIO7, SD_DATA0, SPIQ, HS1_DATA0 *(1)* , U2RTS
3	FLASH_CLK (FCLK)	I/O	GPIO6, SD_CLK, SPICLK, HS1_CLK *(1)* , U1CTS
4	IO21	I/O	GPIO21, VSPIHD, EMAC_TX_EN
5	IO22	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
6	IO19	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
7	IO23	I/O	GPIO23, VSPID, HS1_STROBE
8	IO18	I/O	GPIO18, VSPICLK, HS1_DATA7
9	IO5	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK
10	IO10	I/O	GPIO10, SD_DATA3, SPIWP, HS1_DATA3, U1TXD
11	IO9	I/O	GPIO9, SD_DATA2, SPIHD, HS1_DATA2, U1RXD

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	GPIO11, SD_CMD, SPICS0, HS1_CMD *(1)* , U1RTS
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
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

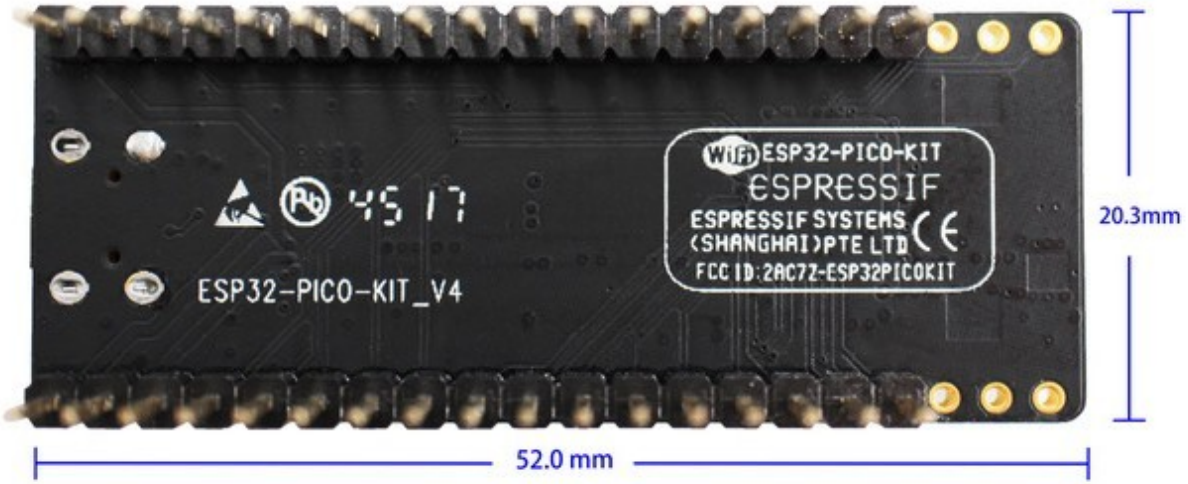


Fig. 13: ESP32-PICO-KIT V4 dimensions - back

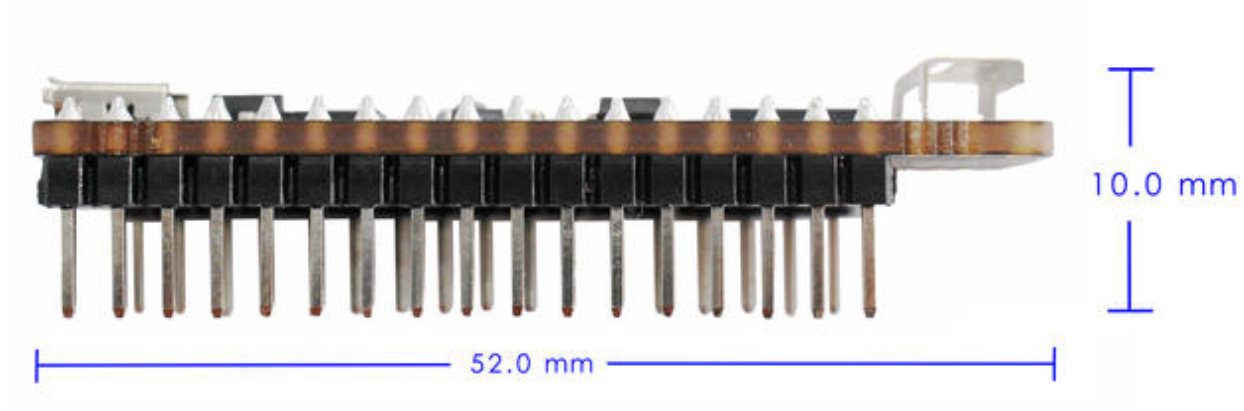


Fig. 14: ESP32-PICO-KIT V4 dimensions - side

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:

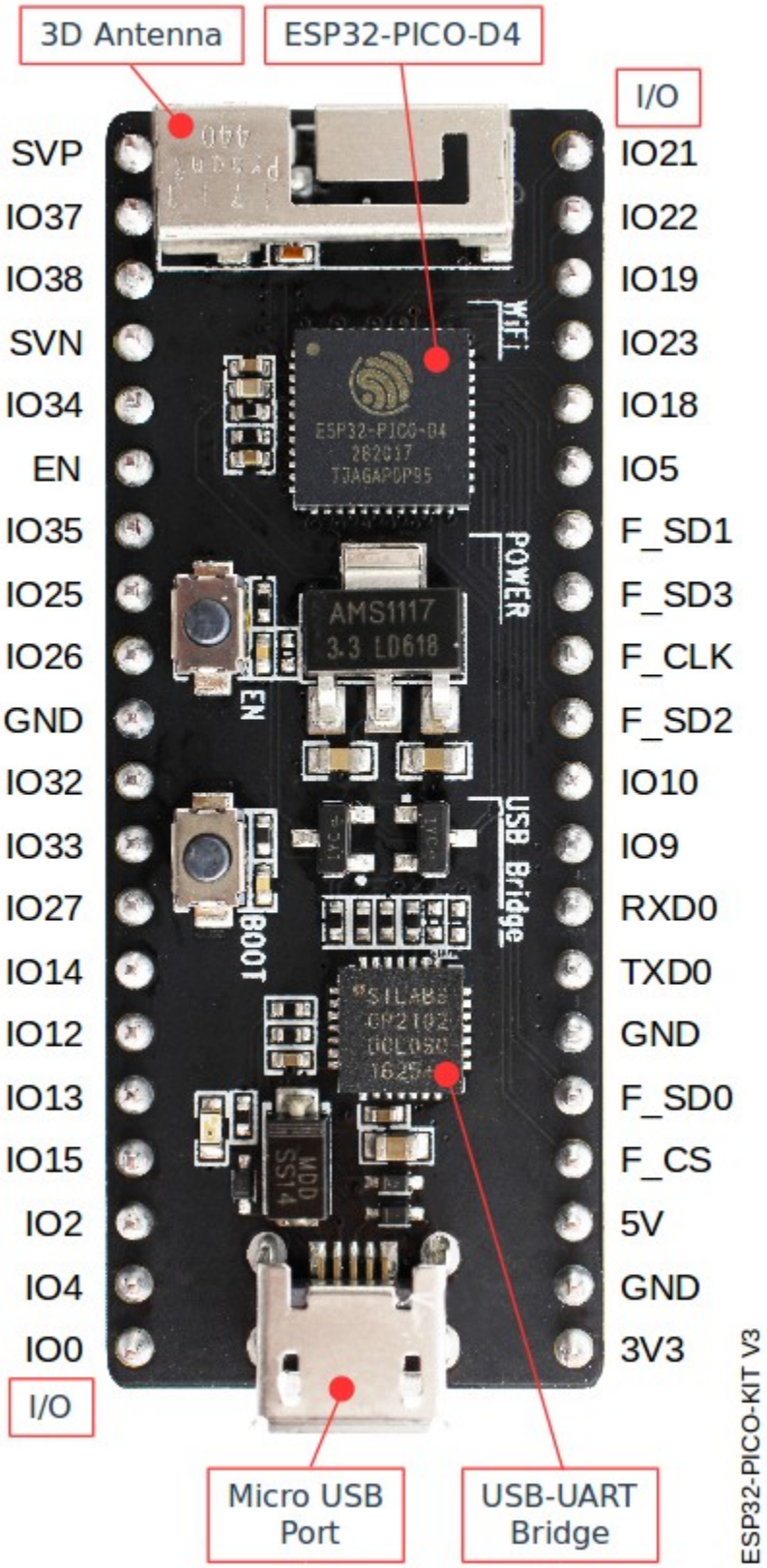


Fig. 15: ESP32-PICO-KIT V3 board layout

- *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/dl/esp32_win32_msys2_environment_and_gcc5_toolchain-20191220.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. Create a directory named `esp` that is a default location to develop ESP32 applications. To do so, run the following shell command:

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

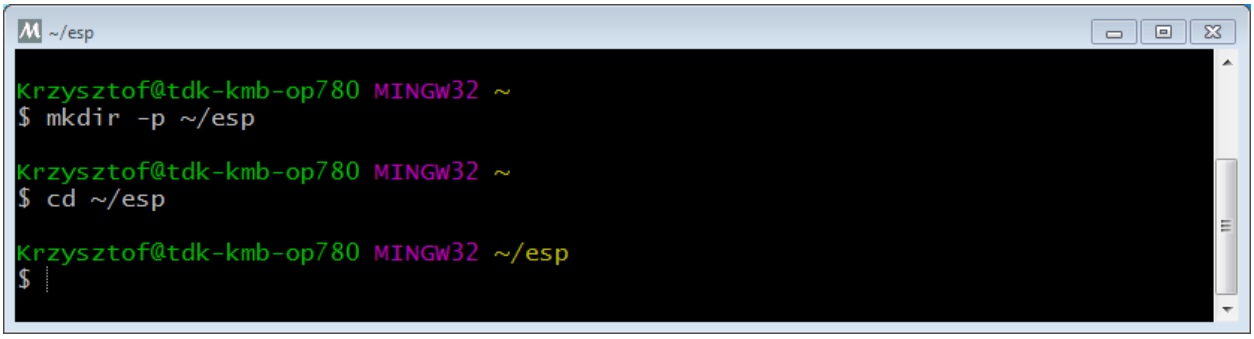


Fig. 16: MSYS2 MINGW32 shell window

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

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

Next Steps

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

Updating The Environment

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

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

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

HTTP Proxy

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

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

Or with credentials:

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

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

Alternative Setup: Just download a toolchain

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

https://dl.espressif.com/dl/xtensa-esp32-elf-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 gcc git wget make ncurses-devel flex bison gperf python pyserial
```

- Ubuntu and Debian:

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

- 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
```

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 gcc git wget make libncurses-dev flex bison gperf python␣
↪python-pip python-setuptools python-serial python-cryptography python-future␣
↪python-pyparsing libffi-dev libssl-dev
```

- Arch:

```
sudo pacman -S --needed gcc git make ncurses flex bison gperf python2-pyserial␣
↪python2-cryptography python2-future python2-pyparsing
```

Compile the Toolchain from Source

- Install dependencies:
  - CentOS 7:

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

  - Ubuntu pre-16.04:

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

  - Ubuntu 16.04 or newer:

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

  - Debian 9:

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

  - Arch:

    ```
    TODO
    ```

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*.

Windows	Linux	Mac OS

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 v3.1.7 --recursive https://github.com/espressif/esp-idf.git
```

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

Note: The `git clone` option `-b v3.1.7` tells git to clone the tag in the ESP-IDF repository corresponding to this version of the documentation.

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

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
```

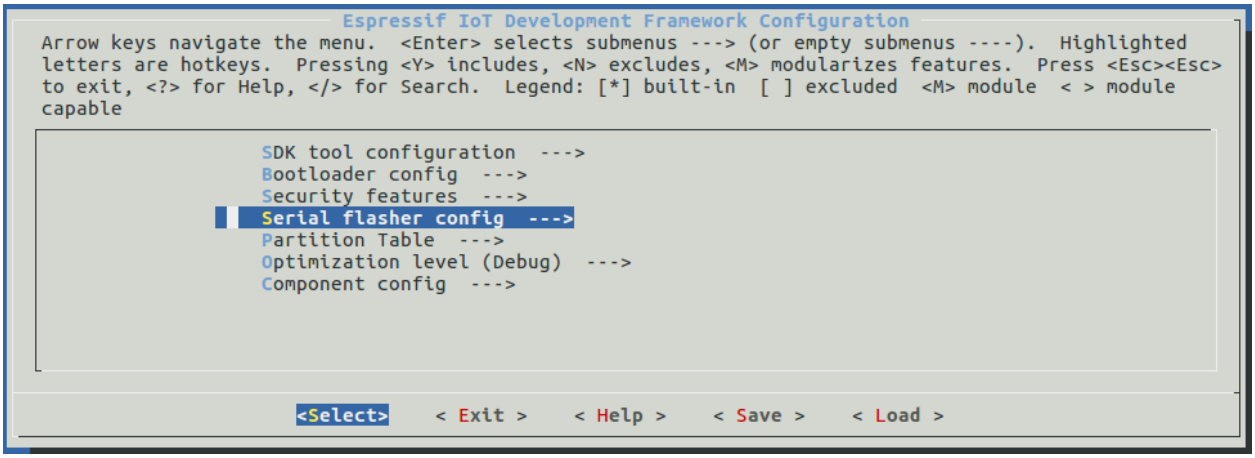


Fig. 17: Project configuration - Home window

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.)

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`.

Attention: When using ESP32-DevKitC board with ESP32-SOLO-1 module, enable single core mode (*`CONFIG_FREERTOS_UNICORE`*) in menuconfig before flashing example applications.

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...
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 `CONFIG_ESP32_XTAL_FREQ_SEL` to 26MHz, then *build and flash* the application again. This is found under `make menuconfig` under Component config –> ESP32-specific –> Main XTAL frequency.

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

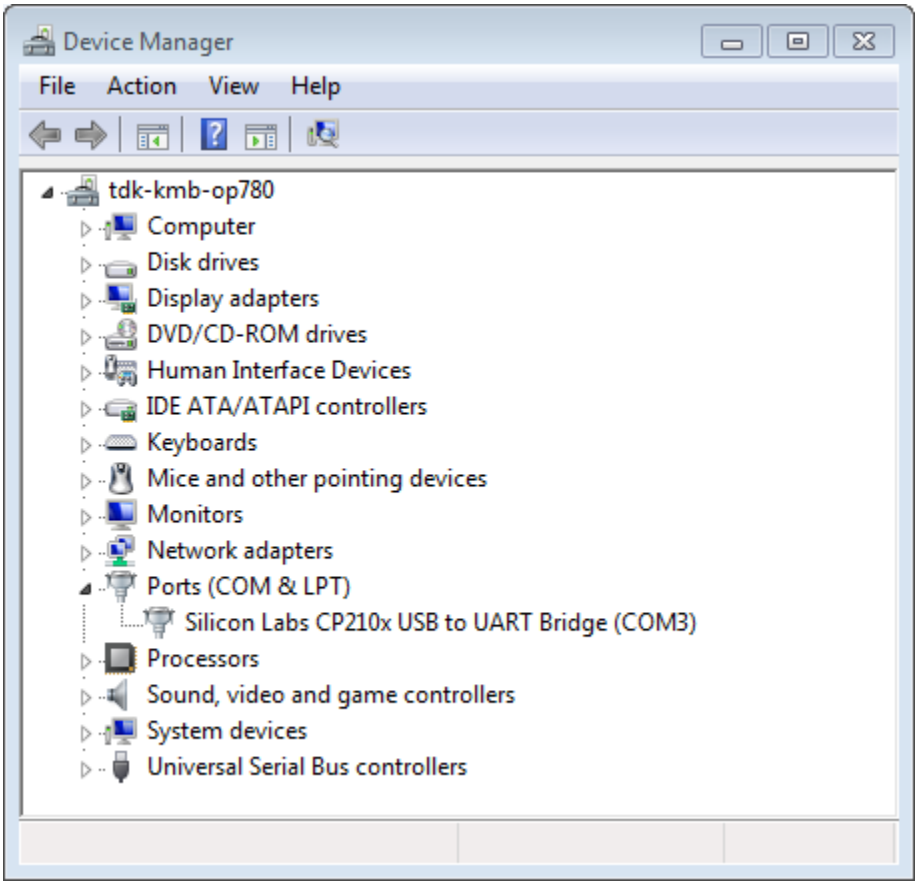


Fig. 18: USB to UART bridge of ESP32-DevKitC in Windows Device Manager

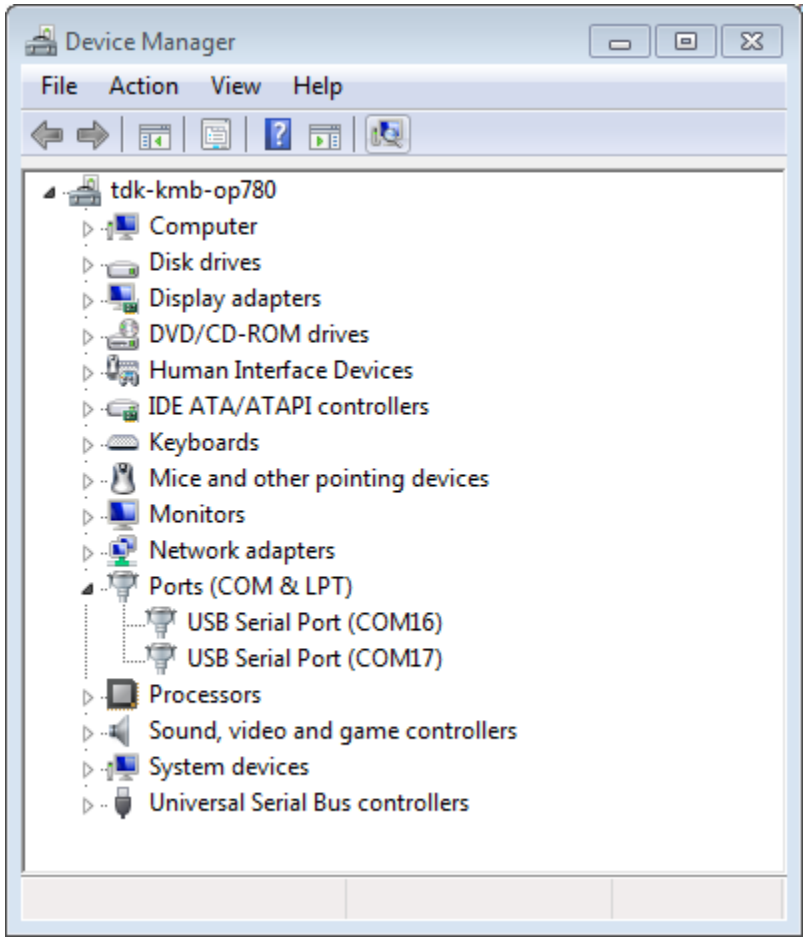


Fig. 19: Two USB Serial Ports of ESP-WROVER-KIT in Windows Device Manager

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.*
```

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.

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

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

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

rst:0x7 (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

...
```

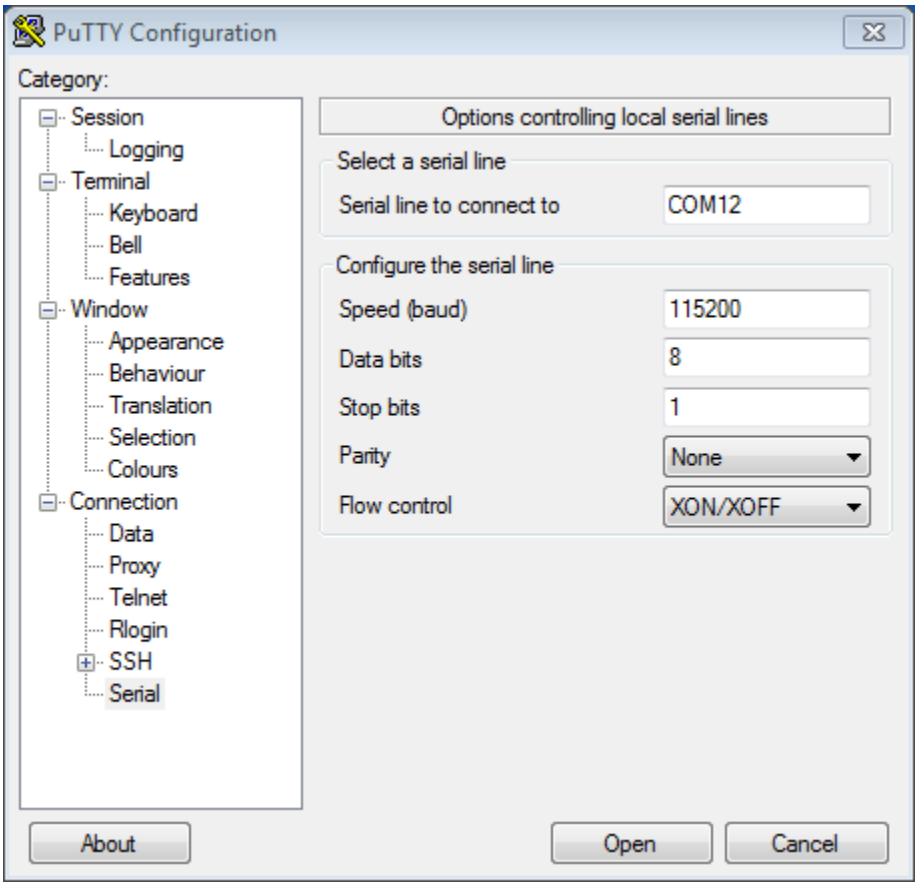


Fig. 20: Setting Serial Communication in PuTTY on Windows

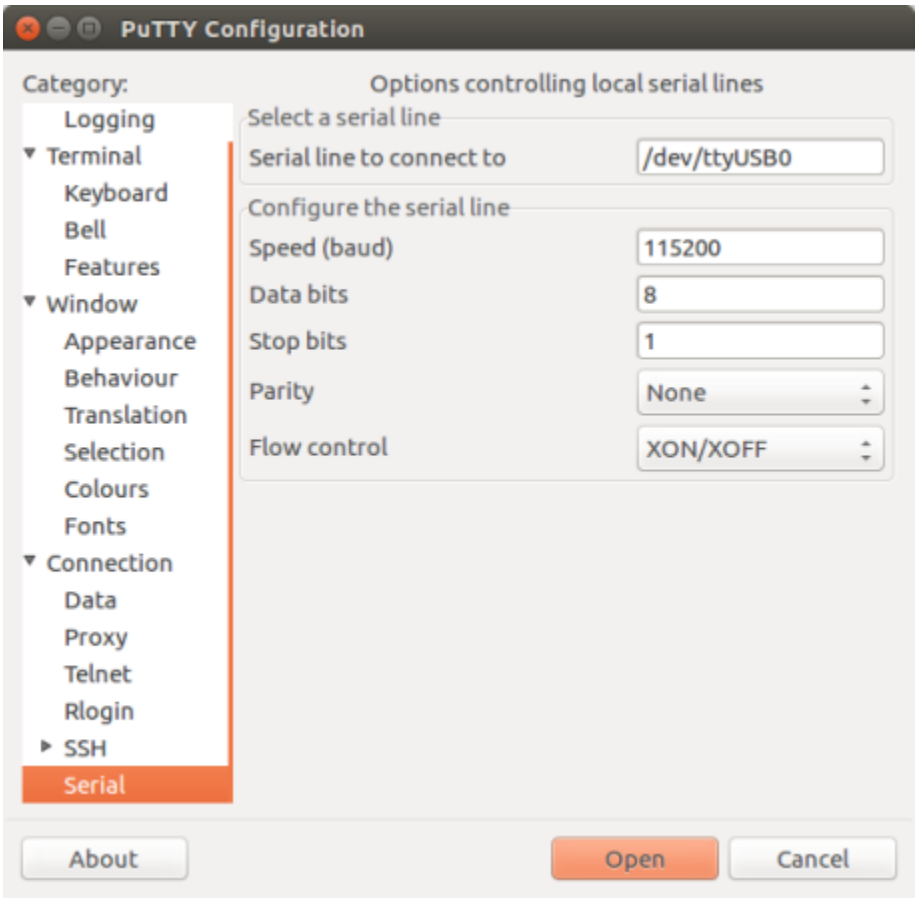


Fig. 21: Setting Serial Communication in PuTTY on Linux

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.
- 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 installed as part of IDF setup, if this is not already listed on the PATH. A typical path to the toolchain looks like `/home/user-name/esp/xtensa-esp32-elf/bin`. Note that you need to add a colon `:` before the appended path.
- 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.

Navigate to "C/C++ Build" -> "Behavior" property page:

- Check "Enable parallel build" to enable multiple build jobs in parallel.

Building in Eclipse

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

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

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

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

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

Flash from Eclipse

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

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

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

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

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/user-name/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.

Navigate to "C/C++ Build" -> "Behavior" property page:

- Check "Enable parallel build" to enable multiple build jobs in parallel.
- Setting the number of jobs slightly higher than the "optimal" may give the absolute fastest builds under Windows, depending on the specific hardware being used.

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 features.

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 *CONFIG_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.

Filtering the Output

The IDF monitor can be invoked as `make monitor PRINT_FILTER=""` with specifying a custom `PRINT_FILTER` option for filtering outputs. The default value is an empty string which means that everything will be printed. Restrictions on what to print can be specified as a series of `<tag>:<log_level>` items where `<tag>` is the tag string and `<log_level>` is a character from set `{N, E, W, I, D, V, *}` referring to a level for *logging*. For example, `PRINT_FILTER="tag1:W"` will match and print (only) the outputs written with `ESP_LOGW("tag1", ...)` or at lower verbosity level, i.e. `ESP_LOGE("tag1", ...)`. Not specifying a `<log_level>` or using `*` defaults to Verbose level.

Note: The primary logging is set up at compilation time through the *logging library*. Output filtering by the IDF monitor is only a secondary solution because one cannot filter something which has been disabled at compilation time. The advantage of the secondary filtering is that one can use various filtering options without recompiling the application.

A restriction applies to tags when one wants to use them together with output filtering: they cannot contain spaces, asterisks `*` and semicolons `:`.

If the last line of the output is written without an end of line then the output filtering might get confused, i.e. the monitor starts to print the line and only later finds out that the line should have not been written. This is a known issue and can be avoided by always adding an end of line after printing something (especially when no output follows immediately afterwards).

Examples Of Filtering Rules:

- Asterisk can be used to match any tags. However, specifying `PRINT_FILTER="*:I tag1:E"` will print for `tag1` only errors because the rule for `tag1` has a precedence over the rule for `*`.
- The default (empty) rule is equivalent to `*:V` because matching every tag at level Verbose or lower means matching everything.
- Rule `"tag1:W tag1:E"` is equivalent to `"tag1:E"` because any consequent occurrence of the same tag name overwrites the previous one.
- Rule `"tag1:I tag2:W"` will print only `tag1` at verbosity level Info or lower and `tag2` at verbosity level Warning or lower.
- Rule `"tag1:I tag2:W tag3:N"` is essentially equivalent to the previous one because `tag3:N` specifies that `tag3` should not be printed.
- `tag3:N` in rule `"tag1:I tag2:W tag3:N *:V"` is more meaningful because in this context the result will be that `tag3` will not be printed, `tag1` and `tag2` will be at the specified (or lower) verbosity level and everything else will be printed by default.
- `"*:N"` will suppress all outputs even prints made by something else than the logging functions, e.g. `printf`. For printing those outputs one need to use `*:E` or higher verbosity level.
- Rules `"tag1:V"`, `"tag1:v"`, `"tag1:"`, `"tag1:*"` and `"tag1"` are all equivalent ones.

A More Complex Filtering Example

The following log snippet was acquired using `make monitor`:

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

The captured output for `make monitor PRINT_FILTER="wifi esp_image:E light_driver:I"` is the following:

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

`make monitor PRINT_FILTER="light_driver:D esp_image:N boot:N cpu_start:N vfs:N wifi:N *:V"` gives the following output:

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

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.

CHAPTER 2

Get Started (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

Important: The following features are not yet supported with the CMake-based build system:

- Eclipse IDE Documentation
- Secure Boot
- Flash Encryption
- ULP toolchain.

Support for these features will be available before CMake becomes the default build system.

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.

2.1 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.

2.2 What You Need

To develop applications for ESP32 you need:

- **PC** loaded with either Windows, Linux or Mac operating system
- **Toolchain** to compile code for ESP32
- **Build tools** CMake and Ninja to build a full **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**

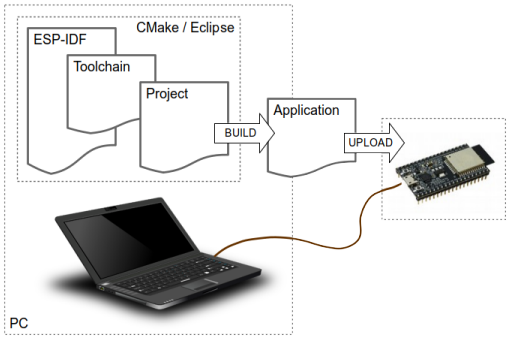


Fig. 1: Development of applications for ESP32

Steps to set up Development Environment:

1. Setup of **Toolchain**
2. Getting **ESP-IDF** from GitHub

Once the development environment is set up, we will follow these steps to create an ESP-IDF application:

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

2.3 Development Board Guides

If you have one of ESP32 development boards listed below, click on the link for hardware setup:

2.3.1 ESP32-DevKitC V4 Getting Started Guide (CMake)

This user guide shows how to get started with ESP32-DevKitC V4 development board. For description of other versions of the ESP32-DevKitC check *ESP32 Hardware Reference*.

What You Need

- 1 × *ESP32-DevKitC V4 board*
- 1 × USB A / micro USB B cable
- 1 × PC loaded with Windows, Linux or Mac OS

Overview

ESP32-DevKitC V4 is a small-sized ESP32-based development board produced by Espressif. Most of the I/O pins are broken out to the pin headers on both sides for easy interfacing. Developers can connect these pins to peripherals as needed. Standard headers also make development easy and convenient when using a breadboard.

The board comes in two versions, either with *ESP-WROOM-32* or *ESP32-WROVER* module soldered.

Functional Description

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

ESP-WROOM-32 *ESP-WROOM-32* module soldered to the ESP32-DevKitC V4 board.

ESP32-WROVER Optionally *ESP32-WROVER* module may be soldered instead of the ESP-WROOM-32.

USB-UART Bridge A single chip USB-UART bridge provides up to 3 Mbps transfers rates.

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.

Micro USB Port USB interface. It functions as the power supply for the board and the communication interface between PC and the ESP module.

5V Power On LED This LED lights when the USB or an external 5V power supply is applied to the board. For details see schematic in *Related Documents*.

EN Reset button: pressing this button resets the system.

I/O Most of the pins on the ESP module 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.

> **Note:** Some of broken out pins are used internally be the ESP32 module to communicate with SPI memory. They are grouped on one side of the board besides the USB connector and labeled D0, D1, D2, D3, CMD and CLK. In general these pins should be left unconnected or access to the SPI flash memory / SPI RAM may be disturbed.

> **Note:** GPIO16 and 17 are used internally by the ESP32-WROVER module. They are broken out and avialable for use only for boards that have the ESP-WROOM-32 module installed.

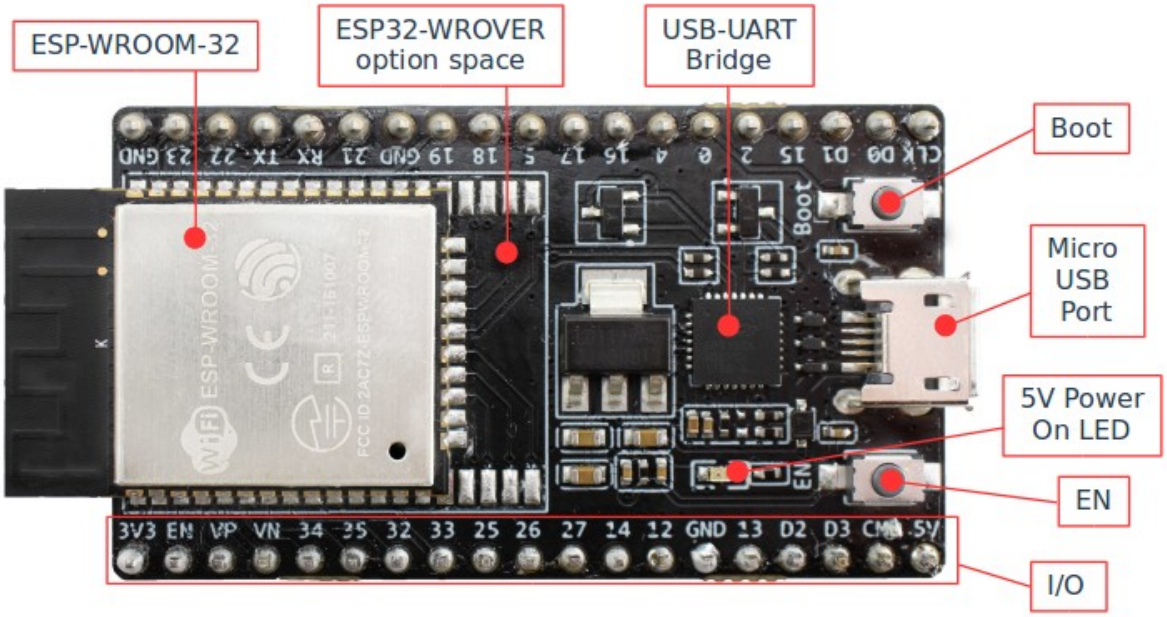


Fig. 2: ESP32-DevKitC V4 with ESP-WROOM-32 module soldered

Power Supply Options

There following options are available to provide power supply to this board:

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 (CMake)*, 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 The Project* for an example application
- *Flash To A Device* to run code on the ESP32
- *Monitor* instantly what the application is doing

Board Dimensions

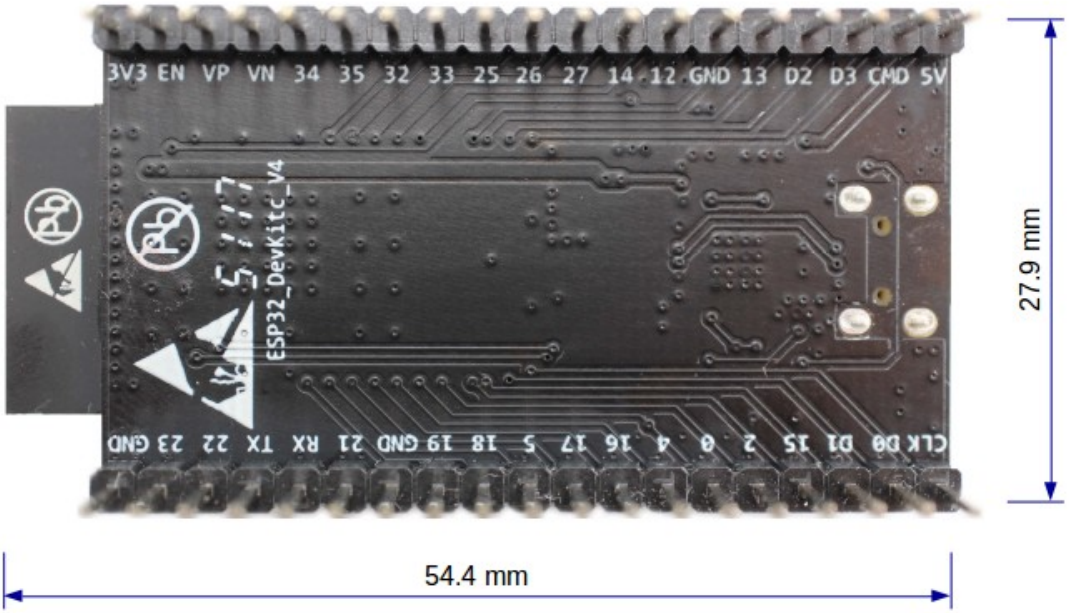


Fig. 3: ESP32 DevKitC board dimensions - back

Related Documents

- ESP32-DevKitC V4 schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP-WROOM-32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)

ESP32-DevKitC V2 Getting Started Guide (CMake)

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

What You Need

- 1 × *ESP32-DevKitC V2 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.

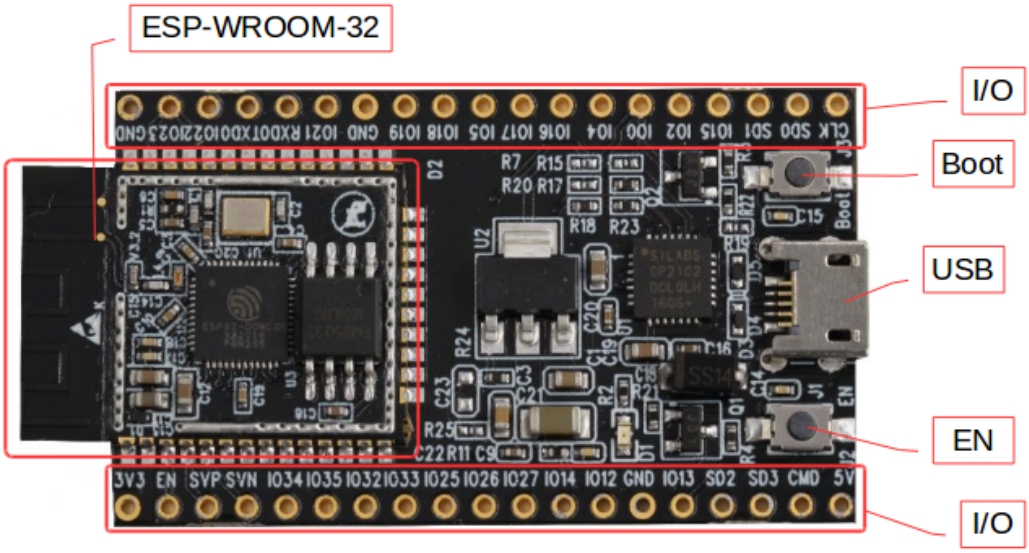


Fig. 4: ESP32-DevKitC V2 board layout

Power Supply Options

There following options are available to provide power supply to this board:

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 (CMake)*, 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 The Project* for an example application
- *Flash To A Device* to run code on the ESP32
- *Monitor* instantly what the application is doing

Related Documents

- ESP32-DevKitC schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP-WROOM-32 Datasheet (PDF)

2.3.2 ESP-WROVER-KIT V3 Getting Started Guide (CMake)

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.

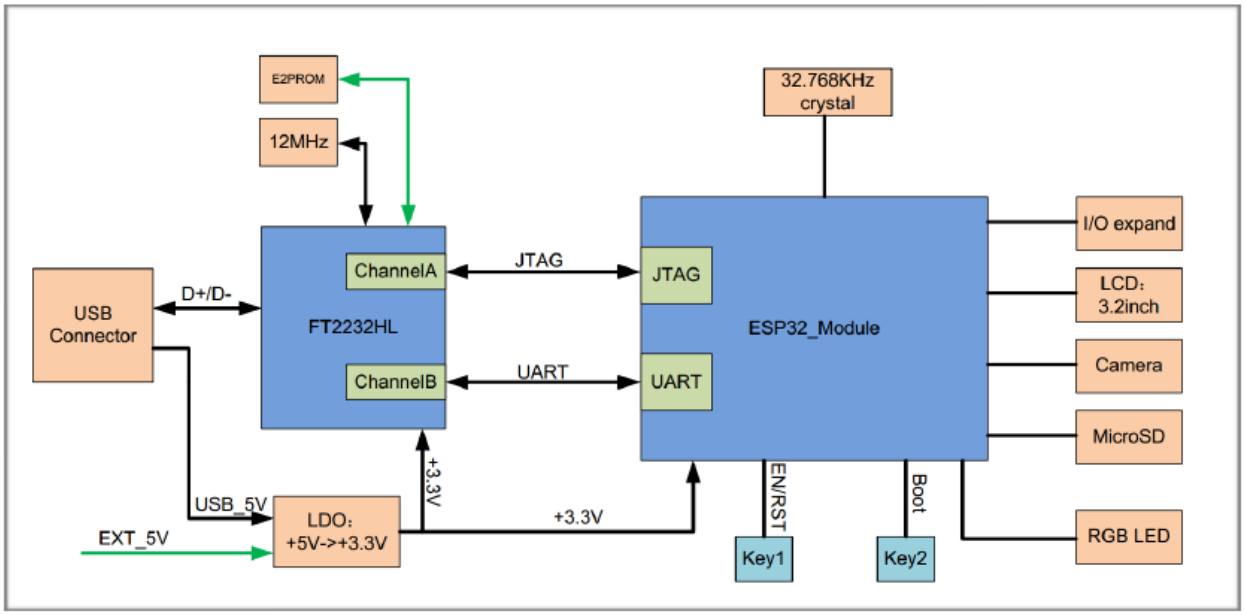


Fig. 5: ESP-WROVER-KIT block diagram

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*.

Setup Options

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

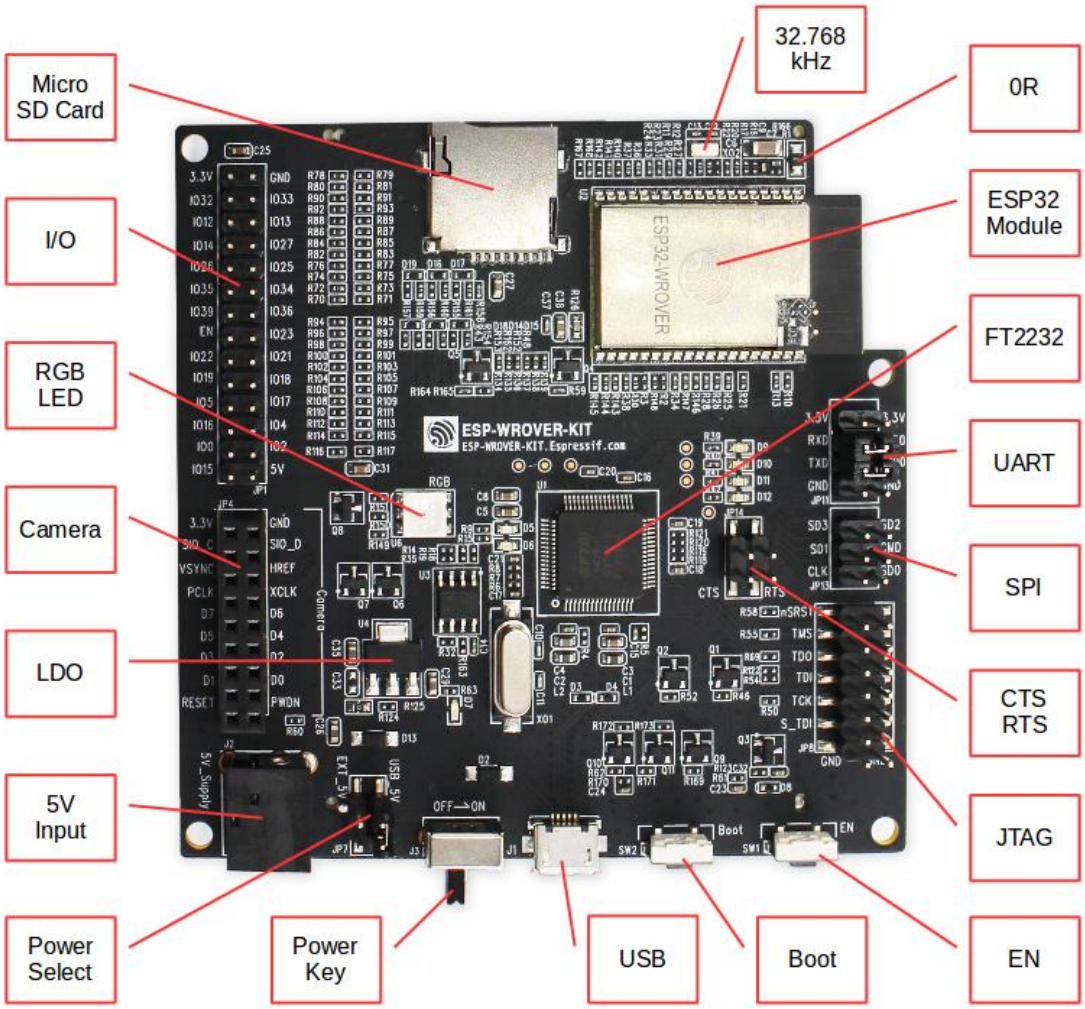


Fig. 6: ESP-WROVER-KIT board layout - front

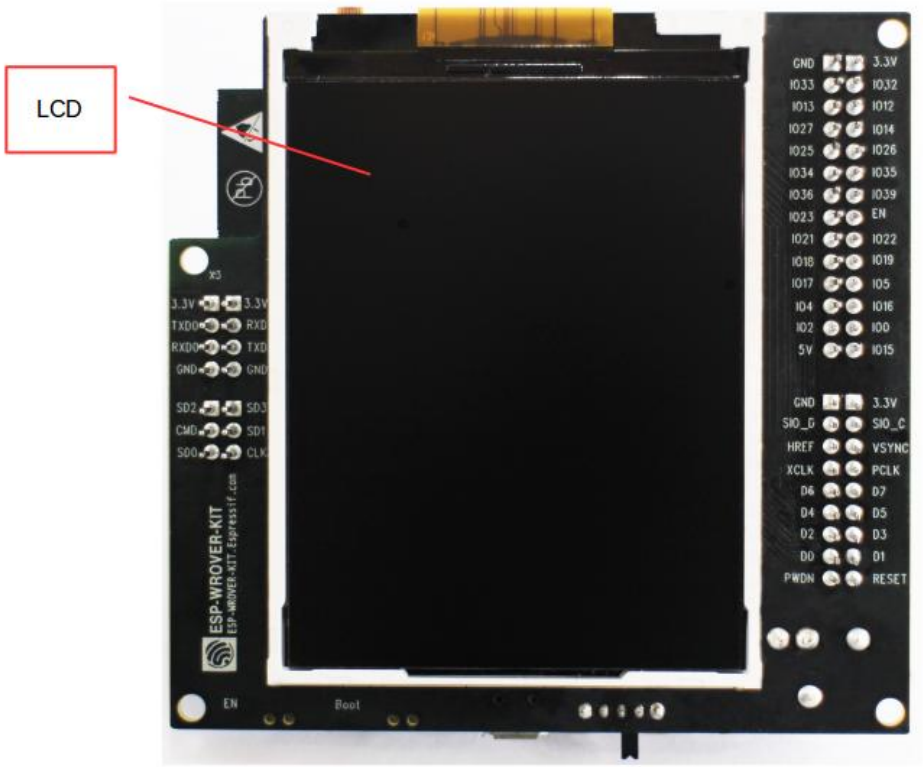
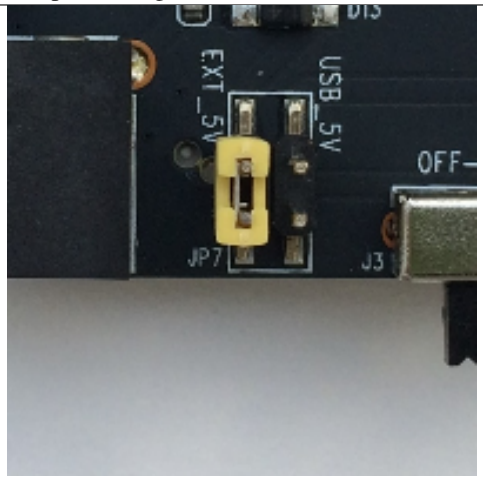
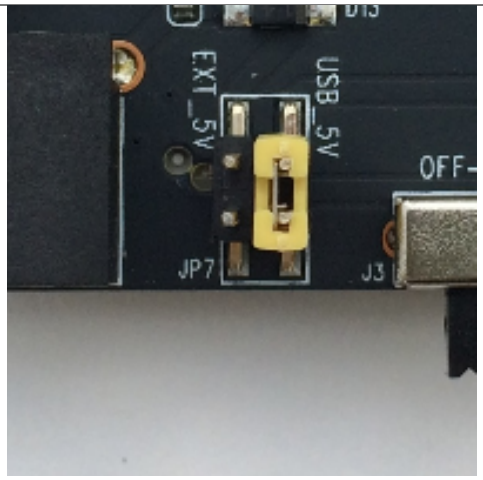
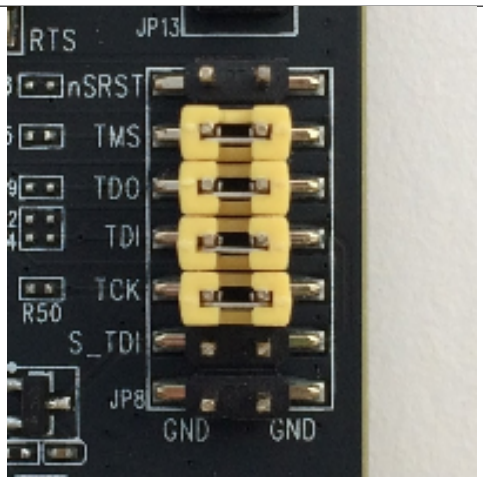
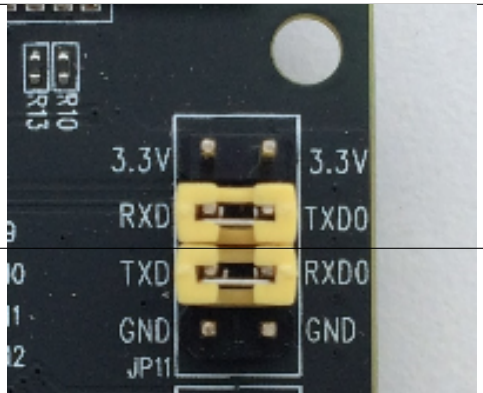


Fig. 7: ESP-WROVER-KIT board layout - back

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
		

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

Important: The module's flash bus is connected to the pin header JP13 through 0-Ohm resistors R140 ~ R145. If the flash frequency needs to operate at 80 MHz, to improve integrity of the bus signals, it is recommended to desolder resistors R140 ~ R145. At this point, the module's flash bus is disconnected with the pin header JP13.

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

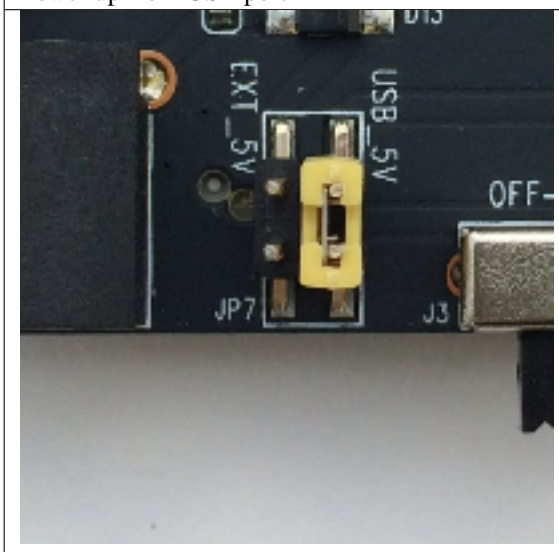
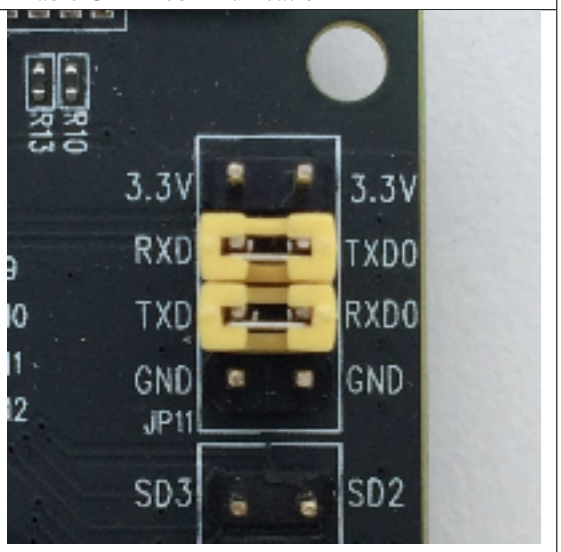
	ESP32 Pin	LCD Signal
1	GPIO18	RESET
2	GPIO19	SCL
3	GPIO21	D/C
4	GPIO22	CS
5	GPIO23	SDA
6	GPIO25	SDO
7	GPIO5	Backlight

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.

Power up from USB port	Enable UART communication
	

Do not install any other jumpers.

Now to Development

To start development of applications for ESP-WROVER-KIT, proceed to section *Get Started (CMake)*, 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 The Project* for an example application
- *Flash To A Device* to run code on 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 (CMake)

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.

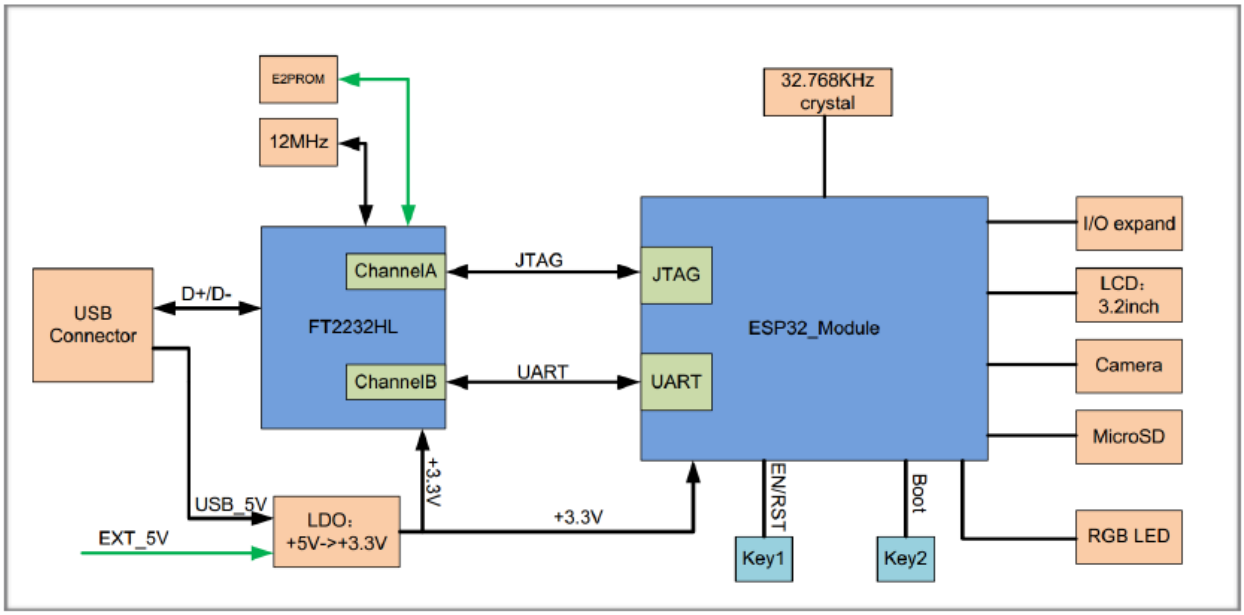


Fig. 8: ESP-WROVER-KIT block diagram

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, uses 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*.

Setup Options

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

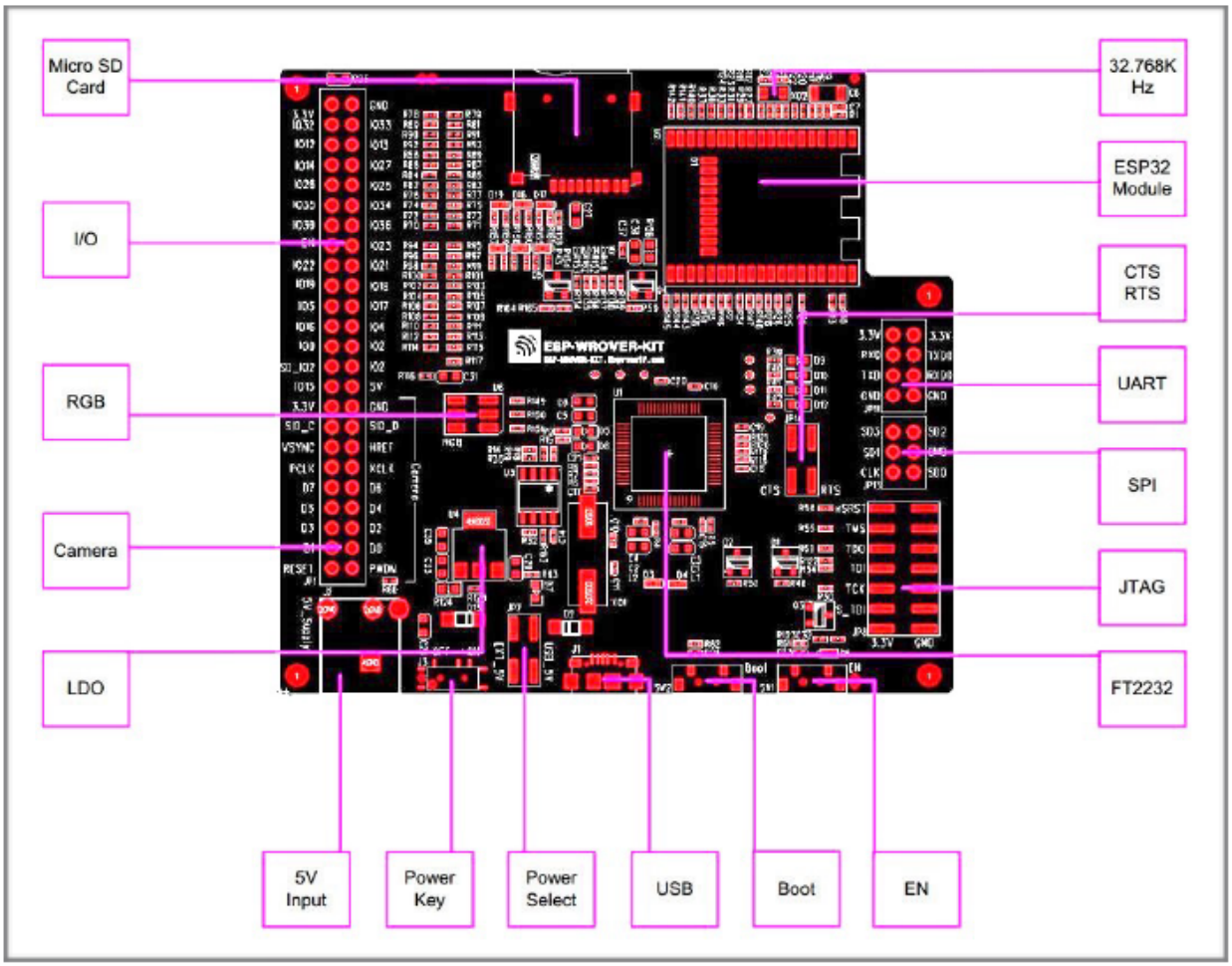


Fig. 9: ESP-WROVER-KIT board layout - front

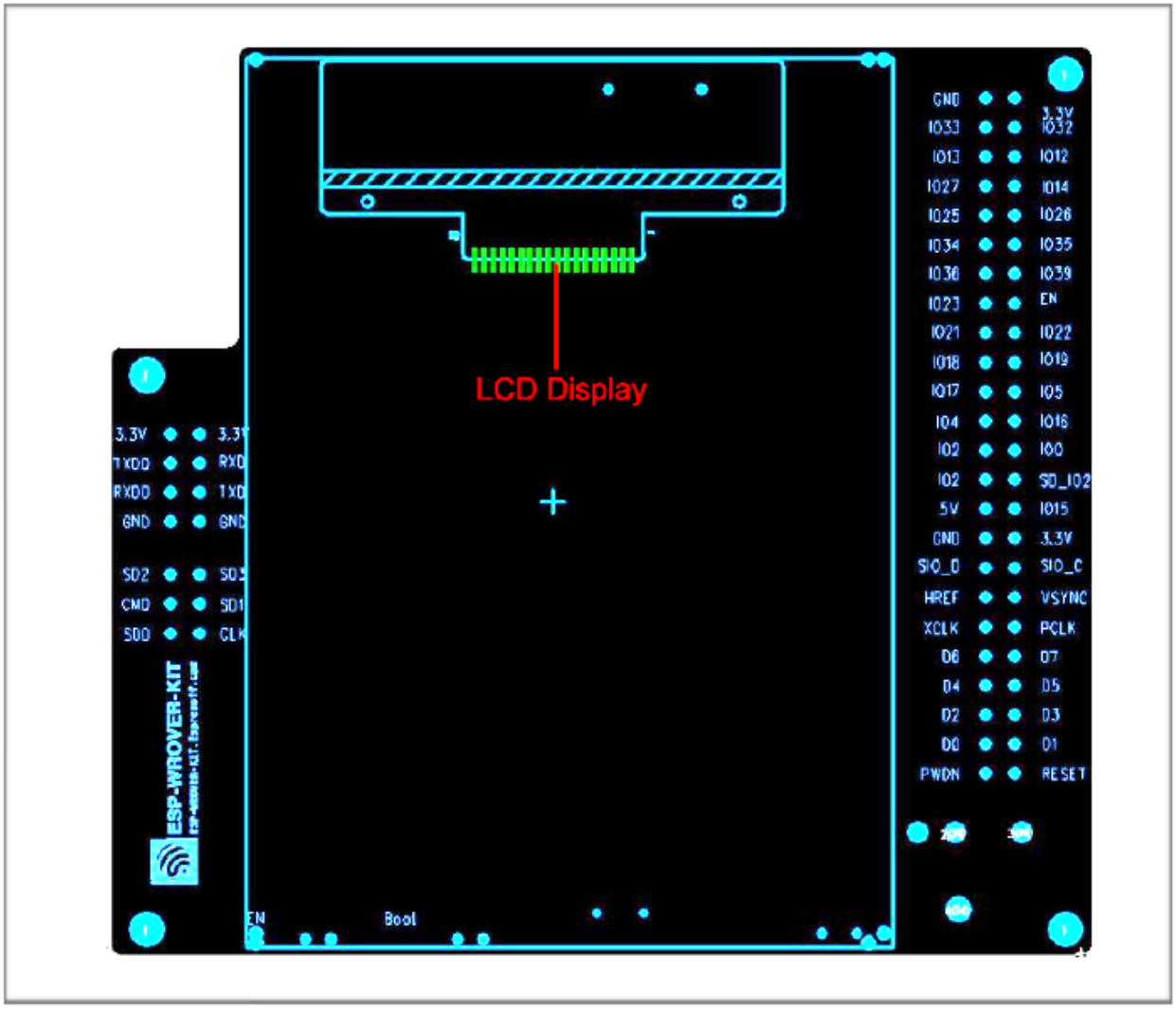
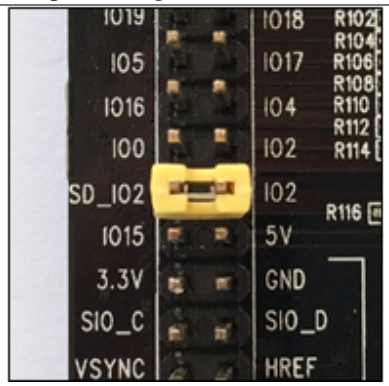
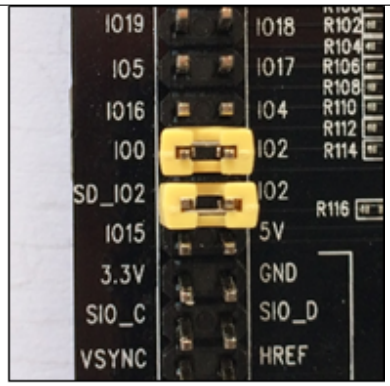
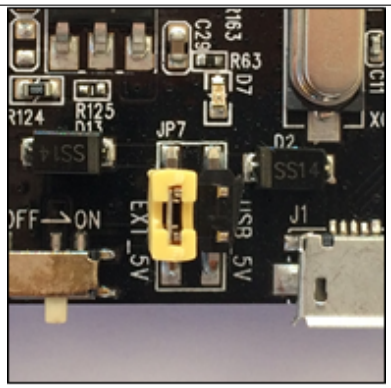
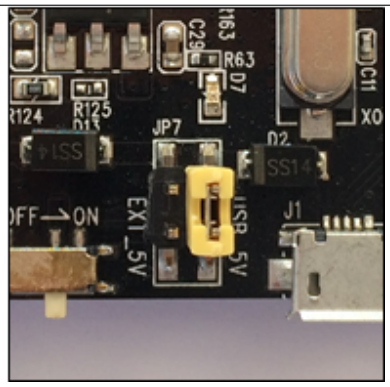
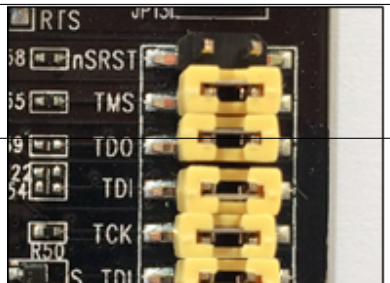


Fig. 10: ESP-WROVER-KIT board layout - back

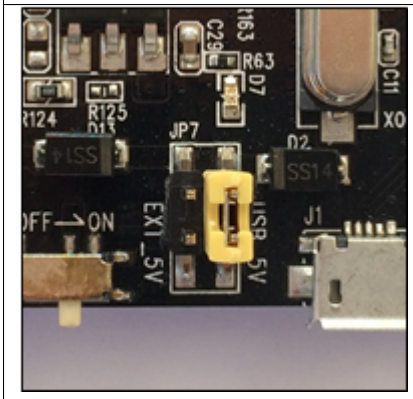
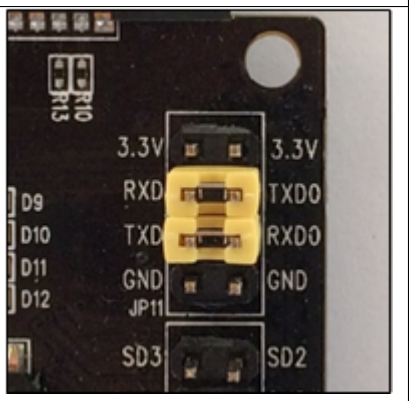
Header	Jumper Setting	Description of Functionality
JP1		Enable pull up for the Micro SD Card
JP1		Assert GPIO2 low during each download (by jumping it to GPIO0)
JP7		Power ESP-WROVER-KIT board from an external power supply
JP7		Power ESP-WROVER-KIT board from an USB port
		

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.

Power up from USB port	Enable UART communication
	

Do not install any other jumpers.

Now to Development

To start development of applications for ESP32-DevKitC, proceed to section *Get Started (CMake)*, 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 The Project* for an example application
- *Flash To A Device* to run code on 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*

2.3.3 ESP32-PICO-KIT V4 Getting Started Guide (CMake)

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*.

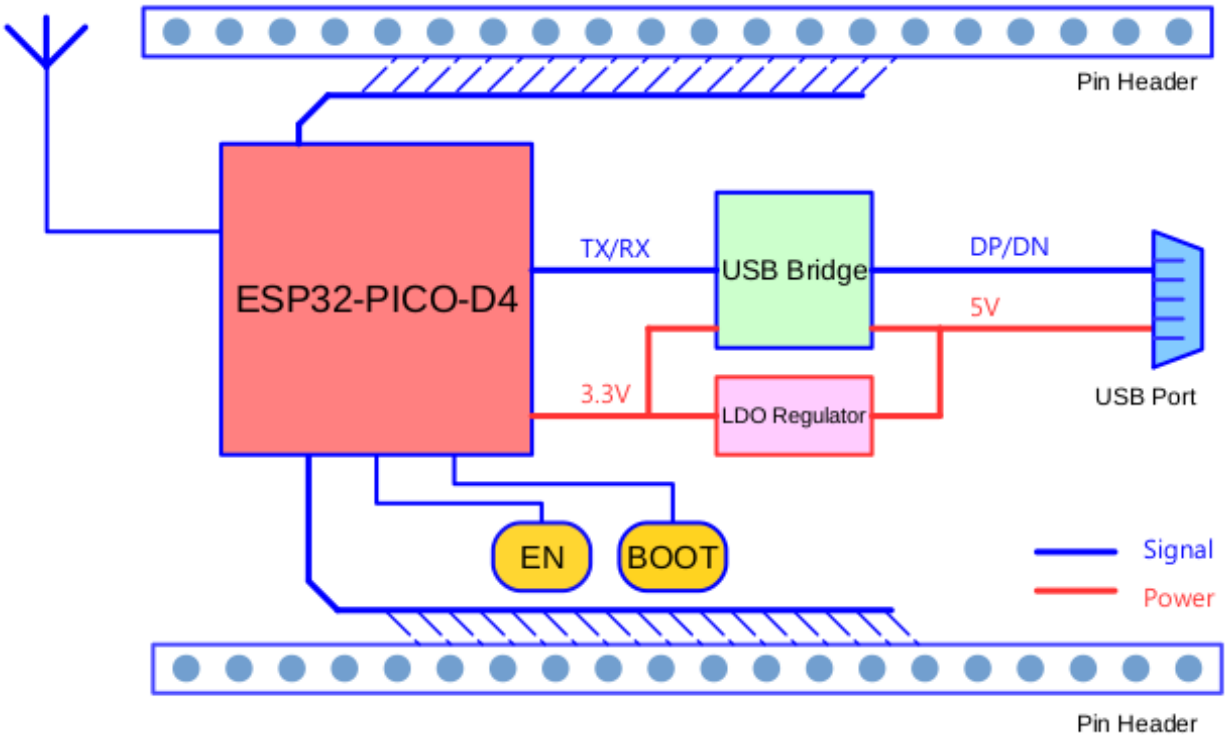


Fig. 11: ESP32-PICO-KIT V4 functional block diagram

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.

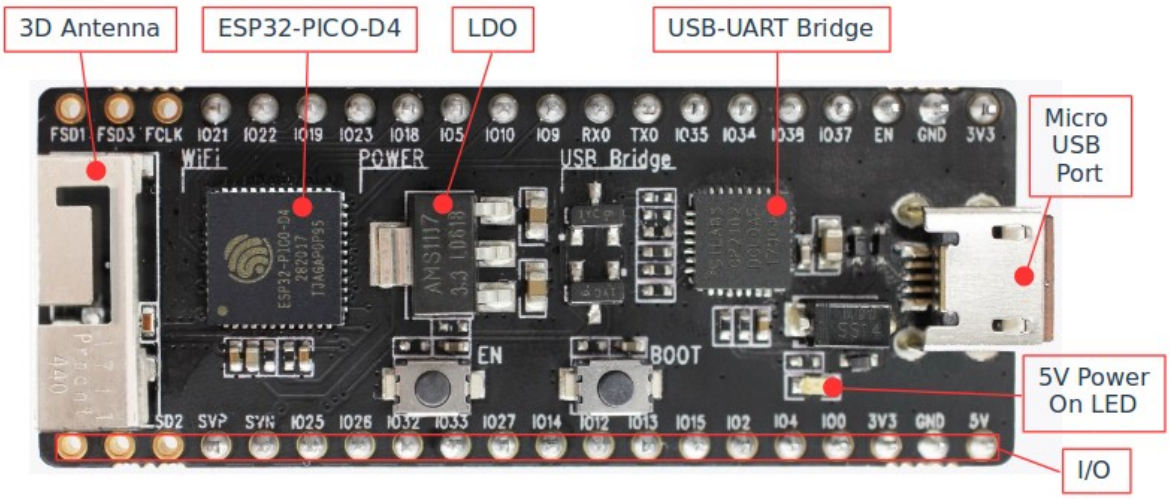


Fig. 12: ESP32-PICO-KIT V4 board layout

To start development of applications, proceed to section *Get Started (CMake)*, 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 The Project* for an example application
- *Flash To A Device* to run code on 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, SPID, HS1_DATA1 *(1)* , U2CTS
2	FLASH_SD3 (FSD3)	I/O	GPIO7, SD_DATA0, SPIQ, HS1_DATA0 *(1)* , U2RTS
3	FLASH_CLK (FCLK)	I/O	GPIO6, SD_CLK, SPICLK, HS1_CLK *(1)* , U1CTS
4	IO21	I/O	GPIO21, VSPIHD, EMAC_TX_EN
5	IO22	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
6	IO19	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
7	IO23	I/O	GPIO23, VSPID, HS1_STROBE
8	IO18	I/O	GPIO18, VSPICLK, HS1_DATA7
9	IO5	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK
10	IO10	I/O	GPIO10, SD_DATA3, SPIWP, HS1_DATA3, U1TXD
11	IO9	I/O	GPIO9, SD_DATA2, SPIHD, HS1_DATA2, U1RXD

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	GPIO11, SD_CMD, SPICS0, HS1_CMD *(1)* , U1RTS
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
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

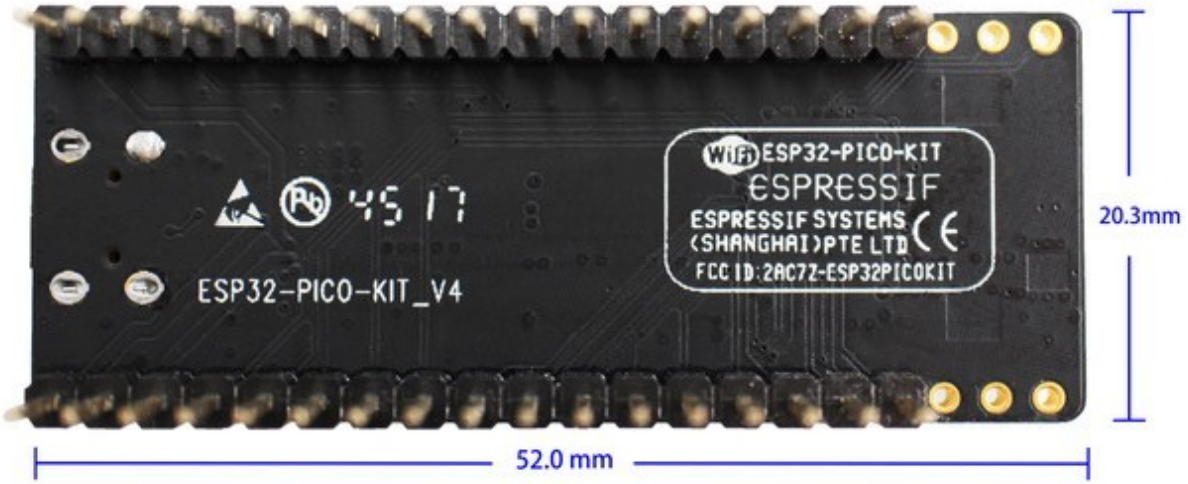


Fig. 13: ESP32-PICO-KIT V4 dimensions - back

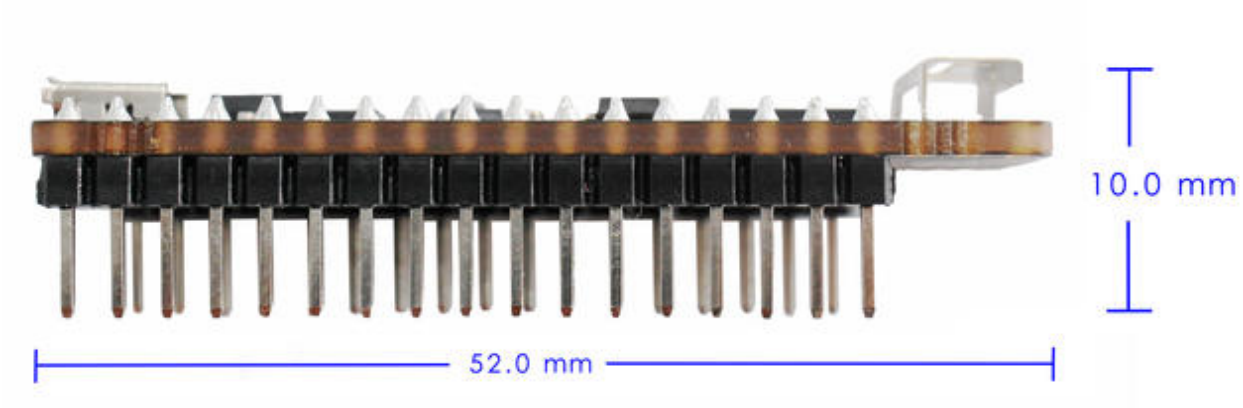


Fig. 14: ESP32-PICO-KIT V4 dimensions - side

Related Documents

- ESP32-PICO-KIT V4 schematic (PDF)
- ESP32-PICO-D4 Datasheet (PDF)
- *ESP32 Hardware Reference*

ESP32-PICO-KIT V3 Getting Started Guide (CMake)

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.

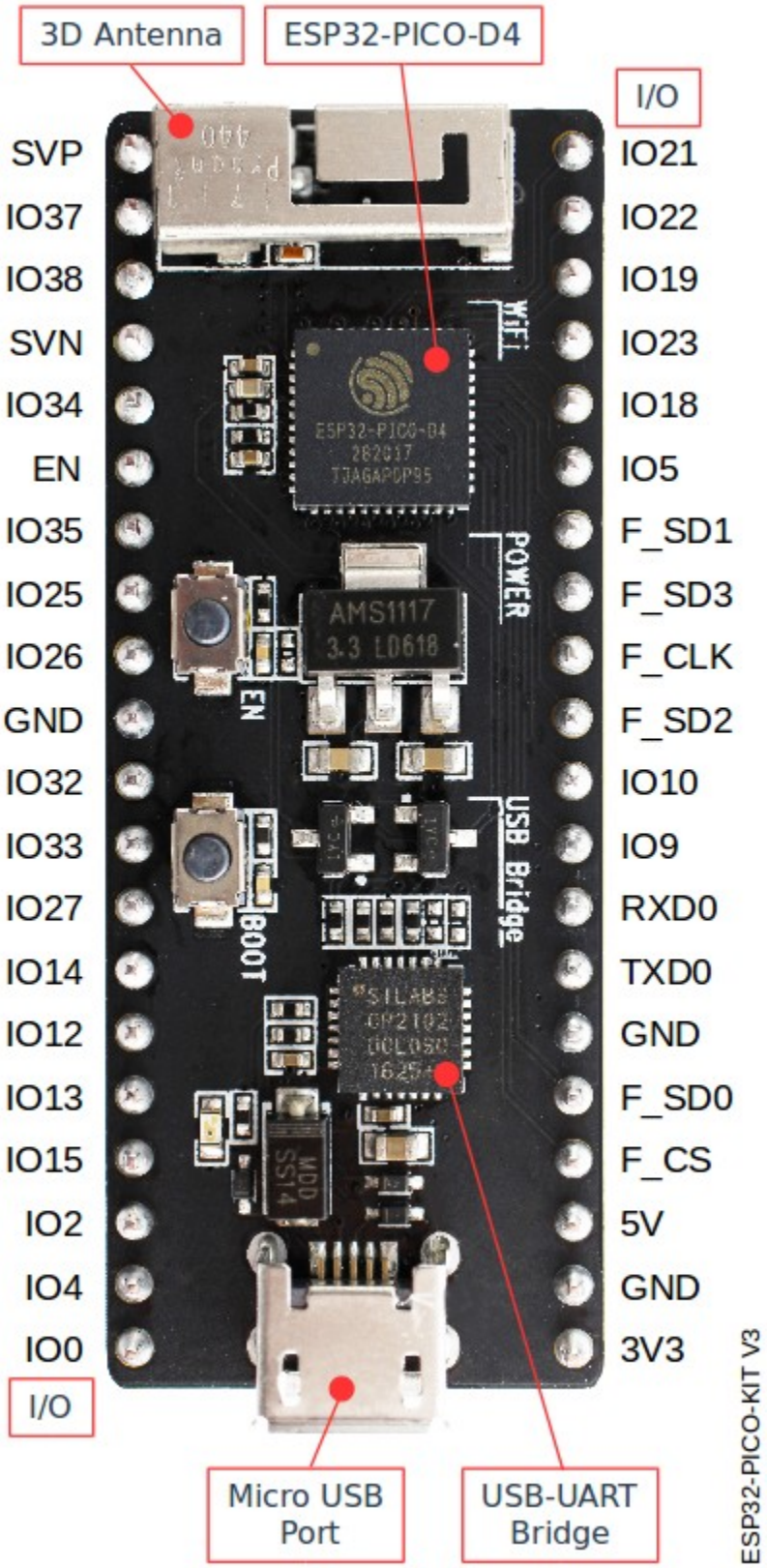


Fig. 15: ESP32-PICO-KIT V3 board layout

To start development of applications, proceed to section *Get Started (CMake)*, 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 The Project* for an example application
- *Flash To A Device* to run code on 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.

2.4 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.

2.4.1 Standard Setup of Toolchain for Windows (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

Note: The CMake-based build system is only supported on 64-bit versions of Windows.

Introduction

ESP-IDF requires some prerequisite tools to be installed so you can build firmware for the ESP32. The prerequisite tools include Git, a cross-compiler and the CMake build tool. We'll go over each one in this document.

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

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

ESP-IDF Tools Installer

The easiest way to install ESP-IDF's prerequisites is to download the ESP-IDF Tools installer from this URL:

https://dl.espressif.com/dl/esp-idf-tools-setup-1.1.exe

The installer will automatically install the ESP32 Xtensa gcc toolchain, Ninja build tool, and a configuration tool called mconf-idf. The installer can also download and run installers for CMake and Python 2.7 if these are not already installed on the computer.

By default, the installer updates the Windows `Path` environment variable so all of these tools can be run from anywhere. If you disable this option, you will need to configure the environment where you are using ESP-IDF (terminal or chosen IDE) with the correct paths.

Note that this installer is for the ESP-IDF Tools package, it doesn't include ESP-IDF itself.

Installing Git

The ESP-IDF tools installer does not install Git. By default, the getting started guide assumes you will be using Git on the command line. You can download and install a command line Git for Windows (along with the "Git Bash" terminal) from Git For Windows.

If you prefer to use a different graphical Git client, then you can install one such as *Github Desktop*. You will need to translate the Git commands in the Getting Started guide for use with your chosen Git client.

Using a Terminal

For the remaining Getting Started steps, we're going to use a terminal command prompt. It doesn't matter which command prompt you use:

- You can use the built-in Windows Command Prompt, under the Start menu. All Windows command line instructions in this documentation are "batch" commands for use with the Windows Command Prompt.
- You can use the "Git Bash" terminal which is part of Git for Windows. This uses the same "bash" command prompt syntax as is given for Mac OS or Linux. You can find it in the Start menu once installed.
- If you have MSYS2 installed (maybe from a previous ESP-IDF version), then you can also use the MSYS terminal.

Next Steps

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

Related Documents

For advanced users who want to customize the install process:

Setup Windows Toolchain from Scratch (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

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

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

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

Tools

cmake

Download the latest stable release of *CMake* for Windows and run the installer.

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

Ninja build

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

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

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

Python 2.x

Download the latest Python 2.7 for Windows installer, and run it.

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

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

```
pip install pyserial
```

MConf for IDF

Download the configuration tool mconf-idf from the kconfig-frontends releases page. This is the `mconf` configuration tool with some minor customizations for ESP-IDF.

This tool will also need to be unzipped to a directory which is then added to your Path.

Toolchain Setup

Download the precompiled Windows toolchain from dl.espressif.com:

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

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

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

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

Adding Directory to Path

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

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

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

Next Steps

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

2.4.2 Standard Setup of Toolchain for Linux (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

Install Prerequisites

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

- CentOS 7:

```
sudo yum install git wget ncurses-devel flex bison gperf python pyserial cmake␣
↪ninja-build ccache
```

- Ubuntu and Debian:

```
sudo apt-get install git wget libncurses-dev flex bison gperf python python-pip␣
↪python-setuptools python-serial python-cryptography python-future python-
↪pyparsing cmake ninja-build ccache libffi-dev libssl-dev
```

(continues on next page)

(continued from previous page)

- Arch:

```
sudo pacman -S --needed gcc git make ncurses flex bison gperf python2-pyserial␣
↪python2-cryptography python2-future python2-pyparsing cmake ninja ccache
```

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

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
```

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 (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

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 (CMake)*.

Install Prerequisites

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

- CentOS 7:

```
sudo yum install git wget ncurses-devel flex bison gperf python pyserial cmake␣
↪ninja-build ccache
```

- Ubuntu and Debian:

```
sudo apt-get install git wget libncurses-dev flex bison gperf python python-pip␣
↪python-setuptools python-serial python-cryptography python-future python-
↪pyparsing cmake ninja-build ccache libffi-dev libssl-dev
```

- Arch:

```
sudo pacman -S --needed gcc git make ncurses flex bison gperf python2-pyserial␣
↪python2-cryptography python2-future python2-pyparsing cmake ninja ccache
```

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

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 make
    ```

  - Ubuntu pre-16.04:

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

  - Ubuntu 16.04 or newer:

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

  - Debian 9:

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

  - Arch:

    ```
    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*.

2.4.3 Standard Setup of Toolchain for Mac OS (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

Install Prerequisites

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

- install pip:

  ```
  sudo easy_install pip
  ```

- install pyserial:

  ```
  sudo pip install pyserial
  ```

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

    ```
    brew install cmake ninja
    ```

  - If you have MacPorts, you can run:

    ```
    sudo port install cmake ninja
    ```

  - Otherwise, consult the CMake and Ninja home pages for Mac OS installation downloads.
- It is strongly recommended to also install ccache for faster builds. If you have HomeBrew, this can be done via `brew install ccache` or `sudo port install ccache` on MacPorts.

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

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

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

(continues on next page)

(continued from previous page)

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`.

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
```

Next Steps

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

Related Documents

Setup Toolchain for Mac OS from Scratch (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

Package Manager

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

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

Install Prerequisites

- install pip:

```
sudo easy_install pip
```

- install pyserial:

```
sudo pip install pyserial
```

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

```
brew install cmake ninja
```

  - If you have MacPorts, you can run:

```
sudo port install cmake ninja
```

Compile the Toolchain from Source

- Install dependencies:
  - with MacPorts:

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

  - with homebrew:

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

Create a case-sensitive filesystem image:

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

Mount it:

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

Create a symlink to your work directory:

```
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*.

Windows	Linux	Mac OS

Note: We are an using `esp` subdirectory in your user's home directory (`~/esp` on Linux and MacOS, `%userprofile%\esp` on Windows) to install everything needed for ESP-IDF. You can use any different directory, but will need to adjust the 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 (CMake)*.

Once you are done with setting up the toolchain then go to section *Get ESP-IDF*.

2.5 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 get it, open terminal, navigate to the directory you want to put ESP-IDF, and clone it using `git clone` command.

2.5.1 Linux and MacOS

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

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

2.5.2 Windows Command Prompt

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

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

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

2.6 Setup Environment Variables

ESP-IDF requires two environment variables to be set for normal operation:

- `IDF_PATH` should be set to the path to the ESP-IDF root directory.
- `PATH` should include the path to the `tools` directory inside the same `IDF_PATH` directory.

These two variables 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 them permanently in user profile. To do this, follow instructions specific to *Windows* , *Linux and MacOS* in section *Add IDF_PATH & idf.py PATH to User Profile (CMake)*.

2.7 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:

2.7.1 Linux and MacOS

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

2.7.2 Windows Command Prompt

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

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

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

2.8 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 (CMake)*. Note the port number, as it will be required in the next step.

2.9 Configure

Naviagate to the directory of the `hello_world` application copy, and run the `menuconfig` project configuration utility:

2.9.1 Linux and MacOS

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

2.9.2 Windows Command Prompt

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

Note: If you get an error about `idf.py` not being found, check the `tools` directory is part of your Path as described above in *Setup Environment Variables*. If there is no `idf.py` in the `tools` directory, check you have the correct branch for the CMake preview as shown under *Get ESP-IDF*.

Note: Windows users, the Python 2.7 installer will try to configure Windows to associate files with a `.py` extension with Python 2. If a separate installed program (such as Visual Studio Python Tools) has created an association with a different version of Python, then running `idf.py` may not work (it opens the file in Visual Studio instead). You can either run `C:\Python27\python idf.py` each time instead, or change the association that Windows uses for `.py` files.

Note: Linux users, if your default version of Python is 3.x then you may need to run `python2 idf.py` instead.

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

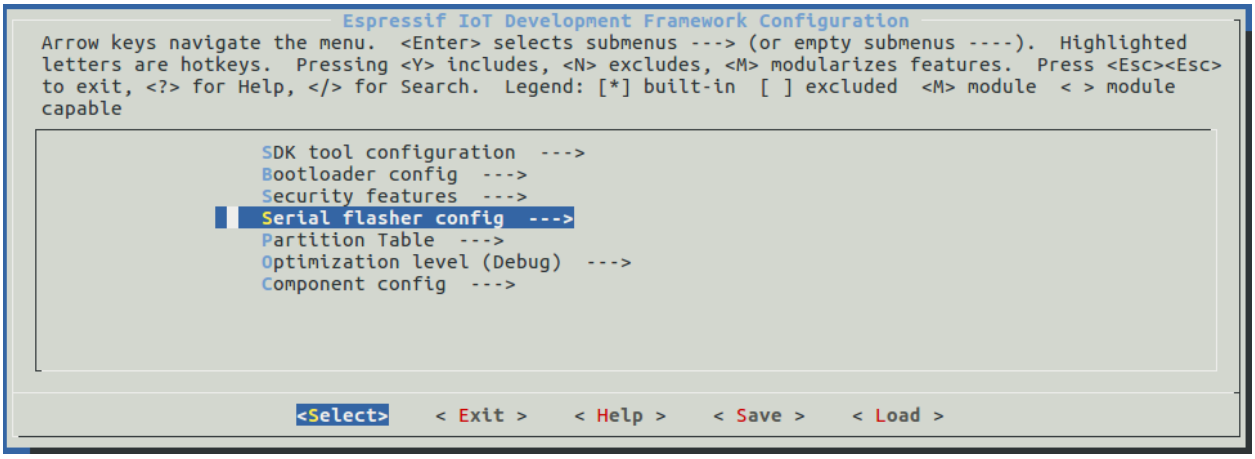


Fig. 16: Project configuration - Home window

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 up a level or 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.

Attention: When using ESP32-DevKitC board with ESP32-SOLO-1 module, enable single core mode (*CONFIG_FREERTOS_UNICORE*) in menuconfig before flashing example applications.

2.10 Build The Project

Now you can build the project. Run:

```
idf.py build
```

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

```
$ idf.py build
Running cmake in directory /path/to/hello_world/build
Executing "cmake -G Ninja --warn-uninitialized /path/to/hello_world"...
Warn about uninitialized values.
-- Found Git: /usr/bin/git (found version "2.17.0")
```

(continues on next page)

(continued from previous page)

```
-- Building empty aws_iot component due to configuration
-- Component names: ...
-- Component paths: ...

... (more lines of build system output)

[527/527] Generating hello-world.bin
esptool.py v2.3.1

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

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

2.11 Flash To A Device

Now you can flash the application to the ESP32 board. Run:

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

Replace PORT with the name of your ESP32 board's serial port. 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 (CMake)* for full details.

This step will flash the binaries that you just built to your ESP32 board.

Note: Running `idf.py build` before `idf.py flash` is not actually necessary, the flash step will automatically build the project if required before flashing.

```
Running esptool.py in directory [...]/esp/hello_world
Executing "python [...]/esp-idf/components/esptool_py/esptool/esptool.py -b 460800␣
↪write_flash @flash_project_args"...
esptool.py -b 460800 write_flash --flash_mode dio --flash_size detect --flash_freq␣
↪40m 0x1000 bootloader/bootloader.bin 0x8000 partition_table/partition-table.bin␣
↪0x10000 hello-world.bin
esptool.py v2.3.1
Connecting....
Detecting chip type... ESP32
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Auto-detected Flash size: 4MB
Flash params set to 0x0220
Compressed 22992 bytes to 13019...
```

(continues on next page)

(continued from previous page)

```
Wrote 22992 bytes (13019 compressed) at 0x00001000 in 0.3 seconds (effective 558.9␣
↪kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 82...
Wrote 3072 bytes (82 compressed) at 0x00008000 in 0.0 seconds (effective 5789.3 kbit/
↪s)...
Hash of data verified.
Compressed 136672 bytes to 67544...
Wrote 136672 bytes (67544 compressed) at 0x00010000 in 1.9 seconds (effective 567.5␣
↪kbit/s)...
Hash of data verified.

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

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

2.12 Monitor

To see if "hello_world" application is indeed running, type `idf.py -p PORT monitor`. This command is launching *IDF Monitor* application:

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

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

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. Most development board designs use 40MHz and the ESP-IDF uses this default value. Exit the monitor, go back to the *menuconfig*, change

CONFIG_ESP32_XTAL_FREQ_SEL to 26MHz, then *build and flash* the application again. This is found under `idf.py menuconfig` under Component config –> ESP32-specific –> Main XTAL frequency.

Note: You can combine building, flashing and monitoring into one step as follows:

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

Check the section *IDF Monitor* for handy shortcuts and more details on using the monitor.

Check the section *idf.py* for a full reference of `idf.py` commands and options.

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.

2.13 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*.

Another solution is to update only what has changed. This method is useful if you have a slow connection to GitHub. To do the update run the following commands:

2.13.1 Linux and MacOS

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

2.13.2 Windows Command Prompt

```
cd %userprofile%\esp\esp-idf
git pull
git submodule update --init --recursive
```

The `git pull` command is fetching and merging changes from ESP-IDF repository on GitHub. Then `git submodule update --init --recursive` is updating existing submodules or getting a fresh copy of new ones. On GitHub the submodules are represented as links to other repositories and require this additional command to get them onto your PC.

It is also possible to check out a specific release of ESP-IDF, e.g. *v2.1*.

2.13.3 Linux and MacOS

```
cd ~/esp
git clone https://github.com/espressif/esp-idf.git esp-idf-v2.1
cd esp-idf-v2.1/
```

(continues on next page)

(continued from previous page)

```
git checkout v2.1
git submodule update --init --recursive
```

2.13.4 Windows Command Prompt

```
cd %userprofile%\esp
git clone https://github.com/espressif/esp-idf.git esp-idf-v2.1
cd esp-idf-v2.1/
git checkout v2.1
git submodule update --init --recursive
```

After that remember to *Add IDF_PATH & idf.py PATH to User Profile (CMake)*, so the toolchain scripts know where to find the ESP-IDF in it's release specific location.

Note: Different versions of ESP-IDF may have different setup or prerequisite requirements, or require different toolchain versions. If you experience any problems, carefully check the Getting Started documentation for the version you are switching to.

2.14 Related Documents

2.14.1 Add IDF_PATH & idf.py PATH to User Profile (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

To use the CMake-based build system and the idf.py tool, two modifications need to be made to system environment variables:

- `IDF_PATH` needs to be set to the path of the directory containing ESP-IDF.
- System `PATH` variable to include the directory containing the `idf.py` tool (part of ESP-IDF).

To preserve setting of these variables between system restarts, add them to the user profile by following the instructions below.

Note: If using an IDE, you can optionally set these environment variables in your IDE's project environment rather than from the command line as described below.

Note: If you don't ever use the command line `idf.py` tool, but run cmake directly or via an IDE, then it is not necessary to set the `PATH` variable - only `IDF_PATH`. However it can be useful to set both.

Note: If you only ever use the command line `idf.py` tool, and never use cmake directly or via an IDE, then it is not necessary to set the `IDF_PATH` variable - `idf.py` will detect the directory it is contained within and set `IDF_PATH` appropriately if it is missing.

Windows

To edit Environment Variables on Windows 10, search for "Edit Environment Variables" under the Start menu.

On earlier Windows versions, open the System Control Panel then choose "Advanced" and look for the Environment Variables button.

You can set these environment variables for all users, or only for the current user, depending on whether other users of your computer will be using ESP-IDF.

- Click `New...` to add a new system variable named `IDF_PATH`. Set the path to directory containing ESP-IDF, for example `C:\Users\user-name\esp\esp-idf`.
- Locate the `Path` environment variable and double-click to edit it. Append the following to the end: `;%IDF_PATH%\tools`. This will allow you to run `idf.py` and other tools from Windows Command Prompt.

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

Linux and MacOS

Set up `IDF_PATH` and add `idf.py` to the PATH by adding the following two lines to your `~/.profile` file:

```
export IDF_PATH=~/esp/esp-idf
export PATH="$PATH:$IDF_PATH/tools"
```

Note: `~/.profile` means a file named `.profile` in your user's home directory (which is abbreviated ~ in the shell).

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

Note: Not all shells use `.profile`. If you have `/bin/bash` and `.bash_profile` exists then update this file instead. For `zsh`, update `.zprofile`. Other shells may use other profile files (consult the shell's documentation).

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.

To verify idf.py is now on the `PATH`, you can run the following:

```
which idf.py
```

A path like `${IDF_PATH}/tools/idf.py` should be printed.

If you do not like to have `IDF_PATH` or `PATH` modifications set, you can enter it manually in terminal window on each restart or logout:

```
export IDF_PATH=~/esp/esp-idf
export PATH="$PATH:$IDF_PATH/tools"
```

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

2.14.2 Establish Serial Connection with ESP32 (CMake)

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

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.*
```

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.

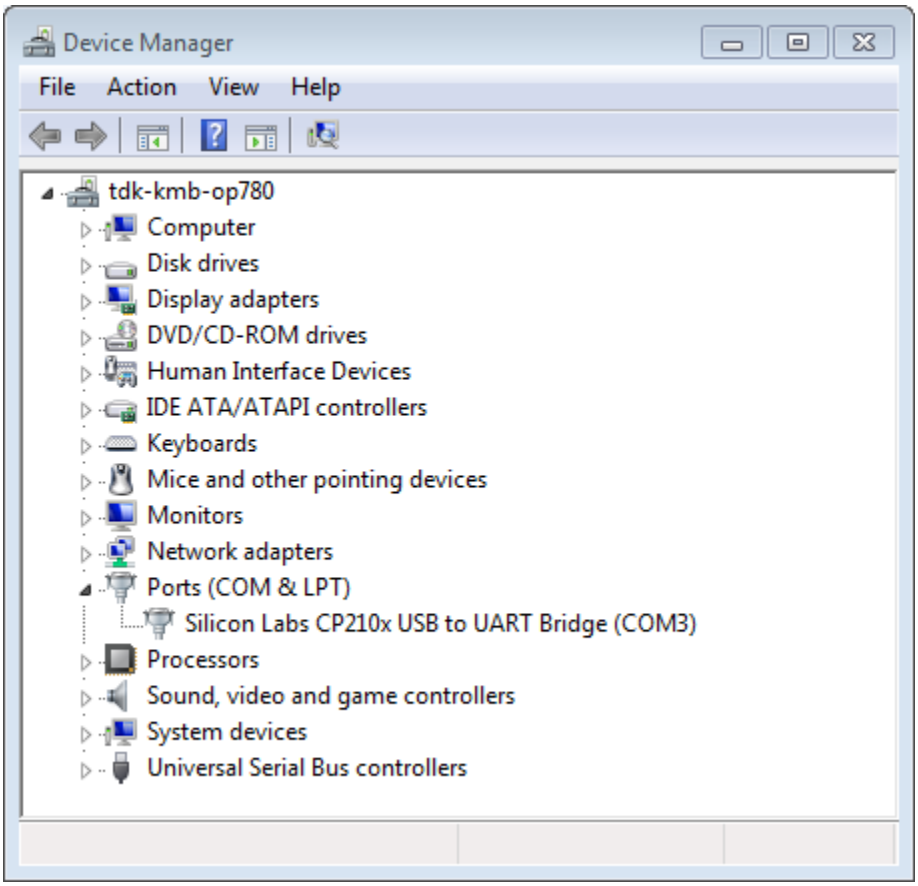


Fig. 17: USB to UART bridge of ESP32-DevKitC in Windows Device Manager

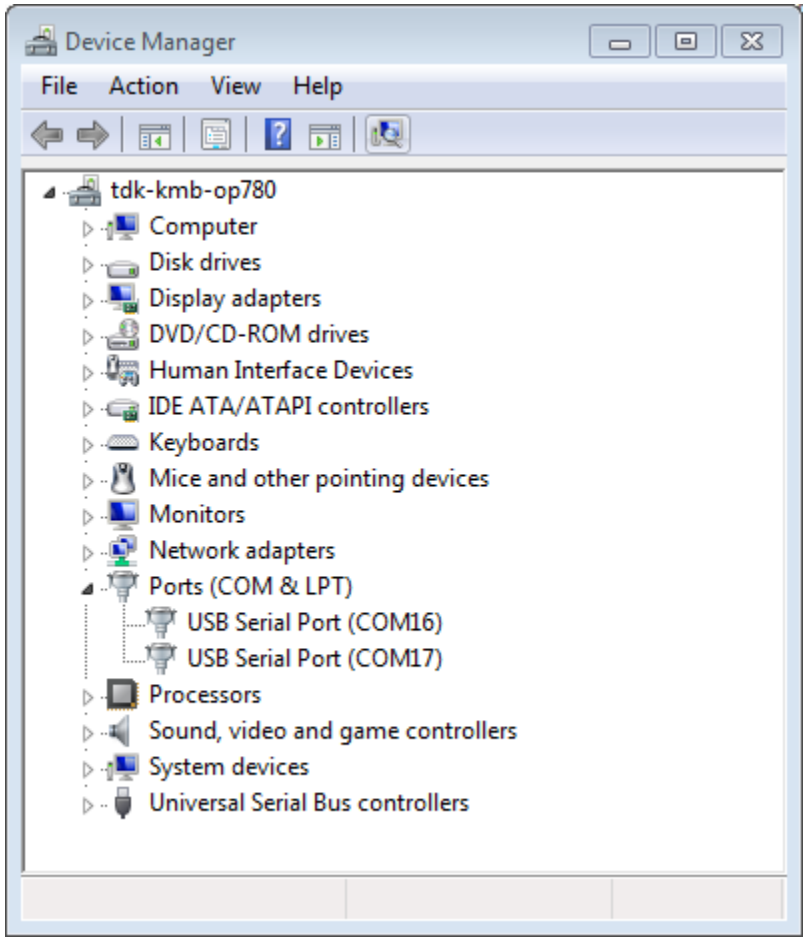


Fig. 18: Two USB Serial Ports of ESP-WROVER-KIT in Windows Device Manager

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.

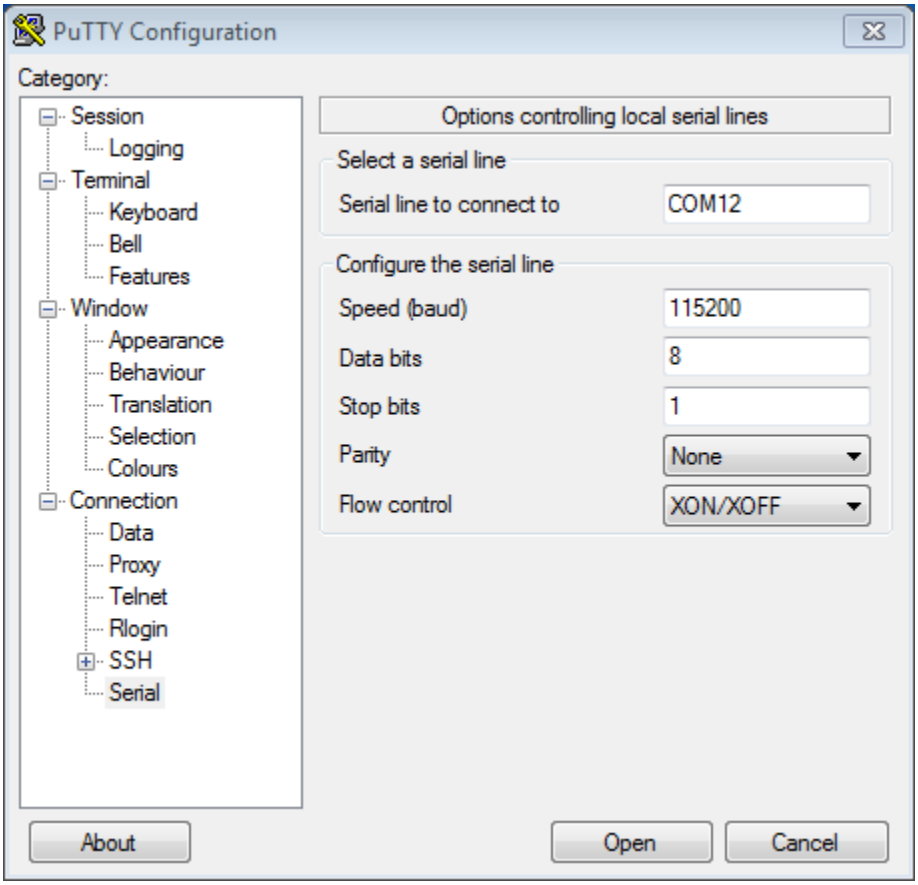


Fig. 19: Setting Serial Communication in PuTTY on Windows

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.

```
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
```

(continues on next page)

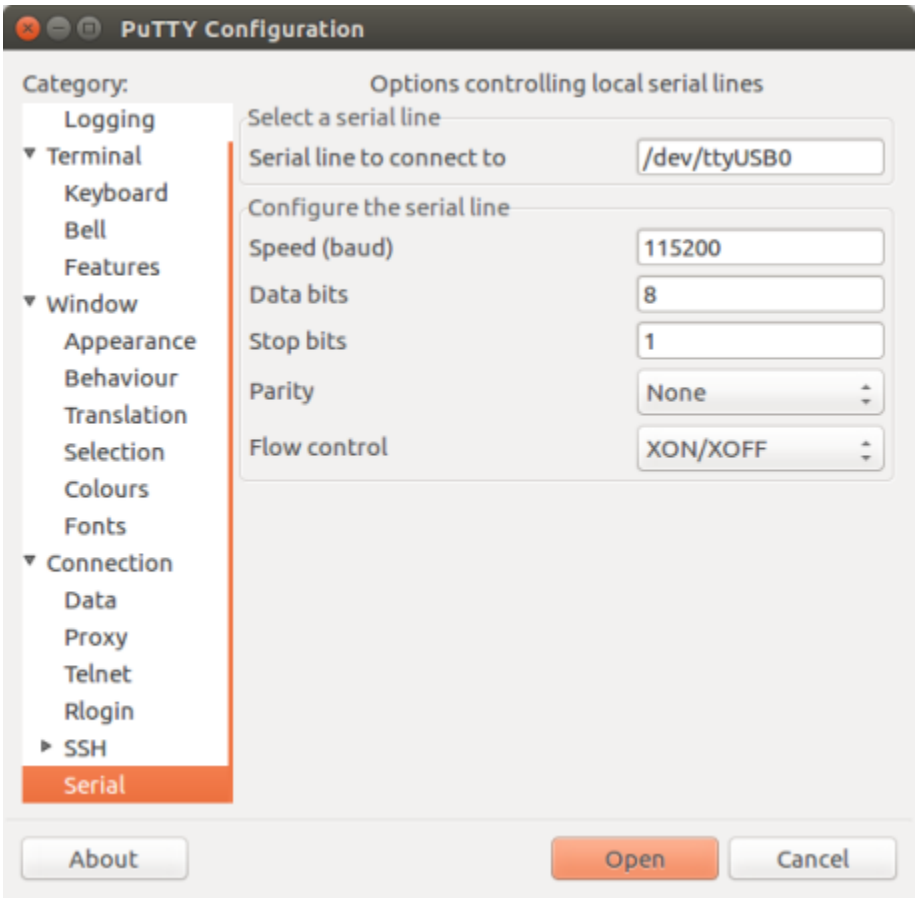


Fig. 20: Setting Serial Communication in PuTTY on Linux

(continued from previous page)

```
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*.

2.14.3 Build and Flash with Eclipse IDE (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

Documentation for Eclipse setup with CMake-based build system and Eclipse CDT is coming soon.

2.14.4 IDF Monitor (CMake)

The idf_monitor tool is a Python program which runs when the `idf.py 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 features.

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 `idf.py menuconfig` and set *CONFIG_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 `idf.py 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 `idf.py app-flash` target, then resume idf_monitor. This is similar to `idf.py 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.

Note: This target only works in the GNU Make based build system, not the CMake-based build system preview.

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.

2.14.5 Customized Setup of Toolchain (CMake)

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.

CHAPTER 3

API Reference

3.1 Wi-Fi API

3.1.1 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 owner initial values, overwrite the default values which are set by WIFI_INIT_CONFIG_DEFAULT, please be notified that the field 'magic' of *wifi_init_config_t* should always be WIFI_INIT_CONFIG_MAGIC!

Return

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

Parameters

- `config`: pointer to WiFi 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_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_set_mode** (*wifi_mode_t* *mode*)

Set the WiFi operating mode.

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

Return

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

Parameters

- `mode`: WiFi operating mode

esp_err_t **esp_wifi_get_mode**(*wifi_mode_t *mode*)

Get current operating mode of WiFi.

Return

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

Parameters

- `mode`: store current WiFi mode

esp_err_t **esp_wifi_start**(void)

Start WiFi according to current configuration If mode is WIFI_MODE_STA, it create station control block and start station If mode is WIFI_MODE_AP, it create soft-AP control block and start soft-AP If mode is WIFI_MODE_APSTA, it create soft-AP and station control block and start soft-AP and station.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_NO_MEM: out of memory
- ESP_ERR_WIFI_CONN: WiFi internal error, station or soft-AP control block wrong
- ESP_FAIL: other WiFi internal errors

esp_err_t **esp_wifi_stop**(void)

Stop WiFi If mode is WIFI_MODE_STA, it stop station and free station control block If mode is WIFI_MODE_AP, it stop soft-AP and free soft-AP control block If mode is WIFI_MODE_APSTA, it stop station/soft-AP and free station/soft-AP control block.

Return

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

esp_err_t **esp_wifi_restore**(void)

Restore WiFi stack persistent settings to default values.

This function will reset settings made using the following APIs:

- esp_wifi_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_FAIL: other WiFi internal errors

esp_err_t **esp_wifi_clear_fast_connect** (void)
Currently this API is just an stub API.

Return

- ESP_OK: succeed
- others: fail

esp_err_t **esp_wifi_deauth_sta** (uint16_t *aid*)
deauthenticate all stations or associated id equals to aid

Return

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

Parameters

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

esp_err_t **esp_wifi_scan_start**(**const** *wifi_scan_config_t* **config*, bool *block*)

Scan all available APs.

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

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

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi was not started by esp_wifi_start
- ESP_ERR_WIFI_TIMEOUT: blocking scan is timeout
- ESP_ERR_WIFI_STATE: wifi still connecting when invoke esp_wifi_scan_start
- others: refer to error code in esp_err.h

Parameters

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

esp_err_t **esp_wifi_scan_stop**(void)

Stop the scan in process.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start

esp_err_t **esp_wifi_scan_get_ap_num**(uint16_t **number*)

Get number of APs found in last scan.

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

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

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

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

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_NO_MEM: out of memory

Parameters

- `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 sta can get the connected ap's phy mode info through the struct member phy_11bphy_11gphy_11nphy_lr in the *wifi_ap_record_t* struct. For example, phy_11b = 1 imply that ap support 802.11b mode

esp_err_t **esp_wifi_set_ps** (*wifi_ps_type_t type*)
Set current WiFi power save type.

Attention Default power save type is WIFI_PS_MIN_MODEM.

Return ESP_OK: succeed

Parameters

- `type`: power save type

esp_err_t **esp_wifi_get_ps** (*wifi_ps_type_t *type*)
Get current WiFi power save type.

Attention Default power save type is WIFI_PS_MIN_MODEM.

Return ESP_OK: succeed

Parameters

- `type`: store current power save type

esp_err_t **esp_wifi_set_protocol** (*wifi_interface_t ifx*, uint8_t *protocol_bitmap*)
Set protocol type of specified interface The default protocol is (WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROTOC

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_INVALID_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_INVALID_ARG: invalid argument
- others: refer to error codes in esp_err.h

Parameters

- `ifx`: interface to be configured
- `bw`: bandwidth

esp_err_t **esp_wifi_get_bandwidth** (*wifi_interface_t ifx*, *wifi_bandwidth_t **bw*)

Get the bandwidth of ESP32 specified interface.

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

Return

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

Parameters

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

esp_err_t **esp_wifi_set_channel** (uint8_t *primary*, *wifi_second_chan_t second*)

Set primary/secondary channel of ESP32.

Attention 1. This API should be called after esp_wifi_start()

Attention 2. When ESP32 is in STA mode, this API should not be called when STA is scanning or connecting to an external AP

Attention 3. When ESP32 is in softAP mode, this API should not be called when softAP has connected to external STAs

Attention 4. When ESP32 is in STA+softAP mode, this API should not be called when in the scenarios described above

Return

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

Parameters

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

esp_err_t **esp_wifi_get_channel** (uint8_t **primary*, *wifi_second_chan_t **second*)

Get the primary/secondary channel of ESP32.

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

Return

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

Parameters

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

esp_err_t **esp_wifi_set_country**(**const** *wifi_country_t* **country*)
configure country info

Attention 1. The default country is {.cc="CN", .schan=1, .nchan=13, policy=WIFI_COUNTRY_POLICY_AUTO}

Attention 2. When the country policy is WIFI_COUNTRY_POLICY_AUTO, the country info of the AP to which the station is connected is used. E.g. if the configured country info is {.cc="USA", .schan=1, .nchan=11} and the country info of the AP to which the station is connected is {.cc="JP", .schan=1, .nchan=14} then the country info that will be used is {.cc="JP", .schan=1, .nchan=14}. If the station disconnected from the AP the country info is set back back to the country info of the station automatically, {.cc="US", .schan=1, .nchan=11} in the example.

Attention 3. When the country policy is WIFI_COUNTRY_POLICY_MANUAL, always use the configured country info.

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

Attention 5. The country configuration is not stored into flash

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

Return

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

Parameters

- `country`: the configured country info

esp_err_t **esp_wifi_get_country**(*wifi_country_t* **country*)
get the current country info

Return

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

Parameters

- `country`: country info

esp_err_t **esp_wifi_set_mac**(*wifi_interface_t* *ifx*, **const** uint8_t *mac*[6])
Set MAC address of the ESP32 WiFi station or the soft-AP interface.

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

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

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

Return

- ESP_OK: succeed

- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_WIFI_MAC: invalid mac address
- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- others: refer to error codes in esp_err.h

Parameters

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

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

Get mac of specified interface.

Return

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

Parameters

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

esp_err_t **esp_wifi_set_promiscuous_rx_cb** (*wifi_promiscuous_cb_t cb*)

Register the RX callback function in the promiscuous mode.

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

Return

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

Parameters

- `cb`: callback

esp_err_t **esp_wifi_set_promiscuous** (bool *en*)

Enable the promiscuous mode.

Return

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

Parameters

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

esp_err_t **esp_wifi_get_promiscuous**(bool **en*)
Get the promiscuous mode.

Return

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

Parameters

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

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

Note The default filter is to filter all packets except WIFI_PKT_MISC

Return

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

Parameters

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

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

Return

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

Parameters

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

esp_err_t **esp_wifi_set_promiscuous_ctrl_filter**(**const** *wifi_promiscuous_filter_t *filter*)
Enable subtype filter of the control packet in promiscuous mode.

Note The default filter is to filter none control packet.

Return

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

Parameters

- `filter`: the subtype of the control packet filtered in promiscuous mode.

esp_err_t **esp_wifi_get_promiscuous_ctrl_filter**(*wifi_promiscuous_filter_t *filter*)
Get the subtype filter of the control packet in promiscuous mode.

Return

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

Parameters

- `filter`: store the current status of subtype filter of the control packet in promiscuous mode

esp_err_t **esp_wifi_set_config** (*wifi_interface_t interface*, *wifi_config_t *conf*)

Set the configuration of the ESP32 STA or AP.

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

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

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

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_WIFI_MODE: invalid mode
- ESP_ERR_WIFI_PASSWORD: invalid password
- ESP_ERR_WIFI_NVS: WiFi internal NVS error
- others: refer to the erro code in esp_err.h

Parameters

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

esp_err_t **esp_wifi_get_config** (*wifi_interface_t interface*, *wifi_config_t *conf*)

Get configuration of specified interface.

Return

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

Parameters

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

esp_err_t **esp_wifi_ap_get_sta_list** (*wifi_sta_list_t *sta*)

Get STAs associated with soft-AP.

Attention SSC only API

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- ESP_ERR_WIFI_CONN: WiFi internal error, the station/soft-AP control block is invalid

Parameters

- `sta`: station list ap can get the connected sta's phy mode info through the struct member phy_11bphy_11gphy_11nphy_lr in the *wifi_sta_info_t* struct. For example, phy_11b = 1 imply that sta support 802.11b mode

esp_err_t **esp_wifi_set_storage** (*wifi_storage_t storage*)

Set the WiFi API configuration storage type.

Attention 1. The default value is WIFI_STORAGE_FLASH

Return

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

Parameters

- `storage`: : storage type

esp_err_t **esp_wifi_set_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_INVALID_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_INVALID_ARG: Invalid argument, including if first byte of vnd_ie is not WIFI_VENDOR_IE_ELEMENT_ID (0xDD) or second byte is an invalid length.
- ESP_ERR_NO_MEM: Out of memory

Parameters

- `enable`: If true, specified IE is enabled. If false, specified IE is removed.
- `type`: Information Element type. Determines the frame type to associate with the IE.
- `idx`: Index to set or clear. Each IE type can be associated with up to two elements (indices 0 & 1).
- `vnd_ie`: Pointer to vendor specific element data. First 6 bytes should be a header with fields matching *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.

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_NOT_ARG: invalid argument

Parameters

- `power`: Maximum WiFi transmitting power, unit is 0.25dBm, range is [40, 82] corresponding to 10dBm - 20.5dBm here.

esp_err_t **esp_wifi_get_max_tx_power** (int8_t **power*)

Get maximum WiFi transmiting power.

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_INVALID_ARG: invalid argument

Parameters

- `power`: Maximum WiFi transmitting power, unit is 0.25dBm.

esp_err_t **esp_wifi_set_event_mask** (uint32_t *mask*)

Set mask to enable or disable some WiFi events.

Attention 1. Mask can be created by logical OR of various WIFI_EVENT_MASK_ constants. Events which have corresponding bit set in the mask will not be delivered to the system event handler.

Attention 2. Default WiFi event mask is WIFI_EVENT_MASK_AP_PROBEREQRECVED.

Attention 3. There may be lots of stations sending probe request data around. Don't unmask this event unless you need to receive probe request data.

Return

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

Parameters

- `mask`: WiFi event mask.

esp_err_t **esp_wifi_get_event_mask** (uint32_t **mask*)

Get mask of WiFi events.

Return

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

Parameters

- `mask`: WiFi event mask.

esp_err_t **esp_wifi_80211_tx** (*wifi_interface_t ifx*, **const** void **buffer*, int *len*, bool *en_sys_seq*)

Send raw ieee80211 data.

Attention Currently only support for sending beacon/probe request/probe response/action and non-QoS data frame

Return

- ESP_OK: success
- ESP_ERR_WIFI_IF: Invalid interface
- ESP_ERR_INVALID_ARG: Invalid parameter
- ESP_ERR_WIFI_NO_MEM: out of memory

Parameters

- `ifx`: interface if the Wi-Fi mode is Station, the ifx should be WIFI_IF_STA. If the Wi-Fi mode is SoftAP, the ifx should be WIFI_IF_AP. If the Wi-Fi mode is Station+SoftAP, the ifx should be WIFI_IF_STA or WIFI_IF_AP. If the ifx is wrong, the API returns ESP_ERR_WIFI_IF.
- `buffer`: raw ieee80211 buffer
- `len`: the length of raw buffer, the len must be <= 1500 Bytes and >= 24 Bytes
- `en_sys_seq`: indicate whether use the internal sequence number. If en_sys_seq is false, the sequence in raw buffer is unchanged, otherwise it will be overwritten by WiFi driver with the system sequence number. Generally, if esp_wifi_80211_tx is called before the Wi-Fi connection has been set up, both en_sys_seq==true and en_sys_seq==false are fine. However, if the API is called after the Wi-Fi connection has been set up, en_sys_seq must be true, otherwise ESP_ERR_WIFI_ARG is returned.

esp_err_t **esp_wifi_set_csi_rx_cb**(*wifi_csi_cb_t cb*, void **ctx*)

Register the RX callback function of CSI data.

Each time a CSI data is received, the callback function will be called.

Return

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

Parameters

- `cb`: callback
- `ctx`: context argument, passed to callback function

esp_err_t **esp_wifi_set_csi_config**(**const** *wifi_csi_config_t **config*)

Set CSI data configuration.

return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_START: WiFi is not started by esp_wifi_start or promiscuous mode is not enabled
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- `config`: configuration

esp_err_t **esp_wifi_set_csi**(bool *en*)

Enable or disable CSI.

return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_START: WiFi is not started by esp_wifi_start or promiscuous mode is not enabled
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

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

esp_err_t **esp_wifi_set_ant_gpio**(**const** *wifi_ant_gpio_config_t* **config*)
Set antenna GPIO configuration.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: Invalid argument, e.g. parameter is NULL, invalid GPIO number etc

Parameters

- `config`: Antenna GPIO configuration.

esp_err_t **esp_wifi_get_ant_gpio**(*wifi_ant_gpio_config_t* **config*)
Get current antenna GPIO configuration.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: invalid argument, e.g. parameter is NULL

Parameters

- `config`: Antenna GPIO configuration.

esp_err_t **esp_wifi_set_ant**(**const** *wifi_ant_config_t* **config*)
Set antenna configuration.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: Invalid argument, e.g. parameter is NULL, invalid antenna mode or invalid GPIO number

Parameters

- `config`: Antenna configuration.

esp_err_t **esp_wifi_get_ant**(*wifi_ant_config_t* **config*)
Get current antenna configuration.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: invalid argument, e.g. parameter is NULL

Parameters

- `config`: Antenna configuration.

Structures

struct wifi_init_config_t

WiFi stack configuration parameters passed to esp_wifi_init call.

Public Members

system_event_handler_t **event_handler**
WiFi event handler

wifi_osi_funcs_t ***osi_funcs**
WiFi OS functions

wpa_crypto_funcs_t **wpa_crypto_funcs**
WiFi station crypto functions when connect

int **static_rx_buf_num**
WiFi static RX buffer number

int **dynamic_rx_buf_num**
WiFi dynamic RX buffer number

int **tx_buf_type**
WiFi TX buffer type

int **static_tx_buf_num**
WiFi static TX buffer number

int **dynamic_tx_buf_num**
WiFi dynamic TX buffer number

int **csi_enable**
WiFi channel state information enable flag

int **ampdu_rx_enable**
WiFi AMPDU RX feature enable flag

int **ampdu_tx_enable**
WiFi AMPDU TX feature enable flag

int **nvs_enable**
WiFi NVS flash enable flag

int **nano_enable**
Nano option for printf/scan family enable flag

int **tx_ba_win**
WiFi Block Ack TX window size

int **rx_ba_win**
WiFi Block Ack RX window size

int **wifi_task_core_id**
WiFi Task Core ID

int **beacon_max_len**
WiFi softAP maximum length of the beacon

int **mgmt_sbuf_num**
WiFi management short buffer number, the minimum value is 6, the maximum value is 32

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

Macros

ESP_ERR_WIFI_NOT_INIT
WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED
WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED
WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF
WiFi interface error

ESP_ERR_WIFI_MODE
WiFi mode error

ESP_ERR_WIFI_STATE
WiFi internal state error

ESP_ERR_WIFI_CONN
WiFi internal control block of station or soft-AP error

ESP_ERR_WIFI_NVS
WiFi internal NVS module error

ESP_ERR_WIFI_MAC
MAC address is invalid

ESP_ERR_WIFI_SSID
SSID is invalid

ESP_ERR_WIFI_PASSWORD
Password is invalid

ESP_ERR_WIFI_TIMEOUT
Timeout error

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

ESP_ERR_WIFI_WOULD_BLOCK
The caller would block

ESP_ERR_WIFI_NOT_CONNECT
Station still in disconnect status

ESP_ERR_WIFI_POST
Failed to post the event to WiFi task

ESP_ERR_WIFI_INIT_STATE
Invalod WiFi state when init/deinit is called

ESP_ERR_WIFI_STOP_STATE
Returned when WiFi is stopping

WIFI_STATIC_TX_BUFFER_NUM

WIFI_DYNAMIC_TX_BUFFER_NUM

WIFI_CSI_ENABLED

WIFI_AMPDU_RX_ENABLED

WIFI_AMPDU_TX_ENABLED

WIFI_NVS_ENABLED

WIFI_NANO_FORMAT_ENABLED

WIFI_INIT_CONFIG_MAGIC

WIFI_DEFAULT_TX_BA_WIN

WIFI_DEFAULT_RX_BA_WIN

WIFI_TASK_CORE_ID

WIFI_SOFTAP_BEACON_MAX_LEN

WIFI_MGMT_SBUF_NUM

WIFI_INIT_CONFIG_DEFAULT()

Type Definitions

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

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

Parameters

- `buf`: Data received. Type of data in buffer (*wifi_promiscuous_pkt_t* or *wifi_pkt_rx_ctrl_t*) indicated by 'type' parameter.
- `type`: promiscuous packet type.

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

Function signature for received Vendor-Specific Information Element callback.

Parameters

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

typedef void (***wifi_csi_cb_t**)(void *ctx, *wifi_csi_info_t* *data)

The RX callback function of Channel State Information(CSI) data.

Each time a CSI data is received, the callback function will be called.

Parameters

- `ctx`: context argument, passed to esp_wifi_set_csi_rx_cb() when registering callback function.
- `data`: CSI data received. The memory that it points to will be deallocated after callback function returns.

Header File

- 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

int8_t **max_tx_power**
This field is used for getting WiFi maximum transmitting power, call esp_wifi_set_max_tx_power to set the maximum transmitting power.

wifi_country_policy_t **policy**
country policy

struct wifi_active_scan_time_t

Range of active scan times per channel.

Public Members

uint32_t **min**

minimum active scan time per channel, units: millisecond

uint32_t **max**

maximum active scan time per channel, units: millisecond, values above 1500ms may cause station to disconnect from AP and are not recommended.

struct wifi_scan_config_t

Parameters for an SSID scan.

Public Members

uint8_t ***ssid**

SSID of AP

uint8_t ***bssid**

MAC address of AP

uint8_t **channel**

channel, scan the specific channel

bool **show_hidden**

enable to scan AP whose SSID is hidden

wifi_scan_type_t **scan_type**

scan type, active or passive

wifi_scan_time_t **scan_time**

scan time per channel

struct wifi_ap_record_t

Description of a WiFi AP.

Public Members

uint8_t **bssid**[6]

MAC address of AP

uint8_t **ssid**[33]

SSID of AP

uint8_t **primary**

channel of AP

wifi_second_chan_t **second**

secondary channel of AP

int8_t **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

wifi_ant_t **ant**
antenna used to receive beacon from AP

uint32_t **phy_11b**
bit: 0 flag to identify if 11b mode is enabled or not

uint32_t **phy_11g**
bit: 1 flag to identify if 11g mode is enabled or not

uint32_t **phy_11n**
bit: 2 flag to identify if 11n mode is enabled or not

uint32_t **phy_lr**
bit: 3 flag to identify if low rate is enabled or not

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

uint32_t **reserved**
bit: 5..31 reserved

wifi_country_t **country**
country information of AP

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 10

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.

uint16_t **listen_interval**
Listen interval for ESP32 station to receive beacon when WIFI_PS_MAX_MODEM is set. Units: AP beacon intervals. Defaults to 3 if set to 0.

wifi_sort_method_t **sort_method**
sort the connect AP in the list by rssi or security mode

wifi_scan_threshold_t **threshold**
When scan_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

int8_t **rssi**
current average rssi of sta connected

uint32_t **phy_11b**
bit: 0 flag to identify if 11b mode is enabled or not

uint32_t **phy_11g**
bit: 1 flag to identify if 11g mode is enabled or not

uint32_t **phy_11n**
bit: 2 flag to identify if 11n mode is enabled or not

uint32_t **phy_lr**
bit: 3 flag to identify if low rate is enabled or not

uint32_t **reserved**
bit: 4..31 reserved

struct wifi_sta_list_t
List of stations associated with the ESP32 Soft-AP.

Public Members

wifi_sta_info_t **sta**[**ESP_WIFI_MAX_CONN_NUM**]
station list

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

struct vendor_ie_data_t
Vendor Information Element header.

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

Public Members

uint8_t **element_id**
Should be set to WIFI_VENDOR_IE_ELEMENT_ID (0xDD)

uint8_t **length**
Length of all bytes in the element data following this field. Minimum 4.

uint8_t **vendor_oui**[3]
Vendor identifier (OUI).

uint8_t **vendor_oui_type**
Vendor-specific OUI type.

uint8_t **payload**[0]
Payload. Length is equal to value in 'length' field, minus 4.

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

Public Members

signed **rssi**
Received Signal Strength Indicator(RSSI) of packet. unit: dBm

unsigned **rate**
PHY rate encoding of the packet. Only valid for non HT(11bg) packet

unsigned **__pad0__**
reserve

unsigned **sig_mode**
0: non HT(11bg) packet; 1: HT(11n) packet; 3: VHT(11ac) packet

unsigned **__pad1__**
reserve

unsigned **mcs**
Modulation Coding Scheme. If is HT(11n) packet, shows the modulation, range from 0 to 76(MSC0 ~ MCS76)

unsigned **cwb**
Channel Bandwidth of the packet. 0: 20MHz; 1: 40MHz

unsigned **__pad2__**
reserve

unsigned **smoothing**
reserve

unsigned **not_sounding**
reserve

unsigned **__pad3__**
reserve

unsigned **aggregation**
Aggregation. 0: MPDU packet; 1: AMPDU packet

unsigned **stbc**
Space Time Block Code(STBC). 0: non STBC packet; 1: STBC packet

unsigned **fec_coding**
Flag is set for 11n packets which are LDPC

unsigned **sgi**
Short Guide Interval(SGI). 0: Long GI; 1: Short GI

signed **noise_floor**
noise floor of Radio Frequency Module(RF). unit: 0.25dBm

unsigned **ampdu_cnt**
ampdu cnt

unsigned **channel**
primary channel on which this packet is received

unsigned **secondary_channel**
secondary channel on which this packet is received. 0: none; 1: above; 2: below

unsigned **__pad4__**
reserve

unsigned **timestamp**
timestamp. The local time when this packet is received. It is precise only if modem sleep or light sleep is not enabled. unit: microsecond

unsigned **__pad5__**
reserve

unsigned **__pad6__**
reserve

unsigned **ant**
antenna number from which this packet is received. 0: WiFi antenna 0; 1: WiFi antenna 1

unsigned **sig_len**
length of packet including Frame Check Sequence(FCS)

unsigned **__pad7__**
reserve

unsigned **rx_state**
state of the packet. 0: no error; others: error numbers which are not public

struct wifi_promiscuous_pkt_t
Payload passed to 'buf' parameter of promiscuous mode RX callback.

Public Members

wifi_pkt_rx_ctrl_t **rx_ctrl**
metadata header

uint8_t **payload**[0]
Data or management payload. Length of payload is described by rx_ctrl.sig_len. Type of content determined by packet type argument of callback.

struct wifi_promiscuous_filter_t
Mask for filtering different packet types in promiscuous mode.

Public Members

uint32_t **filter_mask**
OR of one or more filter values WIFI_PROMIS_FILTER_*

struct wifi_csi_config_t
Channel state information(CSI) configuration type.

Public Members

bool **lltf_en**
enable to receive legacy long training field(lltf) data. Default enabled

bool **htltf_en**
enable to receive HT long training field(htltf) data. Default enabled

bool **stbc_htltf2_en**
enable to receive space time block code HT long training field(stbc-htltf2) data. Default enabled

bool **ltf_merge_en**
enable to generate htlft data by averaging lltf and ht_ltf data when receiving HT packet. Otherwise, use ht_ltf data directly. Default enabled

bool **channel_filter_en**
enable to turn on channel filter to smooth adjacent sub-carrier. Disable it to keep independence of adjacent sub-carrier. Default enabled

bool **manu_scale**
manually scale the CSI data by left shifting or automatically scale the CSI data. If set true, please set the shift bits. false: automatically. true: manually. Default false

uint8_t **shift**
manually left shift bits of the scale of the CSI data. The range of the left shift bits is 0~15

struct wifi_csi_info_t

CSI data type.

Public Members

wifi_pkt_rx_ctrl_t **rx_ctrl**
received packet radio metadata header of the CSI data

uint8_t **mac**[6]
source MAC address of the CSI data

bool **first_word_invalid**
first four bytes of the CSI data is invalid or not

int8_t ***buf**
buffer of CSI data

uint16_t **len**
length of CSI data

struct wifi_ant_gpio_t

WiFi GPIO configuration for antenna selection.

Public Members

uint8_t **gpio_select**
Whether this GPIO is connected to external antenna switch

uint8_t **gpio_num**
The GPIO number that connects to external antenna switch

struct wifi_ant_gpio_config_t

WiFi GPIOs configuration for antenna selection.

Public Members

wifi_ant_gpio_t **gpio_cfg**[4]
The configurations of GPIOs that connect to external antenna switch

struct wifi_ant_config_t

WiFi antenna configuration.

Public Members

wifi_ant_mode_t **rx_ant_mode**
WiFi antenna mode for receiving

wifi_ant_t **rx_ant_default**
Default antenna mode for receiving, it's ignored if rx_ant_mode is not WIFI_ANT_MODE_AUTO

wifi_ant_mode_t **tx_ant_mode**
WiFi antenna mode for transmission, it can be set to WIFI_ANT_MODE_AUTO only if rx_ant_mode is set to WIFI_ANT_MODE_AUTO

uint8_t **enabled_ant0**
Index (in antenna GPIO configuration) of enabled WIFI_ANT_MODE_ANT0

uint8_t **enabled_ant1**
Index (in antenna GPIO configuration) of enabled WIFI_ANT_MODE_ANT1

struct wifi_ht2040_coex_t
Configuration for STA's HT2040 coexist management.

Public Members

int **enable**
Indicate whether STA's HT2040 coexist management is enabled or not

struct wifi_ioctl_config_t
Configuration for WiFi ioctl.

Public Members

wifi_ht2040_coex_t **ht2040_coex**
Configuration of STA's HT2040 coexist management

union *wifi_ioctl_config_t*::**[anonymous] data**
Configuration of ioctl command

Macros

WIFI_IF_STA

WIFI_IF_AP

WIFI_PS_MODEM

WIFI_PROTOCOL_11B

WIFI_PROTOCOL_11G

WIFI_PROTOCOL_11N

WIFI_PROTOCOL_LR

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

WIFI_VENDOR_IE_ELEMENT_ID

WIFI_PROMIS_FILTER_MASK_ALL
filter all packets

WIFI_PROMIS_FILTER_MASK_MGMT
filter the packets with type of WIFI_PKT_MGMT

WIFI_PROMIS_FILTER_MASK_CTRL
filter the packets with type of WIFI_PKT_CTRL

WIFI_PROMIS_FILTER_MASK_DATA
filter the packets with type of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_MISC
filter the packets with type of WIFI_PKT_MISC

WIFI_PROMIS_FILTER_MASK_DATA_MPDU
filter the MPDU which is a kind of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_DATA_AMPDU
filter the AMPDU which is a kind of WIFI_PKT_DATA

WIFI_PROMIS_CTRL_FILTER_MASK_ALL
filter all control packets

WIFI_PROMIS_CTRL_FILTER_MASK_WRAPPER
filter the control packets with subtype of Control Wrapper

WIFI_PROMIS_CTRL_FILTER_MASK_BAR
filter the control packets with subtype of Block Ack Request

WIFI_PROMIS_CTRL_FILTER_MASK_BA
filter the control packets with subtype of Block Ack

WIFI_PROMIS_CTRL_FILTER_MASK_PSPOLL
filter the control packets with subtype of PS-Poll

WIFI_PROMIS_CTRL_FILTER_MASK_RTS
filter the control packets with subtype of RTS

WIFI_PROMIS_CTRL_FILTER_MASK_CTS
filter the control packets with subtype of CTS

WIFI_PROMIS_CTRL_FILTER_MASK_ACK
filter the control packets with subtype of ACK

WIFI_PROMIS_CTRL_FILTER_MASK_CFEND
filter the control packets with subtype of CF-END

WIFI_PROMIS_CTRL_FILTER_MASK_CFENDACK
filter the control packets with subtype of CF-END+CF-ACK

WIFI_EVENT_MASK_ALL
mask all WiFi events

WIFI_EVENT_MASK_NONE
mask none of the WiFi events

WIFI_EVENT_MASK_AP_PROBEREQRECVED
mask SYSTEM_EVENT_AP_PROBEREQRECVED event

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 secondary channel is above the primary channel

WIFI_SECOND_CHAN_BELOW
the channel width is HT40 and the secondary channel is below the primary channel

enum wifi_scan_type_t

Values:

WIFI_SCAN_TYPE_ACTIVE = 0
active scan

WIFI_SCAN_TYPE_PASSIVE
passive scan

enum wifi_cipher_type_t

Values:

WIFI_CIPHER_TYPE_NONE = 0
the cipher type is none

WIFI_CIPHER_TYPE_WEP40
the cipher type is WEP40

WIFI_CIPHER_TYPE_WEP104
the cipher type is WEP104

WIFI_CIPHER_TYPE_TKIP
the cipher type is TKIP

WIFI_CIPHER_TYPE_CCMP
the cipher type is CCMP

WIFI_CIPHER_TYPE_TKIP_CCMP
the cipher type is TKIP and CCMP

WIFI_CIPHER_TYPE_UNKNOWN
the cipher type is unknown

enum wifi_ant_t
WiFi antenna.

Values:

WIFI_ANT_ANT0
WiFi antenna 0

WIFI_ANT_ANT1
WiFi antenna 1

WIFI_ANT_MAX
Invalid WiFi antenna

enum wifi_scan_method_t
Values:

WIFI_FAST_SCAN = 0
Do fast scan, scan will end after find SSID match AP

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

enum wifi_sort_method_t
Values:

WIFI_CONNECT_AP_BY_SIGNAL = 0
Sort match AP in scan list by RSSI

WIFI_CONNECT_AP_BY_SECURITY
Sort match AP in scan list by security mode

enum wifi_ps_type_t
Values:

WIFI_PS_NONE
No power save

WIFI_PS_MIN_MODEM
Minimum modem power saving. In this mode, station wakes up to receive beacon every DTIM period

WIFI_PS_MAX_MODEM
Maximum modem power saving. In this mode, interval to receive beacons is determined by the listen_interval parameter in *wifi_sta_config_t*

enum wifi_bandwidth_t
Values:

WIFI_BW_HT20 = 1

WIFI_BW_HT40

enum wifi_storage_t

Values:

WIFI_STORAGE_FLASH
all configuration will strore in both memory and flash

WIFI_STORAGE_RAM
all configuration will only store in the memory

enum wifi_vendor_ie_type_t

Vendor Information Element type.

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

Values:

WIFI_VND_IE_TYPE_BEACON

WIFI_VND_IE_TYPE_PROBE_REQ

WIFI_VND_IE_TYPE_PROBE_RESP

WIFI_VND_IE_TYPE_ASSOC_REQ

WIFI_VND_IE_TYPE_ASSOC_RESP

enum wifi_vendor_ie_id_t

Vendor Information Element index.

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

Values:

WIFI_VND_IE_ID_0

WIFI_VND_IE_ID_1

enum wifi_promiscuous_pkt_type_t

Promiscuous frame type.

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

Values:

WIFI_PKT_MGMT
Management frame, indicates 'buf' argument is *wifi_promiscuous_pkt_t*

WIFI_PKT_CTRL
Control frame, indicates 'buf' argument is *wifi_promiscuous_pkt_t*

WIFI_PKT_DATA
Data frame, indiciates 'buf' argument is *wifi_promiscuous_pkt_t*

WIFI_PKT_MISC
Other type, such as MIMO etc. 'buf' argument is *wifi_promiscuous_pkt_t* but the payload is zero length.

enum wifi_ant_mode_t

WiFi antenna mode.

Values:

WIFI_ANT_MODE_ANT0
Enable WiFi antenna 0 only

WIFI_ANT_MODE_ANT1
Enable WiFi antenna 1 only

WIFI_ANT_MODE_AUTO
Enable WiFi antenna 0 and 1, automatically select an antenna

WIFI_ANT_MODE_MAX
Invalid WiFi enabled antenna

enum wifi_ioctl_cmd_t
WiFi ioctl command type.

Values:

WIFI_IOCTL_SET_STA_HT2040_COEX = 1
Set the configuration of STA's HT2040 coexist management

WIFI_IOCTL_GET_STA_HT2040_COEX
Get the configuration of STA's HT2040 coexist management

WIFI_IOCTL_MAX

3.1.2 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 to *wifi_config_t*.
  - 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

3.1.3 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:

```
-------------------------------------------------------------------------------------
↪--
| MAC Header | Category Code | Organization Identifier | Vendor Specific Content |␣
↪FCS |
-------------------------------------------------------------------------------------
↪--
                    1 byte              3 bytes                   7~255 bytes
```

- 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 | Length | Organization Identifier | Type | Version |    Body    |
-------------------------------------------------------------------------------
    1 byte     1 byte            3 bytes         1 byte   1 byte   0~250 bytes
```

- 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 transmiting 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.

3.2 Mesh API

3.2.1 ESP-MESH Programming Guide

This is a programming guide for ESP-MESH, including the API reference and coding examples. This guide is split into the following parts:

1. *ESP-MESH Programming Model*
2. *Writing an ESP-MESH Application*
3. *Application Examples*
4. *API Reference*

For documentation regarding the ESP-MESH protocol, please see the *ESP-MESH API Guide*.

ESP-MESH Programming Model

Software Stack

The ESP-MESH software stack is built atop the Wi-Fi Driver/FreeRTOS and may use the LwIP Stack in some instances (i.e. the root node). The following diagram illustrates the ESP-MESH software stack.

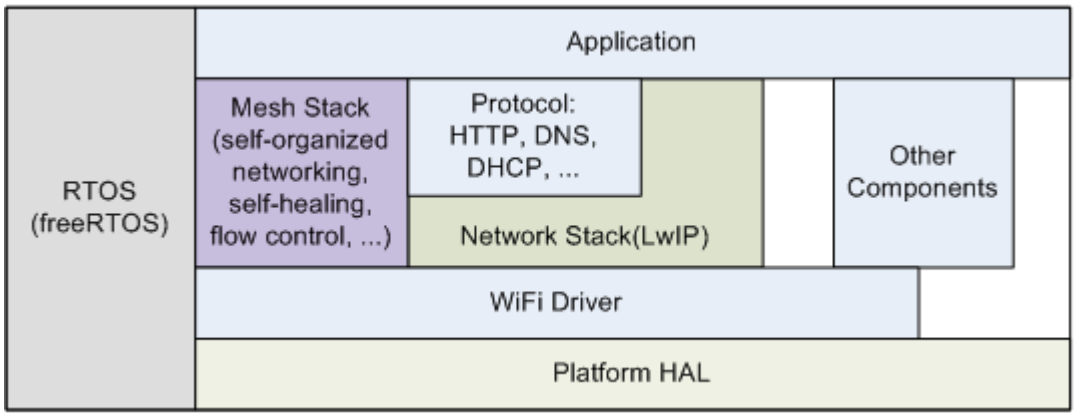


Fig. 1: ESP-MESH Software Stack

System Events

An application interfaces with ESP-MESH via **ESP-MESH Events**. Since ESP-MESH is built atop the Wi-Fi stack, it is also possible for the application to interface with the Wi-Fi driver via the **Wi-Fi Event Task**. The following diagram illustrates the interfaces for the various System Events in an ESP-MESH application.

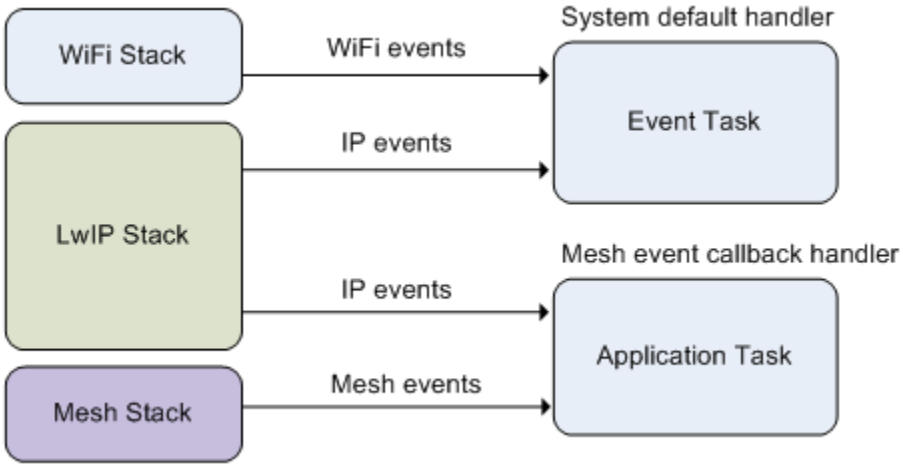


Fig. 2: ESP-MESH System Events Delivery

The `mesh_event_id_t` defines all possible ESP-MESH system events and can indicate events such as the connection/disconnection of parent/child. Before ESP-MESH system events can be used, the application must register a **Mesh Event Callback** via `esp_mesh_set_config()`. The callback is used to receive events from the ESP-MESH stack as well as the LwIP Stack and should contain handlers for each event relevant to the application.

Typical use cases of system events include using events such as *MESH_EVENT_PARENT_CONNECTED* and *MESH_EVENT_CHILD_CONNECTED* to indicate when a node can begin transmitting data upstream and downstream respectively. Likewise, *MESH_EVENT_ROOT_GOT_IP* and *MESH_EVENT_ROOT_LOST_IP* can be used to indicate when the root node can and cannot transmit data to the external IP network.

Warning: When using ESP-MESH under self-organized mode, users must ensure that no calls to Wi-Fi API are made. This is due to the fact that the self-organizing mode will internally make Wi-Fi API calls to connect/disconnect/scan etc. **Any Wi-Fi calls from the application (including calls from callbacks and handlers of Wi-Fi events) may interfere with ESP-MESH's self-organizing behavior**. Therefore, user's should not call Wi-Fi APIs after *esp_mesh_start()* is called, and before *esp_mesh_stop()* is called.

LwIP & ESP-MESH

The application can access the ESP-MESH stack directly without having to go through the LwIP stack. The LwIP stack is only required by the root node to transmit/receive data to/from an external IP network. However, since every node can potentially become the root node (due to automatic root node selection), each node must still initialize the LwIP stack.

Each node is required to initialize LwIP by calling `tcpip_adapter_init()`. In order to prevent non-root node access to LwIP, the application should stop the following services after LwIP initialization:

- DHCP server service on the softAP interface.
- DHCP client service on the station interface.

The following code snippet demonstrates how to initialize LwIP for ESP-MESH applications.

```
/*  tcpip initialization */
tcpip_adapter_init();
/*
 * for mesh
 * stop DHCP server on softAP interface by default
 * stop DHCP client on station interface by default
 */
ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP));
ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));
/* do not specify system event callback, use NULL instead. */
ESP_ERROR_CHECK(esp_event_loop_init(NULL, NULL));
```

Note: ESP-MESH requires a root node to be connected with a router. Therefore, in the event that a node becomes the root, **the corresponding handler must start the DHCP client service and immediately obtain an IP address**. Doing so will allow other nodes to begin transmitting/receiving packets to/from the external IP network. However, this step is unnecessary if static IP settings are used.

Writing an ESP-MESH Application

The prerequisites for starting ESP-MESH is to initialize LwIP and Wi-Fi, The following code snippet demonstrates the necessary prerequisite steps before ESP-MESH itself can be initialized.

```
tcpip_adapter_init();
/*
 * for mesh
```

(continues on next page)

(continued from previous page)

```
 * stop DHCP server on softAP interface by default
 * stop DHCP client on station interface by default
 */
ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP));
ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));
/* do not specify system event callback, use NULL instead. */
ESP_ERROR_CHECK(esp_event_loop_init(NULL, NULL));

/*  Wi-Fi initialization */
wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
ESP_ERROR_CHECK(esp_wifi_init(&config));
ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_FLASH));
ESP_ERROR_CHECK(esp_wifi_start());
```

After initializing LwIP and Wi-Fi, the process of getting an ESP-MESH network up and running can be summarized into the following three steps:

1. *Initialize Mesh*
2. *Configuring an ESP-MESH Network*
3. *Start Mesh*

Initialize Mesh

The following code snippet demonstrates how to initialize ESP-MESH

```
/*  mesh initialization */
ESP_ERROR_CHECK(esp_mesh_init());
```

Configuring an ESP-MESH Network

ESP-MESH is configured via `esp_mesh_set_config()` which receives its arguments using the `mesh_cfg_t` structure. The structure contains the following parameters used to configure ESP-MESH:

Parameter	Description
Channel	Range from 1 to 14
Event Callback	Callback for Mesh Events, see `mesh_event_cb_t`
Mesh ID	ID of ESP-MESH Network, see `mesh_addr_t`
Router	Router Configuration, see `mesh_router_t`
Mesh AP	Mesh AP Configuration, see `mesh_ap_cfg_t`
Crypto Functions	Crypto Functions for Mesh IE, see `mesh_crypto_funcs_t`

The following code snippet demonstrates how to configure ESP-MESH.

```
/* Enable the Mesh IE encryption by default */
mesh_cfg_t cfg = MESH_INIT_CONFIG_DEFAULT();
/* mesh ID */
memcpy((uint8_t *) &cfg.mesh_id, MESH_ID, 6);
/* mesh event callback */
cfg.event_cb = &mesh_event_handler;
/* channel (must match the router's channel) */
```

(continues on next page)

(continued from previous page)

```
cfg.channel = CONFIG_MESH_CHANNEL;
/* router */
cfg.router.ssid_len = strlen(CONFIG_MESH_ROUTER_SSID);
memcpy((uint8_t *) &cfg.router.ssid, CONFIG_MESH_ROUTER_SSID, cfg.router.ssid_len);
memcpy((uint8_t *) &cfg.router.password, CONFIG_MESH_ROUTER_PASSWD,
       strlen(CONFIG_MESH_ROUTER_PASSWD));
/* mesh softAP */
cfg.mesh_ap.max_connection = CONFIG_MESH_AP_CONNECTIONS;
memcpy((uint8_t *) &cfg.mesh_ap.password, CONFIG_MESH_AP_PASSWD,
       strlen(CONFIG_MESH_AP_PASSWD));
ESP_ERROR_CHECK(esp_mesh_set_config(&cfg));
```

Start Mesh

The following code snippet demonstrates how to start ESP-MESH.

```
/* mesh start */
ESP_ERROR_CHECK(esp_mesh_start());
```

After starting ESP-MESH, the application should check for ESP-MESH events to determine when it has connected to the network. After connecting, the application can start transmitting and receiving packets over the ESP-MESH network using `esp_mesh_send()` and `esp_mesh_recv()`.

Application Examples

ESP-IDF contains these ESP-MESH example projects:

The Internal Communication Example demonstrates how to setup a ESP-MESH network and have the root node send a data packet to every node within the network.

The Manual Networking Example demonstrates how to use ESP-MESH without the self-organizing features. This example shows how to program a node to manually scan for a list of potential parent nodes and select a parent node based on custom criteria.

API Reference

Header File

- esp32/include/esp_mesh.h

Functions

esp_err_t **esp_mesh_init** (void)

Mesh initialization.

- Check whether Wi-Fi is started.
- Initialize mesh global variables with default values.

Attention This API shall be called after Wi-Fi is started.

Return

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_deinit** (void)

Mesh de-initialization.

- Release resources and stop the mesh

Return

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_start** (void)

Start mesh.

- Initialize mesh IE.
- Start mesh network management service.
- Create TX and RX queues according to the configuration.
- Register mesh packets receive callback.

Attention This API shall be called after mesh initialization and configuration.

Return

- ESP_OK
- ESP_FAIL
- ESP_ERR_MESH_NOT_INIT
- ESP_ERR_MESH_NOT_CONFIG
- ESP_ERR_MESH_NO_MEMORY

esp_err_t **esp_mesh_stop** (void)

Stop mesh.

- Deinitialize mesh IE.
- Disconnect with current parent.
- Disassociate all currently associated children.
- Stop mesh network management service.
- Unregister mesh packets receive callback.
- Delete TX and RX queues.
- Release resources.
- Restore Wi-Fi softAP to default settings if Wi-Fi dual mode is enabled.

Return

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_send**(**const** *mesh_addr_t* **to*, **const** *mesh_data_t* **data*, int *flag*, **const** *mesh_opt_t opt*[], int *opt_count*)

Send a packet over the mesh network.

- Send a packet to any device in the mesh network.
- Send a packet to external IP network.

Attention This API is not reentrant.

Return

- ESP_OK
- ESP_FAIL
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_DISCONNECTED
- ESP_ERR_MESH_OPT_UNKNOWN
- ESP_ERR_MESH_EXCEED_MTU
- ESP_ERR_MESH_NO_MEMORY
- ESP_ERR_MESH_TIMEOUT
- ESP_ERR_MESH_QUEUE_FULL
- ESP_ERR_MESH_NO_ROUTE_FOUND
- ESP_ERR_MESH_DISCARD

Parameters

- `to`: the address of the final destination of the packet
  - If the packet is to the root, set this parameter to NULL.
  - If the packet is to an external IP network, set this parameter to the IPv4:PORT combination. This packet will be delivered to the root firstly, then the root will forward this packet to the final IP server address.
- `data`: pointer to a sending mesh packet
  - Field proto should be set to data protocol in use (default is MESH_PROTO_BIN for binary).
  - Field tos should be set to transmission tos (type of service) in use (default is MESH_TOS_P2P for point-to-point reliable).
- `flag`: bitmap for data sent
  - Speed up the route search
    - If the packet is to the root and "to" parameter is NULL, set this parameter to 0.
    - If the packet is to an internal device, MESH_DATA_P2P should be set.
    - If the packet is to the root ("to" parameter isn't NULL) or to external IP network, MESH_DATA_TODS should be set.
    - If the packet is from the root to an internal device, MESH_DATA_FROMDS should be set.
  - Specify whether this API is block or non-block, block by default
    - If needs non-block, MESH_DATA_NONBLOCK should be set.

  - In the situation of the root change, MESH_DATA_DROP identifies this packet can be dropped by the new root for upstream data to external IP network, we try our best to avoid data loss caused by the root change, but there is a risk that the new root is running out of memory because most of memory is occupied by the pending data which isn't read out in time by esp_mesh_recv_toDS().

    Generally, we suggest esp_mesh_recv_toDS() is called after a connection with IP network is created. Thus data outgoing to external IP network via socket is just from reading esp_mesh_recv_toDS() which avoids unnecessary memory copy.

- `opt`: options
  - In case of sending a packet to a certain group, MESH_OPT_SEND_GROUP is a good choice. In this option, the value field should be set to the target receiver addresses in this group.
  - Root sends a packet to an internal device, this packet is from external IP network in case the receiver device responds this packet, MESH_OPT_RECV_DS_ADDR is required to attach the target DS address.
- `opt_count`: option count
  - Currently, this API only takes one option, so opt_count is only supported to be 1.

esp_err_t **esp_mesh_recv**(*mesh_addr_t *from*, *mesh_data_t *data*, int *timeout_ms*, int **flag*, *mesh_opt_t opt*[], int *opt_count*)

Receive a packet targeted to self over the mesh network.

flag could be MESH_DATA_FROMDS or MESH_DATA_TODS.

Attention Mesh RX queue should be checked regularly to avoid running out of memory.

- Use esp_mesh_get_rx_pending() to check the number of packets available in the queue waiting to be received by applications.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_TIMEOUT
- ESP_ERR_MESH_DISCARD

Parameters

- `from`: the address of the original source of the packet
- `data`: pointer to the received mesh packet
  - Field proto is the data protocol in use. Should follow it to parse the received data.
  - Field tos is the transmission tos (type of service) in use.
- `timeout_ms`: wait time if a packet isn't immediately available (0:no wait, portMAX_DELAY:wait forever)
- `flag`: bitmap for data received
  - MESH_DATA_FROMDS represents data from external IP network
  - MESH_DATA_TODS represents data directed upward within the mesh network

Parameters

- `opt`: options desired to receive

  - MESH_OPT_RECV_DS_ADDR attaches the DS address
- `opt_count`: option count desired to receive
  - Currently, this API only takes one option, so opt_count is only supported to be 1.

esp_err_t **esp_mesh_recv_toDS**(*mesh_addr_t *from*, *mesh_addr_t *to*, *mesh_data_t *data*, int *timeout_ms*, int **flag*, *mesh_opt_t opt*[], int *opt_count*)

Receive a packet targeted to external IP network.

- Root uses this API to receive packets destined to external IP network
- Root forwards the received packets to the final destination via socket.
- If no socket connection is ready to send out the received packets and this esp_mesh_recv_toDS() hasn't been called by applications, packets from the whole mesh network will be pending in toDS queue.

Use esp_mesh_get_rx_pending() to check the number of packets available in the queue waiting to be received by applications in case of running out of memory in the root.

Using esp_mesh_set_xon_qsize() users may configure the RX queue size, default:32. If this size is too large, and esp_mesh_recv_toDS() isn't called in time, there is a risk that a great deal of memory is occupied by the pending packets. If this size is too small, it will impact the efficiency on upstream. How to decide this value depends on the specific application scenarios.

flag could be MESH_DATA_TODS.

Attention This API is only called by the root.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_TIMEOUT
- ESP_ERR_MESH_DISCARD

Parameters

- `from`: the address of the original source of the packet
- `to`: the address contains remote IP address and port (IPv4:PORT)
- `data`: pointer to the received packet
  - Contain the protocol and applications should follow it to parse the data.
- `timeout_ms`: wait time if a packet isn't immediately available (0:no wait, portMAX_DELAY:wait forever)
- `flag`: bitmap for data received
  - MESH_DATA_TODS represents the received data target to external IP network. Root shall forward this data to external IP network via the association with router.

Parameters

- `opt`: options desired to receive
- `opt_count`: option count desired to receive

esp_err_t **esp_mesh_set_config**(**const** *mesh_cfg_t *config*)

Set mesh stack configuration.

- Use MESH_INIT_CONFIG_DEFAULT() to initialize the default values, mesh IE is encrypted by default.
- Mesh network is established on a fixed channel (1-14).
- Mesh event callback is mandatory.
- Mesh ID is an identifier of an MBSS. Nodes with the same mesh ID can communicate with each other.
- Regarding to the router configuration, if the router is hidden, BSSID field is mandatory.

If BSSID field isn't set and there exists more than one router with same SSID, there is a risk that more roots than one connected with different BSSID will appear. It means more than one mesh network is established with the same mesh ID.

Root conflict function could eliminate redundant roots connected with the same BSSID, but couldn't handle roots connected with different BSSID. Because users might have such requirements of setting up routers with same SSID for the future replacement. But in that case, if the above situations happen, please make sure applications implement forward functions on the root to guarantee devices in different mesh networks can communicate with each other. max_connection of mesh softAP is limited by the max number of Wi-Fi softAP supported (max:10).

Attention This API shall be called before mesh is started after mesh is initialized.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- `config`: pointer to mesh stack configuration

esp_err_t **esp_mesh_get_config**(*mesh_cfg_t *config*)

Get mesh stack configuration.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- `config`: pointer to mesh stack configuration

esp_err_t **esp_mesh_set_router**(**const** *mesh_router_t *router*)

Get router configuration.

Attention This API is used to dynamically modify the router configuration after mesh is configured.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- `router`: pointer to router configuration

esp_err_t **esp_mesh_get_router**(*mesh_router_t* **router*)
Get router configuration.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- `router`: pointer to router configuration

esp_err_t **esp_mesh_set_id**(**const** *mesh_addr_t* **id*)
Set mesh network ID.

Attention This API is used to dynamically modify the mesh network ID.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT: invalid argument

Parameters

- `id`: pointer to mesh network ID

esp_err_t **esp_mesh_get_id**(*mesh_addr_t* **id*)
Get mesh network ID.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- `id`: pointer to mesh network ID

esp_err_t **esp_mesh_set_type**(*mesh_type_t type*)
Designate device type over the mesh network.

- MESH_ROOT: designates the root node for a mesh network
- MESH_LEAF: designates a device as a standalone Wi-Fi station

Return

- ESP_OK
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- `type`: device type

mesh_type_t **esp_mesh_get_type**(void)
Get device type over mesh network.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Return mesh type

esp_err_t **esp_mesh_set_max_layer** (int *max_layer*)
Set network max layer value (max:25, default:25)

- Network max layer limits the max hop count.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- `max_layer`: max layer value

int **esp_mesh_get_max_layer** (void)
Get max layer value.

Return max layer value

esp_err_t **esp_mesh_set_ap_password** (**const** uint8_t **pwd*, int *len*)
Set mesh softAP password.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- `pwd`: pointer to the password
- `len`: password length

esp_err_t **esp_mesh_set_ap_authmode** (*wifi_auth_mode_t authmode*)
Set mesh softAP authentication mode.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- `authmode`: authentication mode

wifi_auth_mode_t **esp_mesh_get_ap_authmode**(void)
Get mesh softAP authentication mode.

Return authentication mode

esp_err_t **esp_mesh_set_ap_connections**(int *connections*)
Set mesh softAP max connection value.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- `connections`: the number of max connections

int **esp_mesh_get_ap_connections**(void)
Get mesh softAP max connection configuration.

Return the number of max connections

int **esp_mesh_get_layer**(void)
Get current layer value over the mesh network.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Return layer value

esp_err_t **esp_mesh_get_parent_bssid**(*mesh_addr_t* **bssid*)
Get the parent BSSID.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `bssid`: pointer to parent BSSID

bool **esp_mesh_is_root**(void)
Return whether the device is the root node of the network.

Return true/false

esp_err_t **esp_mesh_set_self_organized**(bool *enable*, bool *select_parent*)
Enable/disable self-organized networking.

- Self-organized networking has three main functions: select the root node; find a preferred parent; initiate reconnection if a disconnection is detected.
- Self-organized networking is enabled by default.

- If self-organized is disabled, users should set a parent for the device via esp_mesh_set_parent().

Attention This API is used to dynamically modify whether to enable the self organizing.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `enable`: enable or disable self-organized networking
- `select_parent`:
  - If self-organized networking is enabled, let the device search for a new parent or keep connecting to the previous parent.

bool **esp_mesh_get_self_organized**(void)

Return whether enable self-organized networking or not.

Return true/false

esp_err_t **esp_mesh_waive_root**(**const** *mesh_vote_t* **vote*, int *reason*)

Cause the root device to give up (waive) its mesh root status.

- A device is elected root primarily based on RSSI from the external router.
- If external router conditions change, users can call this API to perform a root switch.
- In this API, users could specify a desired root address to replace itself or specify an attempts value to ask current root to initiate a new round of voting. During the voting, a better root candidate would be expected to find to replace the current one.
- If no desired root candidate, the vote will try a specified number of attempts (at least 15). If no better root candidate is found, keep the current one. If a better candidate is found, the new better one will send a root switch request to the current root, current root will respond with a root switch acknowledgment.
- After that, the new candidate will connect to the router to be a new root, the previous root will disconnect with the router and choose another parent instead.

Root switch is completed with minimal disruption to the whole mesh network.

Attention This API is only called by the root.

Return

- ESP_OK
- ESP_ERR_MESH_QUEUE_FULL
- ESP_ERR_MESH_DISCARD
- ESP_FAIL

Parameters

- `vote`: vote configuration
  - If this parameter is set NULL, the vote will perform the default 15 times.
  - Field percentage threshold is 0.9 by default.

  - Field is_rc_specified shall be false.
  - Field attempts shall be at least 15 times.
- `reason`: only accept MESH_VOTE_REASON_ROOT_INITIATED for now

esp_err_t **esp_mesh_set_vote_percentage** (float *percentage*)
Set vote percentage threshold for approval of being a root.

- During the networking, only obtaining vote percentage reaches this threshold, the device could be a root.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `percentage`: vote percentage threshold

float **esp_mesh_get_vote_percentage** (void)
Get vote percentage threshold for approval of being a root.

Return percentage threshold

esp_err_t **esp_mesh_set_ap_assoc_expire** (int *seconds*)
Set mesh softAP associate expired time.

- If mesh softAP hasn't received any data from an associated child within this time, mesh softAP will take this child inactive and disassociate it.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `seconds`: the expired time

int **esp_mesh_get_ap_assoc_expire** (void)
Get mesh softAP associate expired time.

Return seconds

int **esp_mesh_get_total_node_num** (void)
Get total number of devices in current network (including the root)

Attention The returned value might be incorrect when the network is changing.

Return total number of devices (including the root)

int **esp_mesh_get_routing_table_size** (void)
Get the number of devices in this device's sub-network (including self)

Return the number of devices over this device's sub-network (including self)

esp_err_t **esp_mesh_get_routing_table**(*mesh_addr_t* **mac*, int *len*, int **size*)
Get routing table of this device's sub-network (including itself)

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- `mac`: pointer to routing table
- `len`: routing table size(in bytes)
- `size`: pointer to the number of devices in routing table (including itself)

esp_err_t **esp_mesh_post_toDS_state**(bool *reachable*)
Post the toDS state to the mesh stack.

Attention This API is only for the root.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `reachable`: this state represents whether the root is able to access external IP network

esp_err_t **esp_mesh_get_tx_pending**(*mesh_tx_pending_t* **pending*)
Return the number of packets pending in the queue waiting to be sent by the mesh stack.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `pending`: pointer to the TX pending

esp_err_t **esp_mesh_get_rx_pending**(*mesh_rx_pending_t* **pending*)
Return the number of packets available in the queue waiting to be received by applications.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `pending`: pointer to the RX pending

int **esp_mesh_available_txupQ_num**(**const** *mesh_addr_t* **addr*, uint32_t **xseqno_in*)
Return the number of packets could be accepted from the specified address.

Return the number of upQ for a certain address

Parameters

- `addr`: self address or an associate children address
- `xseqno_in`: sequence number of the last received packet from the specified address

esp_err_t **esp_mesh_set_xon_qsize**(int *qsize*)
Set the number of queue.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `qsize`: default:32 (min:16)

int **esp_mesh_get_xon_qsize**(void)
Get queue size.

Return the number of queue

esp_err_t **esp_mesh_allow_root_conflicts**(bool *allowed*)
Set whether allow more than one root existing in one network.

Return

- ESP_OK
- ESP_WIFI_ERR_NOT_INIT
- ESP_WIFI_ERR_NOT_START

Parameters

- `allowed`: allow or not

bool **esp_mesh_is_root_conflicts_allowed**(void)
Check whether allow more than one root to exist in one network.

Return true/false

esp_err_t **esp_mesh_set_group_id**(**const** *mesh_addr_t* **addr*, int *num*)
Set group ID addresses.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- `addr`: pointer to new group ID addresses
- `num`: the number of group ID addresses

esp_err_t **esp_mesh_delete_group_id**(**const** *mesh_addr_t* **addr*, int *num*)
Delete group ID addresses.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- `addr`: pointer to deleted group ID address
- `num`: the number of group ID addresses

int **esp_mesh_get_group_num**(void)

Get the number of group ID addresses.

Return the number of group ID addresses

esp_err_t **esp_mesh_get_group_list**(*mesh_addr_t* **addr*, int *num*)

Get group ID addresses.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- `addr`: pointer to group ID addresses
- `num`: the number of group ID addresses

bool **esp_mesh_is_my_group**(**const** *mesh_addr_t* **addr*)

Check whether the specified group address is my group.

Return true/false

esp_err_t **esp_mesh_set_capacity_num**(int *num*)

Set mesh network capacity.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_NOT_ALLOWED
- ESP_MESH_ERR_ARGUMENT

Parameters

- `num`: mesh network capacity

int **esp_mesh_get_capacity_num**(void)

Get mesh network capacity.

Return mesh network capacity

esp_err_t **esp_mesh_set_ie_crypto_funcs**(**const** mesh_crypto_funcs_t **crypto_funcs*)

Set mesh IE crypto functions.

Attention This API can be called at any time after mesh is initialized.

Return

- ESP_OK

Parameters

- `crypto_funcs`: crypto functions for mesh IE
  - If crypto_funcs is set to NULL, mesh IE is no longer encrypted.

esp_err_t **esp_mesh_set_ie_crypto_key**(**const** char **key*, int *len*)

Set mesh IE crypto key.

Attention This API can be called at any time after mesh is initialized.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- `key`: ASCII crypto key
- `len`: length in bytes, range:8~64

esp_err_t **esp_mesh_get_ie_crypto_key**(char **key*, int *len*)

Get mesh IE crypto key.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- `key`: ASCII crypto key
- `len`: length in bytes, range:8~64

esp_err_t **esp_mesh_set_root_healing_delay**(int *delay_ms*)

Set delay time before starting root healing.

Return

- ESP_OK

Parameters

- `delay_ms`: delay time in milliseconds

int **esp_mesh_get_root_healing_delay**(void)

Get delay time before network starts root healing.

Return delay time in milliseconds

esp_err_t **esp_mesh_set_event_cb**(**const** *mesh_event_cb_t* *event_cb*)

Set mesh event callback.

Return

- ESP_OK

Parameters

- `event_cb`: mesh event call back

esp_err_t **esp_mesh_fix_root**(bool *enable*)

Enable network Fixed Root Setting.

- Enabling fixed root disables automatic election of the root node via voting.
- All devices in the network shall use the same Fixed Root Setting (enabled or disabled).
- If Fixed Root is enabled, users should make sure a root node is designated for the network.

Return

- ESP_OK

Parameters

- `enable`: enable or not

bool **esp_mesh_is_root_fixed**(void)

Check whether network Fixed Root Setting is enabled.

- Enable/disable network Fixed Root Setting by API esp_mesh_fix_root().
- Network Fixed Root Setting also changes with the "flag" value in parent networking IE.

Return true/false

esp_err_t **esp_mesh_set_parent**(**const** *wifi_config_t* **parent*, **const** *mesh_addr_t* **parent_mesh_id*, *mesh_type_t my_type*, int *my_layer*)

Set a specified parent for the device.

Attention This API can be called at any time after mesh is configured.

Return

- ESP_OK
- ESP_ERR_ARGUMENT
- ESP_ERR_MESH_NOT_CONFIG

Parameters

- `parent`: parent configuration, the SSID and the channel of the parent are mandatory.
  - If the BSSID is set, make sure that the SSID and BSSID represent the same parent, otherwise the device will never find this specified parent.
- `parent_mesh_id`: parent mesh ID,
  - If this value is not set, the original mesh ID is used.
- `my_type`: mesh type
  - If the parent set for the device is the same as the router in the network configuration, then my_type shall set MESH_ROOT and my_layer shall set MESH_ROOT_LAYER.
- `my_layer`: mesh layer

- my_layer of the device may change after joining the network.
- If my_type is set MESH_NODE, my_layer shall be greater than MESH_ROOT_LAYER.
- If my_type is set MESH_LEAF, the device becomes a standalone Wi-Fi station and no longer has the ability to extend the network.

esp_err_t **esp_mesh_scan_get_ap_ie_len**(int **len*)

Get mesh networking IE length of one AP.

Return

- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_WIFI_ARG
- ESP_ERR_WIFI_FAIL

Parameters

- `len`: mesh networking IE length

esp_err_t **esp_mesh_scan_get_ap_record**(*wifi_ap_record_t* **ap_record*, void **buffer*)

Get AP record.

Attention Different from esp_wifi_scan_get_ap_records(), this API only gets one of APs scanned each time. See "manual_networking" example.

Return

- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_WIFI_ARG
- ESP_ERR_WIFI_FAIL

Parameters

- `ap_record`: pointer to one AP record
- `buffer`: pointer to the mesh networking IE of this AP

esp_err_t **esp_mesh_flush_upstream_packets**(void)

Flush upstream packets pending in to_parent queue and to_parent_p2p queue.

Return

- ESP_OK

esp_err_t **esp_mesh_get_subnet_nodes_num**(**const** *mesh_addr_t* **child_mac*, int **nodes_num*)

Get the number of nodes in the subnet of a specific child.

Return

- ESP_OK
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_ARGUMENT

Parameters

- `child_mac`: an associated child address of this device
- `nodes_num`: pointer to the number of nodes in the subnet of a specific child

esp_err_t **esp_mesh_get_subnet_nodes_list**(**const** *mesh_addr_t* **child_mac*, *mesh_addr_t* **nodes*, int *nodes_num*)

Get nodes in the subnet of a specific child.

Return

- ESP_OK
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_ARGUMENT

Parameters

- `child_mac`: an associated child address of this device
- `nodes`: pointer to nodes in the subnet of a specific child
- `nodes_num`: the number of nodes in the subnet of a specific child

Unions

union mesh_addr_t

#include <esp_mesh.h> Mesh address.

Public Members

uint8_t **addr**[6]
mac address

mip_t **mip**
mip address

union mesh_event_info_t

#include <esp_mesh.h> Mesh event information.

Public Members

mesh_event_channel_switch_t **channel_switch**
channel switch

mesh_event_child_connected_t **child_connected**
child connected

mesh_event_child_disconnected_t **child_disconnected**
child disconnected

mesh_event_routing_table_change_t **routing_table**
routing table change

mesh_event_connected_t **connected**
parent connected

mesh_event_disconnected_t **disconnected**
parent disconnected

mesh_event_no_parent_found_t **no_parent**
no parent found

mesh_event_layer_change_t **layer_change**
layer change

mesh_event_toDS_state_t **toDS_state**
toDS state, devices shall check this state firstly before trying to send packets to external IP network. This state indicates right now whether the root is capable of sending packets out. If not, devices had better to wait until this state changes to be MESH_TODS_REACHABLE.

mesh_event_vote_started_t **vote_started**
vote started

mesh_event_root_got_ip_t **got_ip**
root obtains IP address

mesh_event_root_address_t **root_addr**
root address

mesh_event_root_switch_req_t **switch_req**
root switch request

mesh_event_root_conflict_t **root_conflict**
other powerful root

mesh_event_root_fixed_t **root_fixed**
fixed root

mesh_event_scan_done_t **scan_done**
scan done

union mesh_rc_config_t
#include <esp_mesh.h> Vote address configuration.

Public Members

int **attempts**
max vote attempts before a new root is elected automatically by mesh network. (min:15, 15 by default)

mesh_addr_t **rc_addr**
a new root address specified by users for API esp_mesh_waive_root()

Structures

struct mip_t
IP address and port.

Public Members

ip4_addr_t **ip4**
IP address

uint16_t **port**
port

struct mesh_event_channel_switch_t
Channel switch information.

Public Members

uint8_t **channel**
new channel

struct mesh_event_connected_t
Parent connected information.

Public Members

system_event_sta_connected_t **connected**
parent information, same as Wi-Fi event SYSTEM_EVENT_STA_CONNECTED does

uint8_t **self_layer**
layer

struct mesh_event_no_parent_found_t
No parent found information.

Public Members

int **scan_times**
scan times being through

struct mesh_event_layer_change_t
Layer change information.

Public Members

uint8_t **new_layer**
new layer

struct mesh_event_vote_started_t
vote started information

Public Members

int **reason**
vote reason, vote could be initiated by children or by the root itself

int **attempts**
max vote attempts before stopped

mesh_addr_t **rc_addr**
root address specified by users via API esp_mesh_waive_root()

struct mesh_event_root_switch_req_t
Root switch request information.

Public Members

int **reason**
root switch reason, generally root switch is initialized by users via API esp_mesh_waive_root()

mesh_addr_t **rc_addr**
the address of root switch requester

struct mesh_event_root_conflict_t
Other powerful root address.

Public Members

int8_t **rssi**
rssi with router

uint16_t **capacity**
the number of devices in current network

uint8_t **addr**[6]
other powerful root address

struct mesh_event_routing_table_change_t
Routing table change.

Public Members

uint16_t **rt_size_new**
the new value

uint16_t **rt_size_change**
the changed value

struct mesh_event_root_fixed_t
Root fixed.

Public Members

bool **is_fixed**
status

struct mesh_event_scan_done_t
Scan doneevent information.

Public Members

uint8_t **number**
the number of APs scanned

struct mesh_event_t
Mesh event.

Public Members

mesh_event_id_t **id**
mesh event id

mesh_event_info_t **info**
mesh event info

struct mesh_opt_t
Mesh option.

Public Members

uint8_t **type**
option type

uint16_t **len**
option length

uint8_t ***val**
option value

struct mesh_data_t
Mesh data for esp_mesh_send() and esp_mesh_recv()

Public Members

uint8_t ***data**
data

uint16_t **size**
data size

mesh_proto_t **proto**
data protocol

mesh_tos_t **tos**
data type of service

struct mesh_router_t
Router configuration.

Public Members

uint8_t **ssid**[32]
SSID

uint8_t **ssid_len**
length of SSID

uint8_t **bssid**[6]
BSSID, if router is hidden, this value is mandatory

uint8_t **password**[64]
password

struct mesh_ap_cfg_t
Mesh softAP configuration.

Public Members

uint8_t **password**[64]
mesh softAP password

uint8_t **max_connection**
max number of stations allowed to connect in, max 10

struct mesh_cfg_t
Mesh initialization configuration.

Public Members

uint8_t **channel**
channel, the mesh network on

mesh_event_cb_t **event_cb**
mesh event callback

mesh_addr_t **mesh_id**
mesh network identification

mesh_router_t **router**
router configuration

mesh_ap_cfg_t **mesh_ap**
mesh softAP configuration

const mesh_crypto_funcs_t ***crypto_funcs**
crypto functions

struct mesh_vote_t
Vote.

Public Members

float **percentage**
vote percentage threshold for approval of being a root

bool **is_rc_specified**
if true, rc_addr shall be specified (Unimplemented). if false, attempts value shall be specified to make network start root election.

mesh_rc_config_t **config**
vote address configuration

struct mesh_tx_pending_t
The number of packets pending in the queue waiting to be sent by the mesh stack.

Public Members

int **to_parent**
to parent queue

int **to_parent_p2p**
to parent (P2P) queue

int **to_child**
to child queue

int **to_child_p2p**
to child (P2P) queue

int **mgmt**
management queue

int **broadcast**
broadcast and multicast queue

struct mesh_rx_pending_t
The number of packets available in the queue waiting to be received by applications.

Public Members

int **toDS**
to external DS

int **toSelf**
to self

Macros

MESH_ROOT_LAYER
root layer value

MESH_MTU
max transmit unit(in bytes)

MESH_MPS
max payload size(in bytes)

ESP_ERR_MESH_WIFI_NOT_START
Mesh error code definition.

Wi-Fi isn't started

ESP_ERR_MESH_NOT_INIT
mesh isn't initialized

ESP_ERR_MESH_NOT_CONFIG
mesh isn't configured

ESP_ERR_MESH_NOT_START
mesh isn't started

ESP_ERR_MESH_NOT_SUPPORT
not supported yet

ESP_ERR_MESH_NOT_ALLOWED
operation is not allowed

ESP_ERR_MESH_NO_MEMORY
out of memory

ESP_ERR_MESH_ARGUMENT
illegal argument

ESP_ERR_MESH_EXCEED_MTU
packet size exceeds MTU

ESP_ERR_MESH_TIMEOUT
timeout

ESP_ERR_MESH_DISCONNECTED
disconnected with parent on station interface

ESP_ERR_MESH_QUEUE_FAIL
queue fail

ESP_ERR_MESH_QUEUE_FULL
queue full

ESP_ERR_MESH_NO_PARENT_FOUND
no parent found to join the mesh network

ESP_ERR_MESH_NO_ROUTE_FOUND
no route found to forward the packet

ESP_ERR_MESH_OPTION_NULL
no option found

ESP_ERR_MESH_OPTION_UNKNOWN
unknown option

ESP_ERR_MESH_XON_NO_WINDOW
no window for software flow control on upstream

ESP_ERR_MESH_INTERFACE
low-level Wi-Fi interface error

ESP_ERR_MESH_DISCARD_DUPLICATE
discard the packet due to the duplicate sequence number

ESP_ERR_MESH_DISCARD
discard the packet

ESP_ERR_MESH_VOTING
vote in progress

MESH_DATA_ENC
Flags bitmap for esp_mesh_send() and esp_mesh_recv()

data encrypted (Unimplemented)

MESH_DATA_P2P
point-to-point delivery over the mesh network

MESH_DATA_FROMDS
receive from external IP network

MESH_DATA_TODS
identify this packet is target to external IP network

MESH_DATA_NONBLOCK
esp_mesh_send() non-block

MESH_DATA_DROP
in the situation of the root having been changed, identify this packet can be dropped by new root

MESH_DATA_GROUP
identify this packet is target to a group address

MESH_OPT_SEND_GROUP
Option definitions for esp_mesh_send() and esp_mesh_recv()

data transmission by group; used with esp_mesh_send() and shall have payload

MESH_OPT_RECV_DS_ADDR
return a remote IP address; used with esp_mesh_send() and esp_mesh_recv()

MESH_ASSOC_FLAG_VOTE_IN_PROGRESS
Flag of mesh networking IE.

vote in progress

MESH_ASSOC_FLAG_NETWORK_FREE
no root in current network

MESH_ASSOC_FLAG_ROOTS_FOUND
root conflict is found

MESH_ASSOC_FLAG_ROOT_FIXED
fixed root

MESH_INIT_CONFIG_DEFAULT()

Type Definitions

typedef system_event_sta_got_ip_t **mesh_event_root_got_ip_t**
IP settings from LwIP stack.

typedef *mesh_addr_t* **mesh_event_root_address_t**
Root address.

typedef system_event_sta_disconnected_t **mesh_event_disconnected_t**
Parent disconnected information.

typedef system_event_ap_staconnected_t **mesh_event_child_connected_t**
Child connected information.

typedef system_event_ap_stadisconnected_t **mesh_event_child_disconnected_t**
Child disconnected information.

typedef void (*__mesh_event_cb_t__)(*mesh_event_t* event)
Mesh event callback handler prototype definition.

Parameters

- `event`: *mesh_event_t*

Enumerations

enum mesh_event_id_t
Enumerated list of mesh event id.

Values:

MESH_EVENT_STARTED
mesh is started

MESH_EVENT_STOPPED
mesh is stopped

MESH_EVENT_CHANNEL_SWITCH
channel switch

MESH_EVENT_CHILD_CONNECTED
a child is connected on softAP interface

MESH_EVENT_CHILD_DISCONNECTED
a child is disconnected on softAP interface

MESH_EVENT_ROUTING_TABLE_ADD
routing table is changed by adding newly joined children

MESH_EVENT_ROUTING_TABLE_REMOVE
routing table is changed by removing leave children

MESH_EVENT_PARENT_CONNECTED
parent is connected on station interface

MESH_EVENT_PARENT_DISCONNECTED
parent is disconnected on station interface

MESH_EVENT_NO_PARENT_FOUND
no parent found

MESH_EVENT_LAYER_CHANGE
layer changes over the mesh network

MESH_EVENT_TODS_STATE
state represents whether the root is able to access external IP network

MESH_EVENT_VOTE_STARTED
the process of voting a new root is started either by children or by the root

MESH_EVENT_VOTE_STOPPED
the process of voting a new root is stopped

MESH_EVENT_ROOT_ADDRESS
the root address is obtained. It is posted by mesh stack automatically.

MESH_EVENT_ROOT_SWITCH_REQ
root switch request sent from a new voted root candidate

MESH_EVENT_ROOT_SWITCH_ACK
root switch acknowledgment responds the above request sent from current root

MESH_EVENT_ROOT_GOT_IP
the root obtains the IP address. It is posted by LwIP stack automatically

MESH_EVENT_ROOT_LOST_IP
the root loses the IP address. It is posted by LwIP stack automatically

MESH_EVENT_ROOT_ASKED_YIELD
the root is asked yield by a more powerful existing root. If self organized is disabled and this device is specified to be a root by users, users should set a new parent for this device. if self organized is enabled, this device will find a new parent by itself, users could ignore this event.

MESH_EVENT_ROOT_FIXED
when devices join a network, if the setting of Fixed Root for one device is different from that of its parent, the device will update the setting the same as its parent's. Fixed Root Setting of each device is variable as that setting changes of the root.

MESH_EVENT_SCAN_DONE
if self-organized networking is disabled, user can call esp_wifi_scan_start() to trigger this event, and add the corresponding scan done handler in this event.

MESH_EVENT_MAX

enum mesh_type_t
Device type.

Values:

MESH_IDLE
hasn't joined the mesh network yet

MESH_ROOT
the only sink of the mesh network. Has the ability to access external IP network

MESH_NODE
intermediate device. Has the ability to forward packets over the mesh network

MESH_LEAF
has no forwarding ability

enum mesh_proto_t
Protocol of transmitted application data.

Values:

MESH_PROTO_BIN
binary

MESH_PROTO_HTTP
HTTP protocol

MESH_PROTO_JSON
JSON format

MESH_PROTO_MQTT
MQTT protocol

enum mesh_tos_t
For reliable transmission, mesh stack provides three type of services.

Values:

MESH_TOS_P2P
provide P2P (point-to-point) retransmission on mesh stack by default

MESH_TOS_E2E
provide E2E (end-to-end) retransmission on mesh stack (Unimplemented)

MESH_TOS_DEF
no retransmission on mesh stack

enum mesh_vote_reason_t
Vote reason.

Values:

MESH_VOTE_REASON_ROOT_INITIATED = 1
vote is initiated by the root

MESH_VOTE_REASON_CHILD_INITIATED
vote is initiated by children

enum mesh_disconnect_reason_t

Mesh disconnect reason code.

Values:

MESH_REASON_CYCLIC = 100

cyclic is detected

MESH_REASON_PARENT_IDLE

parent is idle

MESH_REASON_LEAF

the connected device is changed to a leaf

MESH_REASON_DIFF_ID

in different mesh ID

MESH_REASON_ROOTS

root conflict is detected

MESH_REASON_PARENT_STOPPED

parent has stopped the mesh

MESH_REASON_SCAN_FAIL

scan fail

enum mesh_event_toDS_state_t

The reachability of the root to a DS (distribute system)

Values:

MESH_TODS_UNREACHABLE

the root isn't able to access external IP network

MESH_TODS_REACHABLE

the root is able to access external IP network

Example code for this API section is provided in mesh directory of ESP-IDF examples.

3.3 Bluetooth API

3.3.1 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/esp_bt.h

Functions

esp_err_t **esp_ble_tx_power_set** (*esp_ble_power_type_t power_type*, *esp_power_level_t power_level*)

Set BLE TX power Connection Tx power should only be set after connection created.

Return ESP_OK - success, other - failed

Parameters

- `power_type`: : The type of which tx power, could set Advertising/Connection/Default and etc
- `power_level`: Power level(index) corresponding to absolute value(dbm)

esp_power_level_t **esp_ble_tx_power_get** (*esp_ble_power_type_t power_type*)

Get BLE TX power Connection Tx power should only be get after connection created.

Return >= 0 - Power level, < 0 - Invalid

Parameters

- `power_type`: : The type of which tx power, could set Advertising/Connection/Default and etc

esp_err_t **esp_bredr_tx_power_set** (*esp_power_level_t min_power_level*, *esp_power_level_t max_power_level*)

Set BR/EDR TX power BR/EDR power control will use the power in range of minimum value and maximum value. The power level will effect the global BR/EDR TX power, such inquire, page, connection and so on. Please call the function after esp_bt_controller_enable and before any function which cause RF do TX. So you can call the function can before do discover, beofre profile init and so on. For example, if you want BR/EDR use the new TX power to do inquire, you should call this function before inquire. Another word, If call this function when BR/EDR is in inquire(ING), please do inquire again after call this function. Default minimum power level is ESP_PWR_LVL_N0, and maximum power level is ESP_PWR_LVL_P3.

Return ESP_OK - success, other - failed

Parameters

- `min_power_level`: The minimum power level
- `max_power_level`: The maximum power level

esp_err_t **esp_bredr_tx_power_get** (*esp_power_level_t *min_power_level*, *esp_power_level_t *max_power_level*)

Get BR/EDR TX power If the argument is not NULL, then store the corresponding value.

Return ESP_OK - success, other - failed

Parameters

- `min_power_level`: The minimum power level
- `max_power_level`: The maximum power level

esp_err_t **esp_bredr_sco_datapath_set** (*esp_sco_data_path_t data_path*)
set default SCO data path Should be called after controller is enabled, and before (e)SCO link is established

Return ESP_OK - success, other - failed

Parameters

- `data_path`: SCO data path

esp_err_t **esp_bt_controller_init** (*esp_bt_controller_config_t *cfg*)
Initialize BT controller to allocate task and other resource.

Return ESP_OK - success, other - failed

Parameters

- `cfg`: Initial configuration of BT controller. This function should be called only once, before any other BT functions are called.

esp_err_t **esp_bt_controller_deinit** (void)
De-initialize BT controller to free resource and delete task.

This function should be called only once, after any other BT functions are called. This function is not whole completed, esp_bt_controller_init cannot called after this function.

Return ESP_OK - success, other - failed

esp_err_t **esp_bt_controller_enable** (*esp_bt_mode_t mode*)
Enable BT controller. Due to a known issue, you cannot call esp_bt_controller_enable() a second time to change the controller mode dynamically. To change controller mode, call esp_bt_controller_disable() and then call esp_bt_controller_enable() with the new mode.

Return ESP_OK - success, other - failed

Parameters

- `mode`: : the mode(BLE/BT/BTDM) to enable.

esp_err_t **esp_bt_controller_disable** (void)
Disable BT controller.

Return ESP_OK - success, other - failed

esp_bt_controller_status_t **esp_bt_controller_get_status** (void)
Get BT controller is initialised/de-initialised/enabled/disabled.

Return status value

bool **esp_vhci_host_check_send_available** (void)
esp_vhci_host_check_send_available used for check actively if the host can send packet to controller or not.

Return true for ready to send, false means cannot send packet

void **esp_vhci_host_send_packet** (uint8_t **data*, uint16_t *len*)
esp_vhci_host_send_packet host send packet to controller

Should not call this function from within a critical section or when the scheduler is suspended.

Parameters

- `data`: the packet point
- `len`: the packet length

void **esp_vhci_host_register_callback**(**const** *esp_vhci_host_callback_t* **callback*)

esp_vhci_host_register_callback register the vhci referece callback, the call back struct defined by vhci_host_callback structure.

Parameters

- `callback`: *esp_vhci_host_callback* type variable

esp_err_t **esp_bt_controller_mem_release**(*esp_bt_mode_t mode*)

esp_bt_controller_mem_release release the memory by mode, if never use the bluetooth mode it can release the .bss, .data and other section to heap. The total size is about 70k bytes.

esp_bt_controller_mem_release(mode) should be called only before esp_bt_controller_init() or after esp_bt_controller_deinit().

Note that once BT controller memory is released, the process cannot be reversed. It means you can not use the bluetooth mode which you have released by this function.

If your firmware will later upgrade the Bluetooth controller mode (BLE -> BT Classic or disabled -> enabled) then do not call this function.

If the app calls esp_bt_controller_enable(ESP_BT_MODE_BLE) to use BLE only then it is safe to call esp_bt_controller_mem_release(ESP_BT_MODE_CLASSIC_BT) at initialisation time to free unused BT Classic memory.

If user never use bluetooth controller, could call esp_bt_controller_mem_release(ESP_BT_MODE_BTDM) before esp_bt_controller_init or after esp_bt_controller_deinit.

For example, user only use bluetooth to config SSID and PASSWORD of WIFI, after config, will never use bluetooth. Then, could call esp_bt_controller_mem_release(ESP_BT_MODE_BTDM) after esp_bt_controller_deinit.

Return ESP_OK - success, other - failed

Parameters

- `mode`: : the mode want to release memory

esp_err_t **esp_bt_sleep_enable**(void)

enable bluetooth to enter modem sleep

Note that this function shall not be invoked before esp_bt_controller_enable()

There are currently two options for bluetooth modem sleep, one is ORIG mode, and another is EVED Mode. EVED Mode is intended for BLE only.

For ORIG mode: Bluetooth modem sleep is enabled in controller start up by default if CONFIG_BTDM_CONTROLLER_MODEM_SLEEP is set and "ORIG mode" is selected. In ORIG modem sleep mode, bluetooth controller will switch off some components and pause to work every now and then, if there is no event to process; and wakeup according to the scheduled interval and resume the work. It can also wakeup earlier upon external request using function "esp_bt_controller_wakeup_request".

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_bt_sleep_disable**(void)

disable bluetooth modem sleep

Note that this function shall not be invoked before esp_bt_controller_enable()

If esp_bt_sleep_disable() is called, bluetooth controller will not be allowed to enter modem sleep;

If ORIG modem sleep mode is in use, if this function is called, bluetooth controller may not immediately wake up if it is dormant then. In this case, esp_bt_controller_wakeup_request() can be used to shorten the time for wakeup.

Return

- ESP_OK : success
- other : failed

bool **esp_bt_controller_is_sleeping**(void)

to check whether bluetooth controller is sleeping at the instant, if modem sleep is enabled

Note that this function shall not be invoked before esp_bt_controller_enable() This function is supposed to be used ORIG mode of modem sleep

Return true if in modem sleep state, false otherwise

void **esp_bt_controller_wakeup_request**(void)

request controller to wakeup from sleeping state during sleep mode

Note that this function shall not be invoked before esp_bt_controller_enable() Note that this function is supposed to be used ORIG mode of modem sleep Note that after this request, bluetooth controller may again enter sleep as long as the modem sleep is enabled

Profiling shows that it takes several milliseconds to wakeup from modem sleep after this request. Generally it takes longer if 32kHz XTAL is used than the main XTAL, due to the lower frequncy of the former as the bluetooth low power clock source.

Structures

struct esp_bt_controller_config_t

Controller config options, depend on config mask. Config mask indicate which functions enabled, this means some options or parameters of some functions enabled by config mask.

Public Members

uint16_t **controller_task_stack_size**

Bluetooth controller task stack size

uint8_t **controller_task_prio**

Bluetooth controller task priority

uint8_t **hci_uart_no**

If use UART1/2 as HCI IO interface, indicate UART number

uint32_t **hci_uart_baudrate**

If use UART1/2 as HCI IO interface, indicate UART baudrate

uint8_t **scan_duplicate_mode**

If use UART1/2 as HCI IO interface, indicate UART baudrate

uint16_t **normal_adv_size**
Normal adv size for scan duplicate

uint16_t **mesh_adv_size**
Mesh adv size for scan duplicate

uint16_t **send_adv_reserved_size**
Controller minimum memory value

uint32_t **controller_debug_flag**
Controller debug log flag

uint8_t **bt_sco_datapath**
SCO data path, i.e. HCI or PCM module

bool **auto_latency**
BLE auto latency, used to enhance classic BT performance

struct esp_vhci_host_callback
esp_vhci_host_callback used for vhci call host function to notify what host need to do

Public Members

void (*__notify_host_send_available__) (void)
callback used to notify that the host can send packet to controller

int (*__notify_host_recv__) (uint8_t *data, uint16_t len)
callback used to notify that the controller has a packet to send to the host

Macros

BT_CONTROLLER_INIT_CONFIG_DEFAULT ()

Type Definitions

typedef struct *esp_vhci_host_callback* **esp_vhci_host_callback_t**
esp_vhci_host_callback used for vhci call host function to notify what host need to do

Enumerations

enum esp_bt_mode_t
Bluetooth mode for controller enable/disable.

Values:

ESP_BT_MODE_IDLE = 0x00
Bluetooth is not running

ESP_BT_MODE_BLE = 0x01
Run BLE mode

ESP_BT_MODE_CLASSIC_BT = 0x02
Run Classic BT mode

ESP_BT_MODE_BTDM = 0x03
Run dual mode

enum esp_bt_controller_status_t

Bluetooth controller enable/disable/initialised/de-initialised status.

Values:

ESP_BT_CONTROLLER_STATUS_IDLE = 0

ESP_BT_CONTROLLER_STATUS_INITED

ESP_BT_CONTROLLER_STATUS_ENABLED

ESP_BT_CONTROLLER_STATUS_NUM

enum esp_ble_power_type_t

BLE tx power type ESP_BLE_PWR_TYPE_CONN_HDL0-8: for each connection, and only be set after connection completed. when disconnect, the correspond TX power is not effected. ESP_BLE_PWR_TYPE_ADV : for advertising/scan response. ESP_BLE_PWR_TYPE_SCAN : for scan. ESP_BLE_PWR_TYPE_DEFAULT : if each connection's TX power is not set, it will use this default value. if neither in scan mode nor in adv mode, it will use this default value. If none of power type is set, system will use ESP_PWR_LVL_P3 as default for ADV/SCAN/CONN0-9.

Values:

ESP_BLE_PWR_TYPE_CONN_HDL0 = 0
For connection handle 0

ESP_BLE_PWR_TYPE_CONN_HDL1 = 1
For connection handle 1

ESP_BLE_PWR_TYPE_CONN_HDL2 = 2
For connection handle 2

ESP_BLE_PWR_TYPE_CONN_HDL3 = 3
For connection handle 3

ESP_BLE_PWR_TYPE_CONN_HDL4 = 4
For connection handle 4

ESP_BLE_PWR_TYPE_CONN_HDL5 = 5
For connection handle 5

ESP_BLE_PWR_TYPE_CONN_HDL6 = 6
For connection handle 6

ESP_BLE_PWR_TYPE_CONN_HDL7 = 7
For connection handle 7

ESP_BLE_PWR_TYPE_CONN_HDL8 = 8
For connection handle 8

ESP_BLE_PWR_TYPE_ADV = 9
For advertising

ESP_BLE_PWR_TYPE_SCAN = 10
For scan

ESP_BLE_PWR_TYPE_DEFAULT = 11
For default, if not set other, it will use default value

ESP_BLE_PWR_TYPE_NUM = 12
TYPE numbers

enum esp_power_level_t

Bluetooth TX power level(index), it's just a index corresponding to power(dbm).

Values:

ESP_PWR_LVL_N12 = 0
Corresponding to -12dbm

ESP_PWR_LVL_N9 = 1
Corresponding to -9dbm

ESP_PWR_LVL_N6 = 2
Corresponding to -6dbm

ESP_PWR_LVL_N3 = 3
Corresponding to -3dbm

ESP_PWR_LVL_N0 = 4
Corresponding to 0dbm

ESP_PWR_LVL_P3 = 5
Corresponding to +3dbm

ESP_PWR_LVL_P6 = 6
Corresponding to +6dbm

ESP_PWR_LVL_P9 = 7
Corresponding to +9dbm

ESP_PWR_LVL_N14 = ESP_PWR_LVL_N12
Backward compatibility! Setting to -14dbm will actually result to -12dbm

ESP_PWR_LVL_N11 = ESP_PWR_LVL_N9
Backward compatibility! Setting to -11dbm will actually result to -9dbm

ESP_PWR_LVL_N8 = ESP_PWR_LVL_N6
Backward compatibility! Setting to -8dbm will actually result to -6dbm

ESP_PWR_LVL_N5 = ESP_PWR_LVL_N3
Backward compatibility! Setting to -5dbm will actually result to -3dbm

ESP_PWR_LVL_N2 = ESP_PWR_LVL_N0
Backward compatibility! Setting to -2dbm will actually result to 0dbm

ESP_PWR_LVL_P1 = ESP_PWR_LVL_P3
Backward compatibility! Setting to +1dbm will actually result to +3dbm

ESP_PWR_LVL_P4 = ESP_PWR_LVL_P6
Backward compatibility! Setting to +4dbm will actually result to +6dbm

ESP_PWR_LVL_P7 = ESP_PWR_LVL_P9
Backward compatibility! Setting to +7dbm will actually result to +9dbm

enum esp_sco_data_path_t
Bluetooth audio data transport path.

Values:

ESP_SCO_DATA_PATH_HCI = 0
data over HCI transport

ESP_SCO_DATA_PATH_PCM = 1
data over PCM interface

3.3.2 BT COMMON

BT GENERIC DEFINES

Overview

Instructions

Application Example

Instructions

API Reference

Header File

- bt/bluedroid/api/include/api/esp_bt_defs.h

Structures

struct esp_bt_uuid_t
UUID type.

Public Members

uint16_t **len**
UUID length, 16bit, 32bit or 128bit

union *esp_bt_uuid_t*::**[anonymous] uuid**
UUID

Macros

ESP_BLUEDROID_STATUS_CHECK(status)

ESP_BT_OCTET16_LEN

ESP_BT_OCTET8_LEN

ESP_DEFAULT_GATT_IF
Default GATT interface id.

ESP_BLE_CONN_INT_MIN
relate to BTM_BLE_CONN_INT_MIN in stack/btm_ble_api.h

ESP_BLE_CONN_INT_MAX
relate to BTM_BLE_CONN_INT_MAX in stack/btm_ble_api.h

ESP_BLE_CONN_LATENCY_MAX
relate to ESP_BLE_CONN_LATENCY_MAX in stack/btm_ble_api.h

ESP_BLE_CONN_SUP_TOUT_MIN
relate to BTM_BLE_CONN_SUP_TOUT_MIN in stack/btm_ble_api.h

ESP_BLE_CONN_SUP_TOUT_MAX
relate to ESP_BLE_CONN_SUP_TOUT_MAX in stack/btm_ble_api.h

ESP_BLE_CONN_PARAM_UNDEF

ESP_BLE_SCAN_PARAM_UNDEF

ESP_BLE_IS_VALID_PARAM(x, min, max)
Check the param is valid or not.

ESP_UUID_LEN_16

ESP_UUID_LEN_32

ESP_UUID_LEN_128

ESP_BD_ADDR_LEN
Bluetooth address length.

ESP_BLE_ENC_KEY_MASK
Used to exchange the encrytyption key in the init key & response key.

ESP_BLE_ID_KEY_MASK
Used to exchange the IRK key in the init key & response key.

ESP_BLE_CSR_KEY_MASK
Used to exchange the CSRK key in the init key & response key.

ESP_BLE_LINK_KEY_MASK
Used to exchange the link key(this key just used in the BLE & BR/EDR coexist mode) in the init key & response key.

ESP_APP_ID_MIN
Minimum of the application id.

ESP_APP_ID_MAX
Maximum of the application id.

ESP_BD_ADDR_STR

ESP_BD_ADDR_HEX(addr)

Type Definitions

typedef uint8_t **esp_bt_octet16_t**[**ESP_BT_OCTET16_LEN**]

typedef uint8_t **esp_bt_octet8_t**[**ESP_BT_OCTET8_LEN**]

typedef uint8_t **esp_link_key**[**ESP_BT_OCTET16_LEN**]

typedef uint8_t **esp_bd_addr_t**[**ESP_BD_ADDR_LEN**]
Bluetooth device address.

typedef uint8_t **esp_ble_key_mask_t**

Enumerations

enum esp_bt_status_t
Status Return Value.

Values:

ESP_BT_STATUS_SUCCESS = 0

ESP_BT_STATUS_FAIL

ESP_BT_STATUS_NOT_READY

ESP_BT_STATUS_NOMEM

ESP_BT_STATUS_BUSY

ESP_BT_STATUS_DONE = 5

ESP_BT_STATUS_UNSUPPORTED

ESP_BT_STATUS_PARM_INVALID

ESP_BT_STATUS_UNHANDLED

ESP_BT_STATUS_AUTH_FAILURE

ESP_BT_STATUS_RMT_DEV_DOWN = 10

ESP_BT_STATUS_AUTH_REJECTED

ESP_BT_STATUS_INVALID_STATIC_RAND_ADDR

ESP_BT_STATUS_PENDING

ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL

ESP_BT_STATUS_PARAM_OUT_OF_RANGE

ESP_BT_STATUS_TIMEOUT

ESP_BT_STATUS_PEER_LE_DATA_LEN_UNSUPPORTED

ESP_BT_STATUS_CONTROL_LE_DATA_LEN_UNSUPPORTED

ESP_BT_STATUS_ERR_ILLEGAL_PARAMETER_FMT

ESP_BT_STATUS_MEMORY_FULL

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/api/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/api/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. A BR/EDR/LE device type shall have a single Bluetooth device name which shall be identical irrespective of the physical channel used to perform the name discovery procedure.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_ARG : if name is NULL pointer or empty, or string length out of limit
- ESP_ERR_INVALID_STATE : if bluetooth stack is not yet enabled
- ESP_FAIL : others

Parameters

- `name`: : device name to be set

3.3.3 BT LE

GAP API

Overview

Instructions

Application Example

Check bluetooth folder in ESP-IDF examples, which contains the following demos and their tutorials:

- This is a SMP security client demo and its tutorial. This demo initiates its security parameters and acts as a GATT client, which can send a security request to the peer device and then complete the encryption procedure.
  - bluetooth/gatt_security_client
  - GATT Security Client Example Walkthrough
- This is a SMP security server demo and its tutorial. This demo initiates its security parameters and acts as a GATT server, which can send a pair request to the peer device and then complete the encryption procedure.
  - bluetooth/gatt_security_server
  - GATT Security Server Example Walkthrough

API Reference

Header File

- bt/bluedroid/api/include/api/esp_gap_ble_api.h

Functions

esp_err_t **esp_ble_gap_register_callback**(*esp_gap_ble_cb_t callback*)

This function is called to occur gap event, such as scan result.

Return

- ESP_OK : success
- other : failed

Parameters

- `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 sets 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_config_local_icon** (uint16_t *icon*)
set local gap appearance icon

Return

- ESP_OK : success
- other : failed

Parameters

- `icon`: - External appearance value, these values are defined by the Bluetooth SIG, please refer to https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.gap.appearance.xml

esp_err_t **esp_ble_gap_update_whitelist** (bool *add_remove*, *esp_bd_addr_t* *remote_bda*)
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.

esp_err_t **esp_ble_gap_get_local_used_addr**(*esp_bd_addr_t local_used_addr*, uint8_t **addr_type*)
This function is called to get local used address and adress type. uint8_t *esp_bt_dev_get_address(void) get the public address.

Return - ESP_OK : success

- other : failed

Parameters

- `local_used_addr`: - current local used ble address (six bytes)
- `addr_type`: - ble address type

uint8_t ***esp_ble_resolve_adv_data**(uint8_t **adv_data*, uint8_t *type*, uint8_t **length*)
This function is called to get ADV data for a specific type.

Return pointer of ADV data

Parameters

- `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 lagecy 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 lagecy 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 - >= 0 : bonded devices number.

- ESP_FAIL : failed

esp_err_t **esp_ble_get_bond_device_list** (int **dev_num*, *esp_ble_bond_dev_t* **dev_list*)

Get the device from the security database list of peer device. It will return the device bonded information immediately.

Return - ESP_OK : success

- other : failed

Parameters

- `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 registed. 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_pcsrk_keys_t **pcsrk_key**

received peer device SRK

esp_ble_pid_keys_t **pid_key**

peer device ID key

esp_ble_lenc_keys_t **lenc_key**

local encryption reproduction keys LTK = = d1(ER,DIV,0)

esp_ble_lcsrk_keys **lcsrk_key**

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

uint32_t **num_dis**
The number of discard packets

struct ble_scan_rsp_data_cmpl_evt_param

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate the set scan response data operation success status

struct ble_scan_rsp_data_raw_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate the set raw advertising data operation success status

struct ble_scan_start_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_START_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate scan start operation success status

struct ble_scan_stop_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate scan stop operation success status

struct ble_set_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 date 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_pcsrk_keys_t
BLE CSRK keys.

Public Members

uint32_t **counter**
The counter

esp_bt_octet16_t **csrk**
The csrk key

uint8_t **sec_level**
The security level

struct esp_ble_pid_keys_t
BLE pid keys.

Public Members

esp_bt_octet16_t **irk**
The irk value

esp_ble_addr_type_t **addr_type**
The address type

esp_bd_addr_t **static_addr**
The static address

struct esp_ble_lenc_keys_t
BLE Encryption reproduction keys.

Public Members

esp_bt_octet16_t **ltk**
The long term key

uint16_t **div**
The div value

uint8_t **key_size**
The key size of the security link

uint8_t **sec_level**
The security level of the security link

struct esp_ble_lcsrk_keys
BLE SRK keys.

Public Members

uint32_t **counter**
The counter value

uint16_t **div**
The div value

uint8_t **sec_level**
The security level of the security link

esp_bt_octet16_t **csrk**
The csrk key value

struct esp_ble_sec_key_notif_t
Structure associated with ESP_KEY_NOTIF_EVT.

Public Members

esp_bd_addr_t **bd_addr**
peer address

uint32_t **passkey**
the numeric value for comparison. If just_works, do not show this number to UI

struct esp_ble_sec_req_t
Structure of the security request.

Public Members

esp_bd_addr_t **bd_addr**
peer address

struct esp_ble_bond_key_info_t
struct type of the bond key 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_pcsrk_keys_t **pcsrk_key**
received peer device SRK

esp_ble_pid_keys_t **pid_key**
peer device ID key

struct esp_ble_bond_dev_t
struct type of the bond device value

Public Members

esp_bd_addr_t **bd_addr**
peer address

esp_ble_bond_key_info_t **bond_key**
the bond key information

struct esp_ble_key_t
union type of the security key value

Public Members

esp_bd_addr_t **bd_addr**
peer address

esp_ble_key_type_t **key_type**
key type of the security link

esp_ble_key_value_t **p_key_value**
the pointer to the key value

struct esp_ble_local_id_keys_t
structure type of the ble local id keys value

Public Members

esp_bt_octet16_t **ir**
the 16 bits of the ir value

esp_bt_octet16_t **irk**
the 16 bits of the ir key value

esp_bt_octet16_t **dhk**
the 16 bits of the dh key value

struct esp_ble_auth_cmpl_t
Structure associated with ESP_AUTH_CMPL_EVT.

Public Members

esp_bd_addr_t **bd_addr**
BD address peer device.

bool **key_present**
Valid link key value in key element

esp_link_key **key**
Link key associated with peer device.

uint8_t **key_type**
The type of Link Key

bool **success**
TRUE of authentication succeeded, FALSE if failed.

uint8_t **fail_reason**
The HCI reason/error code for when success=FALSE

esp_ble_addr_type_t **addr_type**
Peer device address type

esp_bt_dev_type_t **dev_type**
Device type

esp_ble_auth_req_t **auth_mode**
authentication mode

Macros

ESP_BLE_ADV_FLAG_LIMIT_DISC
BLE_ADV_DATA_FLAG data flag bit definition used for advertising data flag

ESP_BLE_ADV_FLAG_GEN_DISC

ESP_BLE_ADV_FLAG_BREDR_NOT_SPT

ESP_BLE_ADV_FLAG_DMT_CONTROLLER_SPT

ESP_BLE_ADV_FLAG_DMT_HOST_SPT

ESP_BLE_ADV_FLAG_NON_LIMIT_DISC

ESP_LE_KEY_NONE

ESP_LE_KEY_PENC

ESP_LE_KEY_PID

ESP_LE_KEY_PCSRK

ESP_LE_KEY_PLK

ESP_LE_KEY_LLK

ESP_LE_KEY_LENC

ESP_LE_KEY_LID

ESP_LE_KEY_LCSRK

ESP_LE_AUTH_NO_BOND

ESP_LE_AUTH_BOND

ESP_LE_AUTH_REQ_MITM

ESP_LE_AUTH_REQ_BOND_MITM
0101

ESP_LE_AUTH_REQ_SC_ONLY

ESP_LE_AUTH_REQ_SC_BOND

ESP_LE_AUTH_REQ_SC_MITM

ESP_LE_AUTH_REQ_SC_MITM_BOND

ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_DISABLE

ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_ENABLE

ESP_IO_CAP_OUT

ESP_IO_CAP_IO

ESP_IO_CAP_IN

ESP_IO_CAP_NONE

ESP_IO_CAP_KBDISP

ESP_BLE_APPEARANCE_UNKNOWN

ESP_BLE_APPEARANCE_GENERIC_PHONE

ESP_BLE_APPEARANCE_GENERIC_COMPUTER

ESP_BLE_APPEARANCE_GENERIC_WATCH

ESP_BLE_APPEARANCE_SPORTS_WATCH

ESP_BLE_APPEARANCE_GENERIC_CLOCK

ESP_BLE_APPEARANCE_GENERIC_DISPLAY

ESP_BLE_APPEARANCE_GENERIC_REMOTE

ESP_BLE_APPEARANCE_GENERIC_EYEGLASSES

ESP_BLE_APPEARANCE_GENERIC_TAG

ESP_BLE_APPEARANCE_GENERIC_KEYRING

ESP_BLE_APPEARANCE_GENERIC_MEDIA_PLAYER

ESP_BLE_APPEARANCE_GENERIC_BARCODE_SCANNER

ESP_BLE_APPEARANCE_GENERIC_THERMOMETER

ESP_BLE_APPEARANCE_THERMOMETER_EAR

ESP_BLE_APPEARANCE_GENERIC_HEART_RATE

ESP_BLE_APPEARANCE_HEART_RATE_BELT

ESP_BLE_APPEARANCE_GENERIC_BLOOD_PRESSURE

ESP_BLE_APPEARANCE_BLOOD_PRESSURE_ARM

ESP_BLE_APPEARANCE_BLOOD_PRESSURE_WRIST

ESP_BLE_APPEARANCE_GENERIC_HID

ESP_BLE_APPEARANCE_HID_KEYBOARD

ESP_BLE_APPEARANCE_HID_MOUSE

ESP_BLE_APPEARANCE_HID_JOYSTICK

ESP_BLE_APPEARANCE_HID_GAMEPAD

ESP_BLE_APPEARANCE_HID_DIGITIZER_TABLET

ESP_BLE_APPEARANCE_HID_CARD_READER

ESP_BLE_APPEARANCE_HID_DIGITAL_PEN

ESP_BLE_APPEARANCE_HID_BARCODE_SCANNER

ESP_BLE_APPEARANCE_GENERIC_GLUCOSE

ESP_BLE_APPEARANCE_GENERIC_WALKING

ESP_BLE_APPEARANCE_WALKING_IN_SHOE

ESP_BLE_APPEARANCE_WALKING_ON_SHOE

ESP_BLE_APPEARANCE_WALKING_ON_HIP

ESP_BLE_APPEARANCE_GENERIC_CYCLING

ESP_BLE_APPEARANCE_CYCLING_COMPUTER

ESP_BLE_APPEARANCE_CYCLING_SPEED

ESP_BLE_APPEARANCE_CYCLING_CADENCE

ESP_BLE_APPEARANCE_CYCLING_POWER

ESP_BLE_APPEARANCE_CYCLING_SPEED_CADENCE

ESP_BLE_APPEARANCE_GENERIC_PULSE_OXIMETER

ESP_BLE_APPEARANCE_PULSE_OXIMETER_FINGERTIP

ESP_BLE_APPEARANCE_PULSE_OXIMETER_WRIST

ESP_BLE_APPEARANCE_GENERIC_WEIGHT

ESP_BLE_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVICE

ESP_BLE_APPEARANCE_POWERED_WHEELCHAIR

ESP_BLE_APPEARANCE_MOBILITY_SCOOTER

ESP_BLE_APPEARANCE_GENERIC_CONTINUOUS_GLUCOSE_MONITOR

ESP_BLE_APPEARANCE_GENERIC_INSULIN_PUMP

ESP_BLE_APPEARANCE_INSULIN_PUMP_DURABLE_PUMP

ESP_BLE_APPEARANCE_INSULIN_PUMP_PATCH_PUMP

ESP_BLE_APPEARANCE_INSULIN_PEN

ESP_BLE_APPEARANCE_GENERIC_MEDICATION_DELIVERY

ESP_BLE_APPEARANCE_GENERIC_OUTDOOR_SPORTS

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_AND_NAV

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD_AND_NAV

ESP_GAP_BLE_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 lenght 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

ESP_BLE_SM_MIN_KEY_SIZE

ESP_BLE_SM_SET_STATIC_PASSKEY

ESP_BLE_SM_CLEAR_STATIC_PASSKEY

ESP_BLE_SM_ONLY_ACCEPT_SPECIFIED_SEC_AUTH

ESP_BLE_SM_MAX_PARAM

enum esp_ble_scan_type_t

Ble scan type.

Values:

BLE_SCAN_TYPE_PASSIVE = 0x0

Passive scan

BLE_SCAN_TYPE_ACTIVE = 0x1

Active scan

enum esp_ble_scan_filter_t

Ble scan filter type.

Values:

BLE_SCAN_FILTER_ALLOW_ALL = 0x0

Accept all :

1. advertisement packets except directed advertising packets not addressed to this device (default).

BLE_SCAN_FILTER_ALLOW_ONLY_WLST = 0x1

Accept only :

1. advertisement packets from devices where the advertiser's address is in the White list.
2. Directed advertising packets which are not addressed for this device shall be ignored.

BLE_SCAN_FILTER_ALLOW_UND_RPA_DIR = 0x2

Accept all :

1. undirected advertisement packets, and
2. directed advertising packets where the initiator address is a resolvable private address, and
3. directed advertising packets addressed to this device.

BLE_SCAN_FILTER_ALLOW_WLIST_PRA_DIR = 0x3
Accept all :

1. advertisement packets from devices where the advertiser's address is in the White list, and
2. directed advertising packets where the initiator address is a resolvable private address, and
3. directed advertising packets addressed to this device.

enum esp_ble_scan_duplicate_t
Ble scan duplicate type.

Values:

BLE_SCAN_DUPLICATE_DISABLE = 0x0
the Link Layer should generate advertising reports to the host for each packet received

BLE_SCAN_DUPLICATE_ENABLE = 0x1
the Link Layer should filter out duplicate advertising reports to the Host

BLE_SCAN_DUPLICATE_MAX = 0x2
0x02 – 0xFF, Reserved for future use

enum esp_gap_search_evt_t
Sub Event of ESP_GAP_BLE_SCAN_RESULT_EVT.

Values:

ESP_GAP_SEARCH_INQ_RES_EVT = 0
Inquiry result for a peer device.

ESP_GAP_SEARCH_INQ_CMPL_EVT = 1
Inquiry complete.

ESP_GAP_SEARCH_DISC_RES_EVT = 2
Discovery result for a peer device.

ESP_GAP_SEARCH_DISC_BLE_RES_EVT = 3
Discovery result for BLE GATT based service on a peer device.

ESP_GAP_SEARCH_DISC_CMPL_EVT = 4
Discovery complete.

ESP_GAP_SEARCH_DI_DISC_CMPL_EVT = 5
Discovery complete.

ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT = 6
Search cancelled

ESP_GAP_SEARCH_INQ_DISCARD_NUM_EVT = 7
The number of pkt discarded by flow control

enum esp_ble_evt_type_t
Ble scan result event type, to indicate the result is scan response or advertising data or other.

Values:

ESP_BLE_EVT_CONN_ADV = 0x00
Connectable undirected advertising (ADV_IND)

ESP_BLE_EVT_CONN_DIR_ADV = 0x01
Connectable directed advertising (ADV_DIRECT_IND)

ESP_BLE_EVT_DISC_ADV = 0x02
Scannable undirected advertising (ADV_SCAN_IND)

ESP_BLE_EVT_NON_CONN_ADV = 0x03
Non connectable undirected advertising (ADV_NONCONN_IND)

ESP_BLE_EVT_SCAN_RSP = 0x04
Scan Response (SCAN_RSP)

enum esp_ble_wl_opration_t
Values:

ESP_BLE_WHITELIST_REMOVE = 0X00
remove mac from whitelist

ESP_BLE_WHITELIST_ADD = 0X01
add address to whitelist

GATT DEFINES

Overview

Instructions

Application Example

Instructions

API Reference

Header File

- bt/bluedroid/api/include/api/esp_gatt_defs.h

Unions

union esp_gatt_rsp_t
#include <esp_gatt_defs.h> GATT remote read request response type.

Public Members

esp_gatt_value_t **attr_value**
Gatt attribute structure

uint16_t **handle**
Gatt attribute handle

Structures

struct esp_gatt_id_t
Gatt id, include uuid and instance id.

Public Members

esp_bt_uuid_t **uuid**
UUID

uint8_t **inst_id**
Instance id

struct esp_gatt_srvc_id_t
Gatt service id, include id (uuid and instance id) and primary flag.

Public Members

esp_gatt_id_t **id**
Gatt id, include uuid and instance

bool **is_primary**
This service is primary or not

struct esp_attr_desc_t
Attribute description (used to create database)

Public Members

uint16_t **uuid_length**
UUID length

uint8_t ***uuid_p**
UUID value

uint16_t **perm**
Attribute permission

uint16_t **max_length**
Maximum length of the element

uint16_t **length**
Current length of the element

uint8_t ***value**
Element value array

struct esp_attr_control_t
attribute auto response flag

Public Members

uint8_t **auto_rsp**
if auto_rsp set to ESP_GATT_RSP_BY_APP, means the response of Write/Read operation will by 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, true if the service is primary service, else is secondary service

uint16_t **start_handle**
The start handle of the service

uint16_t **end_handle**
The end handle of the service

esp_bt_uuid_t **uuid**
The uuid of the service

struct esp_gattc_char_elem_t
characteristic element

Public Members

uint16_t **char_handle**
The characteristic handle

esp_gatt_char_prop_t **properties**
The characteristic properties

esp_bt_uuid_t **uuid**
The characteristic uuid

struct esp_gattc_descr_elem_t
descriptor element

Public Members

uint16_t **handle**
The characteristic descriptor handle

esp_bt_uuid_t **uuid**
The characteristic descriptor uuid

struct esp_gattc_incl_svc_elem_t
include service element

Public Members

uint16_t **handle**
The include service current attribute handle

uint16_t **incl_srvc_s_handle**
The start handle of the service which has been included

uint16_t **incl_srvc_e_handle**
The end handle of the service which has been included

esp_bt_uuid_t **uuid**
The include service uuid

Macros

ESP_GATT_UUID_IMMEDIATE_ALERT_SVC
All "ESP_GATT_UUID_xxx" is attribute types

ESP_GATT_UUID_LINK_LOSS_SVC

ESP_GATT_UUID_TX_POWER_SVC

ESP_GATT_UUID_CURRENT_TIME_SVC

ESP_GATT_UUID_REF_TIME_UPDATE_SVC

ESP_GATT_UUID_NEXT_DST_CHANGE_SVC

ESP_GATT_UUID_GLUCOSE_SVC

ESP_GATT_UUID_HEALTH_THERMOM_SVC

ESP_GATT_UUID_DEVICE_INFO_SVC

ESP_GATT_UUID_HEART_RATE_SVC

ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC

ESP_GATT_UUID_BATTERY_SERVICE_SVC

ESP_GATT_UUID_BLOOD_PRESSURE_SVC

ESP_GATT_UUID_ALERT_NTF_SVC

ESP_GATT_UUID_HID_SVC

ESP_GATT_UUID_SCAN_PARAMETERS_SVC

ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC

ESP_GATT_UUID_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_NUM_DIGITALS_DESCR

ESP_GATT_UUID_VALUE_TRIGGER_DESCR

ESP_GATT_UUID_ENV_SENSING_CONFIG_DESCR

ESP_GATT_UUID_ENV_SENSING_MEASUREMENT_DESCR

ESP_GATT_UUID_ENV_SENSING_TRIGGER_DESCR

ESP_GATT_UUID_TIME_TRIGGER_DESCR

ESP_GATT_UUID_GAP_DEVICE_NAME

ESP_GATT_UUID_GAP_ICON

ESP_GATT_UUID_GAP_PREF_CONN_PARAM

ESP_GATT_UUID_GAP_CENTRAL_ADDR_RESOL

ESP_GATT_UUID_GATT_SRV_CHGD

ESP_GATT_UUID_ALERT_LEVEL

ESP_GATT_UUID_TX_POWER_LEVEL

ESP_GATT_UUID_CURRENT_TIME

ESP_GATT_UUID_LOCAL_TIME_INFO

ESP_GATT_UUID_REF_TIME_INFO

ESP_GATT_UUID_NW_STATUS

ESP_GATT_UUID_NW_TRIGGER

ESP_GATT_UUID_ALERT_STATUS

ESP_GATT_UUID_RINGER_CP

ESP_GATT_UUID_RINGER_SETTING

ESP_GATT_UUID_GM_MEASUREMENT

ESP_GATT_UUID_GM_CONTEXT

ESP_GATT_UUID_GM_CONTROL_POINT

ESP_GATT_UUID_GM_FEATURE

ESP_GATT_UUID_SYSTEM_ID

ESP_GATT_UUID_MODEL_NUMBER_STR

ESP_GATT_UUID_SERIAL_NUMBER_STR

ESP_GATT_UUID_FW_VERSION_STR

ESP_GATT_UUID_HW_VERSION_STR

ESP_GATT_UUID_SW_VERSION_STR

ESP_GATT_UUID_MANU_NAME

ESP_GATT_UUID_IEEE_DATA

ESP_GATT_UUID_PNP_ID

ESP_GATT_UUID_HID_INFORMATION

ESP_GATT_UUID_HID_REPORT_MAP

ESP_GATT_UUID_HID_CONTROL_POINT

ESP_GATT_UUID_HID_REPORT

ESP_GATT_UUID_HID_PROTO_MODE

ESP_GATT_UUID_HID_BT_KB_INPUT

ESP_GATT_UUID_HID_BT_KB_OUTPUT

ESP_GATT_UUID_HID_BT_MOUSE_INPUT

ESP_GATT_HEART_RATE_MEAS
Heart Rate Measurement.

ESP_GATT_BODY_SENSOR_LOCATION
Body Sensor Location.

ESP_GATT_HEART_RATE_CNTL_POINT
Heart Rate Control Point.

ESP_GATT_UUID_BATTERY_LEVEL

ESP_GATT_UUID_SC_CONTROL_POINT

ESP_GATT_UUID_SENSOR_LOCATION

ESP_GATT_UUID_RSC_MEASUREMENT

ESP_GATT_UUID_RSC_FEATURE

ESP_GATT_UUID_CSC_MEASUREMENT

ESP_GATT_UUID_CSC_FEATURE

ESP_GATT_UUID_SCAN_INT_WINDOW

ESP_GATT_UUID_SCAN_REFRESH

ESP_GATT_ILLEGAL_UUID
GATT INVALID UUID.

ESP_GATT_ILLEGAL_HANDLE
GATT INVALID HANDLE.

ESP_GATT_ATTR_HANDLE_MAX
GATT attribute max handle.

ESP_GATT_MAX_READ_MULTI_HANDLES

ESP_GATT_PERM_READ
Attribute permissions.

ESP_GATT_PERM_READ_ENCRYPTED

ESP_GATT_PERM_READ_ENC_MITM

ESP_GATT_PERM_WRITE

ESP_GATT_PERM_WRITE_ENCRYPTED

ESP_GATT_PERM_WRITE_ENC_MITM

ESP_GATT_PERM_WRITE_SIGNED

ESP_GATT_PERM_WRITE_SIGNED_MITM

ESP_GATT_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
Connectionterminated 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 demos and their tutorials:

- This is a GATT sever demo and its tutorial. This demo creates a GATT service with an attribute table, which releases the user from adding attributes one by one. This is the recommended method of adding attributes.
  - bluetooth/gatt_server_service_table
  - GATT Server Service Table Example Walkthrough
- This is a GATT server demo and its tutorial. This demo creates a GATT service by adding attributes one by one as defined by Bluedroid. The recommended method of adding attributes is presented in example above.
  - bluetooth/gatt_server
  - GATT Server Example Walkthrough
- This is a BLE SPP-Like demo. This demo, which acts as a GATT server, can receive data from UART and then send the data to the peer device automatically.
  - bluetooth/ble_spp_server

API Reference

Header File

- bt/bluedroid/api/include/api/esp_gatts_api.h

Functions

esp_err_t **esp_ble_gatts_register_callback**(*esp_gatts_cb_t callback*)
This function is called to register application callbacks with BTA GATTS module.

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_app_register**(uint16_t *app_id*)
This function is called to register application identifier.

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_app_unregister**(*esp_gatt_if_t gatts_if*)

unregister with GATT Server.

Return

- ESP_OK : success
- other : failed

Parameters

- `gatts_if`: GATT server access interface

esp_err_t **esp_ble_gatts_create_service**(*esp_gatt_if_t gatts_if*, *esp_gatt_srvc_id_t *service_id*, uint16_t *num_handle*)

Create a service. When service creation is done, a callback event 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. This function have to be called between

'esp_ble_gatts_create_service' and 'esp_ble_gatts_add_char'. After included service is included, a callback event 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_srvc_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 demos and their tutorials:

- This is a GATT client demo and its tutorial. This demo can scan for devices, connect to the GATT server and discover its services.
  - bluetooth/gatt_client
  - GATT Client Example Walkthrough
- This is a multiple connection demo and its tutorial. This demo can connect to multiple GATT server devices and discover their services.
  - bluetooth/gattc_multi_connect
  - GATT Client Multi-connection Example Walkthrough
- This is a BLE SPP-Like demo. This demo, which acts as a GATT client, can receive data from UART and then send the data to the peer device automatically.
  - bluetooth/ble_spp_client

API Reference

Header File

- bt/bluedroid/api/include/api/esp_gattc_api.h

Functions

esp_err_t **esp_ble_gattc_register_callback**(*esp_gattc_cb_t callback*)

This function is called to register application callbacks with GATTC module.

Return

- ESP_OK: success
- other: failed

Parameters

- `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 GATTC 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 registed, 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 whith 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 charateristic 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 *result*, uint16_t **count*)

Find the descriptor with the given characteristic handle in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

Return

- ESP_OK: success
- other: failed

Parameters

- `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 hadle 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`: : charateristic 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 hanlde 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.

esp_err_t **esp_ble_gattc_cache_assoc**(*esp_gatt_if_t gattc_if*, *esp_bd_addr_t src_addr*, *esp_bd_addr_t assoc_addr*, bool *is_assoc*)

Add or delete the associated address with the source address. Note: The role of this API is mainly when the client side has stored a server-side database, when it needs to connect another device, but the device's attribute database is the same as the server database stored on the client-side, calling this API can use the database that the device has stored used as the peer server database to reduce the attribute database search and discovery process and speed up the connection time. The associated address mains that device want to used the database has stored in the local cache. The source address mains that device want to share the database to the associated address device.

Return

- ESP_OK: success
- other: failed

Parameters

- `gattc_if`: Gatt client access interface.
- `src_addr`: the source address which provide the attribute table.
- `assoc_addr`: the associated device address which went to share the attribute table with the source address.
- `is_assoc`: true add the associated device address, false remove the associated device address.

esp_err_t **esp_ble_gattc_cache_get_addr_list**(*esp_gatt_if_t gattc_if*)

Get the address list which has store the attribute table in the gattc cache. There will callback ESP_GATTC_GET_ADDR_LIST_EVT event when get address list complete.

Return

- ESP_OK: success
- other: failed

Parameters

- `gattc_if`: Gatt client access interface.

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_set_assoc_addr_cmp_evt_param* **set_assoc_cmp**
Gatt client callback param of ESP_GATTC_SET_ASSOC_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_get_addr_list_evt_param* **get_addr_list**
Gatt client callback param of ESP_GATTC_GET_ADDR_LIST_EVT

struct *esp_ble_gattc_cb_param_t*::*gattc_queue_full_evt_param* **queue_full**
Gatt client callback param of ESP_GATTC_QUEUE_FULL_EVT

struct 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_get_addr_list_evt_param
#include <*esp_gattc_api.h*> ESP_GATTC_GET_ADDR_LIST_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint8_t **num_addr**
The number of address in the gattc cache address list

esp_bd_addr_t ***addr_list**
The pointer to the address list which has been get from the gattc cache

struct gattc_notify_evt_param
#include <*esp_gattc_api.h*> ESP_GATTC_NOTIFY_EVT.

Public Members

uint16_t **conn_id**
Connection id

esp_bd_addr_t **remote_bda**
Remote bluetooth device address

uint16_t **handle**
The Characteristic or descriptor handle

uint16_t **value_len**
Notify attribute value

uint8_t ***value**
Notify attribute value

bool **is_notify**
True means notify, false means indicate

struct gattc_open_evt_param
#include <*esp_gattc_api.h*> ESP_GATTC_OPEN_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

esp_bd_addr_t **remote_bda**
Remote bluetooth device address

uint16_t **mtu**
MTU size

struct gattc_queue_full_evt_param
#include <*esp_gattc_api.h*> ESP_GATTC_QUEUE_FULL_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

bool **is_full**
The gattc command queue is full or not

struct gattc_read_char_evt_param
#include <*esp_gattc_api.h*> ESP_GATTC_READ_CHAR_EVT, ESP_GATTC_READ_DESCR_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

uint16_t **handle**
Characteristic handle

uint8_t ***value**
Characteristic value

uint16_t **value_len**
Characteristic value length

struct gattc_reg_evt_param
#include <*esp_gattc_api.h*> ESP_GATTC_REG_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **app_id**
Application id which input in register API

struct gattc_reg_for_notify_evt_param
#include <*esp_gattc_api.h*> ESP_GATTC_REG_FOR_NOTIFY_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **handle**
The characteristic or descriptor handle

struct gattc_search_cmpl_evt_param
#include <esp_gattc_api.h> ESP_GATTC_SEARCH_CMPL_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

struct gattc_search_res_evt_param
#include <esp_gattc_api.h> ESP_GATTC_SEARCH_RES_EVT.

Public Members

uint16_t **conn_id**
Connection id

uint16_t **start_handle**
Service start handle

uint16_t **end_handle**
Service end handle

esp_gatt_id_t **srvc_id**
Service id, include service uuid and other information

bool **is_primary**
True if this is the primary service

struct gattc_set_assoc_addr_cmp_evt_param
#include <esp_gattc_api.h> ESP_GATTC_SET_ASSOC_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

struct gattc_srvc_chg_evt_param
#include <esp_gattc_api.h> ESP_GATTC_SRVC_CHG_EVT.

Public Members

esp_bd_addr_t **remote_bda**
Remote bluetooth device address

struct gattc_unreg_for_notify_evt_param

#include <esp_gattc_api.h> ESP_GATTC_UNREG_FOR_NOTIFY_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **handle**

The characteristic or descriptor handle

struct gattc_write_evt_param

#include <esp_gattc_api.h> ESP_GATTC_WRITE_CHAR_EVT, ESP_GATTC_PREP_WRITE_EVT, ESP_GATTC_WRITE_DESCR_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

uint16_t **handle**

The Characteristic or descriptor handle

uint16_t **offset**

The prepare write offset, this value is valid only when prepare write

Type Definitions

typedef void (*__esp_gattc_cb_t__)(*esp_gattc_cb_event_t* event, *esp_gatt_if_t* gattc_if, *esp_ble_gattc_cb_param_t* *param)

GATT Client callback function type.

Parameters

- `event`: : Event type
- `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

ESP_GATTC_SET_ASSOC_EVT = 44
When the ble gattc set the associated address complete, the event comes

ESP_GATTC_GET_ADDR_LIST_EVT = 45
When the ble get gattc address list in cache finish, the event comes

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/api/esp_blufi_api.h

Functions

esp_err_t **esp_blufi_register_callbacks**(*esp_blufi_callbacks_t* **callbacks*)

This function is called to receive blufi callback event.

Return ESP_OK - success, other - failed

Parameters

- `callbacks`: callback functions

esp_err_t **esp_blufi_profile_init**(void)

This function is called to initialize blufi_profile.

Return ESP_OK - success, other - failed

esp_err_t **esp_blufi_profile_deinit**(void)

This function is called to de-initialize blufi_profile.

Return ESP_OK - success, other - failed

esp_err_t **esp_blufi_send_wifi_conn_report**(*wifi_mode_t* *opmode*, *esp_blufi_sta_conn_state_t* *sta_conn_state*, uint8_t *softap_conn_num*, *esp_blufi_extra_info_t* **extra_info*)

This function is called to send wifi connection report.

Return ESP_OK - success, other - failed

Parameters

- `opmode`: : wifi opmode

- `sta_conn_state`: : station is already in connection or not
- `softap_conn_num`: : softap connection number
- `extra_info`: : extra information, such as sta_ssid, softap_ssid and etc.

esp_err_t **esp_blufi_send_wifi_list** (uint16_t *apCount*, *esp_blufi_ap_record_t* **list*)
This function is called to send wifi list.

Return ESP_OK - success, other - failed

Parameters

- `apCount`: : wifi list count
- `list`: : wifi list

uint16_t **esp_blufi_get_version** (void)
Get BLUFI profile version.

Return Most 8bit significant is Great version, Least 8bit is Sub version

esp_err_t **esp_blufi_close** (*esp_gatt_if_t* *gatts_if*, uint16_t *conn_id*)
Close a connection a remote device.

Return

- ESP_OK : success
- other : failed

Parameters

- `gatts_if`: GATT server access interface
- `conn_id`: connection ID to be closed.

esp_err_t **esp_blufi_send_error_info** (*esp_blufi_error_state_t* *state*)
This function is called to send blufi error information.

Return ESP_OK - success, other - failed

Parameters

- `state`: : error state

esp_err_t **esp_blufi_send_custom_data** (uint8_t **data*, uint32_t *data_len*)
This function is called to custom data.

Return ESP_OK - success, other - failed

Parameters

- `data`: : custom data value
- `data_len`: : the length of custom data

Unions

union esp_blufi_cb_param_t

#include <esp_blufi_api.h> BLUFI callback parameters union.

Public Members

struct *esp_blufi_cb_param_t*::*blufi_init_finish_evt_param* **init_finish**
Blufi callback param of ESP_BLUFI_EVENT_INIT_FINISH

struct *esp_blufi_cb_param_t*::*blufi_deinit_finish_evt_param* **deinit_finish**
Blufi callback param of ESP_BLUFI_EVENT_DEINIT_FINISH

struct *esp_blufi_cb_param_t*::*blufi_set_wifi_mode_evt_param* **wifi_mode**
Blufi callback param of ESP_BLUFI_EVENT_INIT_FINISH

struct *esp_blufi_cb_param_t*::*blufi_connect_evt_param* **connect**
Blufi callback param of ESP_BLUFI_EVENT_CONNECT

struct *esp_blufi_cb_param_t*::*blufi_disconnect_evt_param* **disconnect**
Blufi callback param of ESP_BLUFI_EVENT_DISCONNECT

struct *esp_blufi_cb_param_t*::*blufi_recv_sta_bssid_evt_param* **sta_bssid**
Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_BSSID

struct *esp_blufi_cb_param_t*::*blufi_recv_sta_ssid_evt_param* **sta_ssid**
Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_SSID

struct *esp_blufi_cb_param_t*::*blufi_recv_sta_passwd_evt_param* **sta_passwd**
Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_PASSWD

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_ssid_evt_param* **softap_ssid**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_SSID

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_passwd_evt_param* **softap_passwd**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_max_conn_num_evt_param* **softap_max_conn_num**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_auth_mode_evt_param* **softap_auth_mode**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE

struct *esp_blufi_cb_param_t*::*blufi_recv_softap_channel_evt_param* **softap_channel**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL

struct *esp_blufi_cb_param_t*::*blufi_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_recv_server_pkey_evt_param* **server_pkey**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY

struct *esp_blufi_cb_param_t*::*blufi_get_error_evt_param* **report_error**
Blufi callback param of ESP_BLUFI_EVENT_REPORT_ERROR

struct *esp_blufi_cb_param_t*::*blufi_recv_custom_data_evt_param* **custom_data**
Blufi callback param of ESP_BLUFI_EVENT_RECV_CUSTOM_DATA

struct blufi_connect_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_CONNECT.

Public Members

esp_bd_addr_t **remote_bda**
Blufi Remote bluetooth device address

uint8_t **server_if**
server interface

uint16_t **conn_id**
Connection id

struct blufi_deinit_finish_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_DEINIT_FINISH.

Public Members

esp_blufi_deinit_state_t **state**
De-initial status

struct blufi_disconnect_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_DISCONNECT.

Public Members

esp_bd_addr_t **remote_bda**
Blufi Remote bluetooth device address

struct blufi_get_error_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_REPORT_ERROR.

Public Members

esp_blufi_error_state_t **state**
Blufi error state

struct blufi_init_finish_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_INIT_FINISH.

Public Members

esp_blufi_init_state_t **state**
Initial status

struct blufi_recv_ca_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CA_CERT.

Public Members

uint8_t ***cert**
CA certificate point

int **cert_len**
CA certificate length

struct blufi_recv_client_cert_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CLIENT_CERT

Public Members

uint8_t ***cert**
Client certificate point

int **cert_len**
Client certificate length

struct blufi_recv_client_pkey_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY

Public Members

uint8_t ***pkey**
Client Private Key point, if Client certificate not contain Key

int **pkey_len**
Client Private key length

struct blufi_recv_custom_data_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CUSTOM_DATA.

Public Members

uint8_t ***data**
Custom data

uint32_t **data_len**
Custom data Length

struct blufi_recv_server_cert_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SERVER_CERT

Public Members

uint8_t ***cert***
Client certificate point

int ***cert_len***
Client certificate length

struct blufi_recv_server_pkey_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY

Public Members

uint8_t ***pkey***
Client Private Key point, if Client certificate not contain Key

int ***pkey_len***
Client Private key length

struct blufi_recv_softap_auth_mode_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE.

Public Members

wifi_auth_mode_t ***auth_mode***
Authentication mode

struct blufi_recv_softap_channel_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL.

Public Members

uint8_t ***channel***
Authentication mode

struct blufi_recv_softap_max_conn_num_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM.

Public Members

int ***max_conn_num***
SSID

struct blufi_recv_softap_passwd_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD.

Public Members

uint8_t ***passwd***
Password

int ***passwd_len***
Password Length

struct blufi_recv_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_recv_sta_bssid_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_STA_BSSID.

Public Members

uint8_t **bssid**[6]
BSSID

struct blufi_recv_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_recv_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_recv_username_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_USERNAME.

Public Members

uint8_t ***name**
Username point

int **name_len**
Username length

struct blufi_set_wifi_mode_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_SET_WIFI_MODE.

Public Members

wifi_mode_t **op_mode**
Wifi operation mode

Structures

struct esp_blufi_extra_info_t
BLUFI extra information structure.

Public Members

uint8_t **sta_bssid**[6]
BSSID of station interface

bool **sta_bssid_set**
is BSSID of station interface set

uint8_t ***sta_ssid**
SSID of station interface

int **sta_ssid_len**
length of SSID of station interface

uint8_t ***sta_passwd**
password of station interface

int **sta_passwd_len**
length of password of station interface

uint8_t ***softap_ssid**
SSID of softap interface

int **softap_ssid_len**
length of SSID of softap interface

uint8_t ***softap_passwd**
password of station interface

int **softap_passwd_len**
length of password of station interface

uint8_t **softap_authmode**
authentication mode of softap interface

bool **softap_authmode_set**
is authentication mode of softap interface set

uint8_t **softap_max_conn_num**
max connection number of softap interface

bool **softap_max_conn_num_set**
is max connection number of softap interface set

uint8_t **softap_channel**
channel of softap interface

bool **softap_channel_set**
is channel of softap interface set

struct esp_blufi_ap_record_t

Description of an WiFi AP.

Public Members

uint8_t **ssid**[33]

SSID of AP

int8_t **rssi**

signal strength of AP

struct esp_blufi_callbacks_t

BLUFI callback functions type.

Public Members

esp_blufi_event_cb_t **event_cb**

BLUFI event callback

esp_blufi_negotiate_data_handler_t **negotiate_data_handler**

BLUFI negotiate data function for negotiate share key

esp_blufi_encrypt_func_t **encrypt_func**

BLUFI encrypt data function with share key generated by negotiate_data_handler

esp_blufi_decrypt_func_t **decrypt_func**

BLUFI decrypt data function with share key generated by negotiate_data_handler

esp_blufi_checksum_func_t **checksum_func**

BLUFI check sum function (FCS)

Type Definitions

typedef void (*__esp_blufi_event_cb_t__)(*esp_blufi_cb_event_t* event, *esp_blufi_cb_param_t* *param)

BLUFI event callback function type.

Parameters

- `event`: : Event type
- `param`: : Point to callback parameter, currently is union type

typedef void (*__esp_blufi_negotiate_data_handler_t__)(uint8_t *data, int len, uint8_t **output_data, int *output_len, bool *need_free)

BLUFI negotiate data handler.

Parameters

- `data`: : data from phone
- `len`: : length of data from phone
- `output_data`: : data want to send to phone
- `output_len`: : length of data want to send to phone

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

ESP_BLUFI_EVENT_GET_WIFI_LIST

ESP_BLUFI_EVENT_REPORT_ERROR

ESP_BLUFI_EVENT_RECV_CUSTOM_DATA

enum esp_blufi_sta_conn_state_t

BLUFI config status.

Values:

ESP_BLUFI_STA_CONN_SUCCESS = 0x00

ESP_BLUFI_STA_CONN_FAIL = 0x01

enum esp_blufi_init_state_t

BLUFI init status.

Values:

ESP_BLUFI_INIT_OK = 0

ESP_BLUFI_INIT_FAILED

enum esp_blufi_deinit_state_t

BLUFI deinit status.

Values:

ESP_BLUFI_DEINIT_OK = 0

ESP_BLUFI_DEINIT_FAILED

enum esp_blufi_error_state_t

Values:

ESP_BLUFI_SEQUENCE_ERROR = 0

ESP_BLUFI_CHECKSUM_ERROR

ESP_BLUFI_DECRYPT_ERROR

ESP_BLUFI_ENCRYPT_ERROR

ESP_BLUFI_INIT_SECURITY_ERROR

ESP_BLUFI_DH_MALLOC_ERROR

ESP_BLUFI_DH_PARAM_ERROR

ESP_BLUFI_READ_PARAM_ERROR

ESP_BLUFI_MAKE_PUBLIC_ERROR

3.3.4 CLASSIC BT

CLASSIC BLUETOOTH GAP API

Overview

Instructions

Application Example

Instructions

API Reference

Header File

- bt/bluedroid/api/include/api/esp_gap_bt_api.h

Functions

uint32_t **esp_bt_gap_get_cod_srvc**(uint32_t *cod*)
get major service field of COD

Return major service bits

Parameters

- `cod`: Class of Device

uint32_t **esp_bt_gap_get_cod_major_dev**(uint32_t *cod*)
get major device field of COD

Return major device bits

Parameters

- `cod`: Class of Device

uint32_t **esp_bt_gap_get_cod_minor_dev**(uint32_t *cod*)
get minor service field of COD

Return minor service bits

Parameters

- `cod`: Class of Device

uint32_t **esp_bt_gap_get_cod_format_type**(uint32_t *cod*)
get format type of COD

Return format type

Parameters

- `cod`: Class of Device

bool **esp_bt_gap_is_valid_cod**(uint32_t *cod*)
decide the integrity of COD

Return

- true if cod is valid
- false otherise

Parameters

- `cod`: Class of Device

esp_err_t **esp_bt_gap_register_callback**(*esp_bt_gap_cb_t callback*)
register callback function. This function should be called after esp_bluedroid_enable() completes successfully

Return

- ESP_OK : Succeed
- ESP_FAIL: others

esp_err_t **esp_bt_gap_set_scan_mode**(*esp_bt_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_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `mode`: : one of the enums of bt_scan_mode_t

esp_err_t **esp_bt_gap_start_discovery**(*esp_bt_inq_mode_t mode*, uint8_t *inq_len*, uint8_t *num_rsps*)
Start device discovery. This function should be called after esp_bluedroid_enable() completes successfully. esp_bt_gap_cb_t will is called with ESP_BT_GAP_DISC_STATE_CHANGED_EVT if discovery is started or halted. esp_bt_gap_cb_t will is called with ESP_BT_GAP_DISC_RES_EVT if discovery result is got.

Return

- ESP_OK : Succeed

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if invalid parameters are provided
- ESP_FAIL: others

Parameters

- `mode`: - inquiry mode
- `inq_len`: - inquiry duration in 1.28 sec units, ranging from 0x01 to 0x30
- `num_rsps`: - number of inquiry responses that can be received, value 0 indicates an unlimited number of responses

esp_err_t **esp_bt_gap_cancel_discovery**(void)

Cancel device discovery. This function should be called after esp_bluedroid_enable() completes successfully esp_bt_gap_cb_t will is called with ESP_BT_GAP_DISC_STATE_CHANGED_EVT if discovery is stopped.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_bt_gap_get_remote_services**(*esp_bd_addr_t remote_bda*)

Start SDP to get remote services. This function should be called after esp_bluedroid_enable() completes successfully. esp_bt_gap_cb_t will is called with ESP_BT_GAP_RMT_SRVCS_EVT after service discovery ends.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_bt_gap_get_remote_service_record**(*esp_bd_addr_t remote_bda*, *esp_bt_uuid_t *uuid*)

Start SDP to look up the service matching uuid on the remote device. This function should be called after esp_bluedroid_enable() completes successfully.

esp_bt_gap_cb_t will is called with ESP_BT_GAP_RMT_SRVC_REC_EVT after service discovery ends

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

uint8_t ***esp_bt_gap_resolve_eir_data**(uint8_t **eir*, *esp_bt_eir_type_t type*, uint8_t **length*)

This function is called to get EIR data for a specific type.

Return pointer of starting position of eir data excluding eir data type, NULL if not found

Parameters

- `eir`: - pointer of raw eir data to be resolved
- `type`: - specific EIR data type

- `length`: - return the length of EIR data excluding fields of length and data type

esp_err_t **esp_bt_gap_set_cod**(*esp_bt_cod_t cod*, *esp_bt_cod_mode_t mode*)

This function is called to set class of device. esp_bt_gap_cb_t will is called with ESP_BT_GAP_SET_COD_EVT after set COD ends Some profile have special restrictions on class of device, changes may cause these profile do not work.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if param is invalid
- ESP_FAIL: others

Parameters

- `cod`: - class of device
- `mode`: - setting mode

esp_err_t **esp_bt_gap_get_cod**(*esp_bt_cod_t *cod*)

This function is called to get class of device.

Return

- ESP_OK : Succeed
- ESP_FAIL: others

Parameters

- `cod`: - class of device

esp_err_t **esp_bt_gap_read_rssi_delta**(*esp_bd_addr_t remote_addr*)

This function is called to read RSSI delta by address after connected. The RSSI value returned by ESP_BT_GAP_READ_RSSI_DELTA_EVT.

Return

- ESP_OK : Succeed
- ESP_FAIL: others

Parameters

- `remote_addr`: - remote device address, corresponding to a certain connection handle.

esp_err_t **esp_bt_gap_remove_bond_device**(*esp_bd_addr_t bd_addr*)

Removes a device from the security database list of peer device.

Return - ESP_OK : success

- ESP_FAIL : failed

Parameters

- `bd_addr`: : BD address of the peer device

int **esp_bt_gap_get_bond_device_num**(void)

Get the device number from the security database list of peer device. It will return the device bonded number immediately.

Return - >= 0 : bonded devices number.

- ESP_FAIL : failed

esp_err_t **esp_bt_gap_get_bond_device_list**(int **dev_num*, *esp_bd_addr_t* **dev_list*)

Get the device from the security database list of peer device. It will return the device bonded information immediately.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `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_bd_addr_t` type. Use for storing the bonded devices address. The dev_list should be allocated by who call this API.

esp_err_t **esp_bt_gap_set_pin**(*esp_bt_pin_type_t* *pin_type*, uint8_t *pin_code_len*, *esp_bt_pin_code_t* *pin_code*)

Set pin type and default pin code for legacy pairing.

Return - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

Parameters

- `pin_type`: Use variable or fixed pin. If pin_type is ESP_BT_PIN_TYPE_VARIABLE, pin_code and pin_code_len will be ignored, and ESP_BT_GAP_PIN_REQ_EVT will come when control requests for pin code. Else, will use fixed pin code and not callback to users.
- `pin_code_len`: Length of pin_code
- `pin_code`: Pin_code

esp_err_t **esp_bt_gap_pin_reply**(*esp_bd_addr_t* *bd_addr*, bool *accept*, uint8_t *pin_code_len*, *esp_bt_pin_code_t* *pin_code*)

Reply the pin_code to the peer device for legacy pairing when ESP_BT_GAP_PIN_REQ_EVT is coming.

Return - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

Parameters

- `bd_addr`: BD address of the peer
- `accept`: Pin_code reply successful or declined.

- `pin_code_len`: Length of pin_code
- `pin_code`: Pin_code

Unions

union esp_bt_gap_cb_param_t
#include <esp_gap_bt_api.h> A2DP state callback parameters.

Public Members

struct *esp_bt_gap_cb_param_t*::*disc_res_param* **disc_res**
discovery result paramter struct

struct *esp_bt_gap_cb_param_t*::*disc_state_changed_param* **disc_st_chg**
discovery state changed parameter struct

struct *esp_bt_gap_cb_param_t*::*rmt_srvcs_param* **rmt_srvcs**
services of remote device parameter struct

struct *esp_bt_gap_cb_param_t*::*rmt_srvc_rec_param* **rmt_srvc_rec**
specific service record from remote device parameter struct

struct *esp_bt_gap_cb_param_t*::*read_rssi_delta_param* **read_rssi_delta**
read rssi parameter struct

struct *esp_bt_gap_cb_param_t*::*auth_cmpl_param* **auth_cmpl**
authentication complete parameter struct

struct *esp_bt_gap_cb_param_t*::*pin_req_param* **pin_req**
pin request parameter struct

struct auth_cmpl_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_AUTH_CMPL_EVT.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

esp_bt_status_t **stat**
authentication complete status

uint8_t **device_name**[**ESP_BT_GAP_MAX_BDNAME_LEN** + 1]
device name

struct disc_res_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_DISC_RES_EVT.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

int **num_prop**
number of properties got

esp_bt_gap_dev_prop_t ***prop**
properties discovered from the new device

struct disc_state_changed_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_DISC_STATE_CHANGED_EVT.

Public Members

esp_bt_gap_discovery_state_t **state**
discovery state

struct pin_req_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_PIN_REQ_EVT.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

bool **min_16_digit**
TRUE if the pin returned must be at least 16 digits

struct read_rssi_delta_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_READ_RSSI_DELTA_EVT *.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

esp_bt_status_t **stat**
read rssi status

int8_t **rssi_delta**
rssi delta value range -128 ~127, The value zero indicates that the RSSI is inside the Golden Receive Power Range, the Golden Receive Power Range is from ESP_BT_GAP_RSSI_LOW_THRLD to ESP_BT_GAP_RSSI_HIGH_THRLD

struct rmt_srvc_rec_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_RMT_SRVC_REC_EVT.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

esp_bt_status_t **stat**
service search status

struct rmt_srvcs_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_RMT_SRVCS_EVT.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

esp_bt_status_t **stat**
service search status

int **num_uuids**
number of UUID in uuid_list

esp_bt_uuid_t ***uuid_list**
list of service UUIDs of remote device

Structures

struct esp_bt_cod_t
Class of device.

Public Members

uint32_t **reserved_2**
undefined

uint32_t **minor**
minor class

uint32_t **major**
major class

uint32_t **service**
service class

uint32_t **reserved_8**
undefined

struct esp_bt_gap_dev_prop_t
Bluetooth Device Property Descriptor.

Public Members

esp_bt_gap_dev_prop_type_t **type**
device property type

int **len**
device property value length

void ***val**
devlice prpoerty value

Macros

ESP_BT_GAP_RSSI_HIGH_THRLD
RSSI threshold.

High RSSI threshold

ESP_BT_GAP_RSSI_LOW_THRLD
Low RSSI threshold

ESP_BT_GAP_MAX_BDNAME_LEN
Maximum bytes of Bluetooth device name.

ESP_BT_GAP_EIR_DATA_LEN
Maximum size of EIR Significant part.

ESP_BT_PIN_CODE_LEN
Max pin code length

ESP_BT_COD_SRVC_BIT_MASK
Bits of major service class field.

Major service bit mask

ESP_BT_COD_SRVC_BIT_OFFSET
Major service bit offset

ESP_BT_COD_MAJOR_DEV_BIT_MASK
Bits of major device class field.

Major device bit mask

ESP_BT_COD_MAJOR_DEV_BIT_OFFSET
Major device bit offset

ESP_BT_COD_MINOR_DEV_BIT_MASK
Bits of minor device class field.

Minor device bit mask

ESP_BT_COD_MINOR_DEV_BIT_OFFSET
Minor device bit offset

ESP_BT_COD_FORMAT_TYPE_BIT_MASK
Bits of format type.

Format type bit mask

ESP_BT_COD_FORMAT_TYPE_BIT_OFFSET
Format type bit offset

ESP_BT_COD_FORMAT_TYPE_1
Class of device format type 1.

ESP_BT_GAP_MIN_INQ_LEN
Minimum and Maximum inquiry length Minimum inquiry duration, unit is 1.28s

ESP_BT_GAP_MAX_INQ_LEN
Maximum inquiry duration, unit is 1.28s

Type Definitions

typedef uint8_t **esp_bt_pin_code_t**[**ESP_BT_PIN_CODE_LEN**]
Pin Code (upto 128 bits) MSB is 0

typedef void (***esp_bt_gap_cb_t**) (*esp_bt_gap_cb_event_t* event, *esp_bt_gap_cb_param_t* *param)
bluetooth GAP callback function type

Parameters

- `event`: : Event type
- `param`: : Pointer to callback parameter

Enumerations

enum esp_bt_cod_mode_t

class of device settings

Values:

ESP_BT_SET_COD_MAJOR_MINOR = 0x01
overwrite major, minor class

ESP_BT_SET_COD_SERVICE_CLASS = 0x02
set the bits in the input, the current bit will remain

ESP_BT_CLR_COD_SERVICE_CLASS = 0x04
clear the bits in the input, others will remain

ESP_BT_SET_COD_ALL = 0x08
overwrite major, minor, set the bits in service class

ESP_BT_INIT_COD = 0x0a
overwrite major, minor, and service class

enum esp_bt_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

enum esp_bt_gap_dev_prop_type_t

Bluetooth Device Property type.

Values:

ESP_BT_GAP_DEV_PROP_BDNAME = 1
Bluetooth device name, value type is int8_t []

ESP_BT_GAP_DEV_PROP_COD
Class of Device, value type is uint32_t

ESP_BT_GAP_DEV_PROP_RSSI
Received Signal strength Indication, value type is int8_t, ranging from -128 to 127

ESP_BT_GAP_DEV_PROP_EIR
Extended Inquiry Response, value type is uint8_t []

enum esp_bt_eir_type_t

Extended Inquiry Response data type.

Values:

ESP_BT_EIR_TYPE_FLAGS = 0x01
Flag with information such as BR/EDR and LE support

ESP_BT_EIR_TYPE_INCMPL_16BITS_UUID = 0x02
Incomplete list of 16-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_16BITS_UUID = 0x03
Complete list of 16-bit service UUIDs

ESP_BT_EIR_TYPE_INCMPL_32BITS_UUID = 0x04
Incomplete list of 32-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_32BITS_UUID = 0x05
Complete list of 32-bit service UUIDs

ESP_BT_EIR_TYPE_INCMPL_128BITS_UUID = 0x06
Incomplete list of 128-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_128BITS_UUID = 0x07
Complete list of 128-bit service UUIDs

ESP_BT_EIR_TYPE_SHORT_LOCAL_NAME = 0x08
Shortened Local Name

ESP_BT_EIR_TYPE_CMPL_LOCAL_NAME = 0x09
Complete Local Name

ESP_BT_EIR_TYPE_TX_POWER_LEVEL = 0x0a
Tx power level, value is 1 octet ranging from -127 to 127, unit is dBm

ESP_BT_EIR_TYPE_MANU_SPECIFIC = 0xff
Manufacturer specific data

enum esp_bt_cod_srvc_t
Major service class field of Class of Device, mutiple bits can be set.

Values:

ESP_BT_COD_SRVC_NONE = 0
None indicates an invalid value

ESP_BT_COD_SRVC_LMTD_DISCOVER = 0x1
Limited Discoverable Mode

ESP_BT_COD_SRVC_POSITIONING = 0x8
Positioning (Location identification)

ESP_BT_COD_SRVC_NETWORKING = 0x10
Networking, e.g. LAN, Ad hoc

ESP_BT_COD_SRVC_RENDERING = 0x20
Rendering, e.g. Printing, Speakers

ESP_BT_COD_SRVC_CAPTURING = 0x40
Capturing, e.g. Scanner, Microphone

ESP_BT_COD_SRVC_OBJ_TRANSFER = 0x80
Object Transfer, e.g. v-Inbox, v-Folder

ESP_BT_COD_SRVC_AUDIO = 0x100
Audio, e.g. Speaker, Microphone, Headerset service

ESP_BT_COD_SRVC_TELEPHONY = 0x200
Telephony, e.g. Cordless telephony, Modem, Headset service

ESP_BT_COD_SRVC_INFORMATION = 0x400
Information, e.g., WEB-server, WAP-server

enum esp_bt_pin_type_t
Values:

ESP_BT_PIN_TYPE_VARIABLE = 0
Refer to BTM_PIN_TYPE_VARIABLE

ESP_BT_PIN_TYPE_FIXED = 1
Refer to BTM_PIN_TYPE_FIXED

enum esp_bt_cod_major_dev_t
Major device class field of Class of Device.

Values:

ESP_BT_COD_MAJOR_DEV_MISC = 0
Miscellaneous

ESP_BT_COD_MAJOR_DEV_COMPUTER = 1
Computer

ESP_BT_COD_MAJOR_DEV_PHONE = 2
Phone(cellular, cordless, pay phone, modem

ESP_BT_COD_MAJOR_DEV_LAN_NAP = 3
LAN, Network Access Point

ESP_BT_COD_MAJOR_DEV_AV = 4
Audio/Video(headset, speaker, stereo, video display, VCR

ESP_BT_COD_MAJOR_DEV_PERIPHERAL = 5
Peripheral(mouse, joystick, keyboard)

ESP_BT_COD_MAJOR_DEV_IMAGING = 6
Imaging(printer, scanner, camera, display

ESP_BT_COD_MAJOR_DEV_WEARABLE = 7
Wearable

ESP_BT_COD_MAJOR_DEV_TOY = 8
Toy

ESP_BT_COD_MAJOR_DEV_HEALTH = 9
Health

ESP_BT_COD_MAJOR_DEV_UNCATEGORIZED = 31
Uncategorized: device not specified

enum esp_bt_gap_discovery_state_t
Bluetooth Device Discovery state

Values:

ESP_BT_GAP_DISCOVERY_STOPPED
device discovery stopped

ESP_BT_GAP_DISCOVERY_STARTED
device discovery started

enum esp_bt_gap_cb_event_t
BT GAP callback events.

Values:

ESP_BT_GAP_DISC_RES_EVT = 0
device discovery result event

ESP_BT_GAP_DISC_STATE_CHANGED_EVT
discovery state changed event

ESP_BT_GAP_RMT_SRVCS_EVT
get remote services event

ESP_BT_GAP_RMT_SRVC_REC_EVT
get remote service record event

ESP_BT_GAP_AUTH_CMPL_EVT
AUTH complete event

ESP_BT_GAP_PIN_REQ_EVT
Legacy Pairing Pin code request

ESP_BT_GAP_READ_RSSI_DELTA_EVT
read rssi event

ESP_BT_GAP_EVT_MAX

enum esp_bt_inq_mode_t
Inquiry Mode

Values:

ESP_BT_INQ_MODE_GENERAL_INQUIRY
General inquiry mode

ESP_BT_INQ_MODE_LIMITED_INQUIRY
Limited inquiry mode

Bluetooth A2DP API

Overview

Instructions

Application Example

Check bluetooth 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/api/esp_a2dp_api.h

Functions

esp_err_t **esp_a2d_register_callback** (*esp_a2d_cb_t callback*)

Register application callback function to A2DP module. This function should be called only after esp_bluedroid_enable() completes successfully, used by both A2DP source and sink.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

Parameters

- `callback`: A2DP event callback function

esp_err_t **esp_a2d_sink_register_data_callback** (*esp_a2d_sink_data_cb_t callback*)

Register A2DP sink data output function; For now the output is PCM data stream decoded from SBC format. This function should be called only after esp_bluedroid_enable() completes successfully, used only by A2DP sink. The callback is invoked in the context of A2DP sink task whose stack size is configurable through menuconfig.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

Parameters

- `callback`: A2DP sink data callback function

esp_err_t **esp_a2d_sink_init** (void)

Initialize the bluetooth A2DP sink module. This function should be called after esp_bluedroid_enable() completes successfully.

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 to remote bluetooth A2DP source device, must after esp_a2d_sink_init()

Return

- ESP_OK: connect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `remote_bda`: remote bluetooth device address

esp_err_t **esp_a2d_sink_disconnect** (*esp_bd_addr_t remote_bda*)

Disconnect from the remote A2DP source device.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `remote_bda`: remote bluetooth device address

esp_err_t **esp_a2d_media_ctrl** (*esp_a2d_media_ctrl_t ctrl*)

media control commands; this API can be used for both A2DP sink and source

Return

- ESP_OK: control command is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `ctrl`: control commands for A2DP data channel

esp_err_t **esp_a2d_source_init** (void)

Initialize the bluetooth A2DP source module. This function should be called after esp_bluedroid_enable() completes successfully.

Return

- ESP_OK: if the initialization request is sent successfully
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_source_deinit** (void)

De-initialize for A2DP source module. This function should be called only after esp_bluedroid_enable() completes successfully.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled

- ESP_FAIL: others

esp_err_t **esp_a2d_source_register_data_callback**(*esp_a2d_source_data_cb_t callback*)

Register A2DP source data input function; For now the input is PCM data stream. This function should be called only after esp_bluedroid_enable() completes successfully. The callback is invoked in the context of A2DP source task whose stack size is configurable through menuconfig.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

Parameters

- `callback`: A2DP source data callback function

esp_err_t **esp_a2d_source_connect**(*esp_bd_addr_t remote_bda*)

Connect to remote A2DP sink device, must after esp_a2d_source_init()

Return

- ESP_OK: connect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `remote_bda`: remote bluetooth device address

esp_err_t **esp_a2d_source_disconnect**(*esp_bd_addr_t remote_bda*)

Disconnect from the remote A2DP sink device.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `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 *esp_a2d_cb_param_t*::*media_ctrl_stat_param* **media_ctrl_stat**
status in acknowledgement to media control commands

struct a2d_audio_cfg_param
#include <*esp_a2dp_api.h*> ESP_A2D_AUDIO_CFG_EVT.

Public Members

esp_bd_addr_t **remote_bda**
remote bluetooth device address

esp_a2d_mcc_t **mcc**
A2DP media codec capability information

struct a2d_audio_stat_param
#include <*esp_a2dp_api.h*> ESP_A2D_AUDIO_STATE_EVT.

Public Members

esp_a2d_audio_state_t **state**
one of the values from esp_a2d_audio_state_t

esp_bd_addr_t **remote_bda**
remote bluetooth device address

struct a2d_conn_stat_param
#include <*esp_a2dp_api.h*> ESP_A2D_CONNECTION_STATE_EVT.

Public Members

esp_a2d_connection_state_t **state**
one of values from esp_a2d_connection_state_t

esp_bd_addr_t **remote_bda**
remote bluetooth device address

esp_a2d_disc_rsn_t **disc_rsn**
reason of disconnection for "DISCONNECTED"

struct media_ctrl_stat_param
#include <*esp_a2dp_api.h*> ESP_A2D_MEDIA_CTRL_ACK_EVT.

Public Members

esp_a2d_media_ctrl_t **cmd**
media control commands to acknowledge

esp_a2d_media_ctrl_ack_t **status**
acknowledgement to media control commands

Structures

struct esp_a2d_mcc_t
A2DP media codec capabilities union.

Public Members

esp_a2d_mct_t **type**
A2DP media codec type

union *esp_a2d_mcc_t*::**[anonymous] cie**
A2DP codec information element

Macros

ESP_A2D_MCT_SBC
Media codec types supported by A2DP.

SBC

ESP_A2D_MCT_M12
MPEG-1, 2 Audio

ESP_A2D_MCT_M24
MPEG-2, 4 AAC

ESP_A2D_MCT_ATRAC
ATRAC family

ESP_A2D_MCT_NON_A2DP

ESP_A2D_CIE_LEN_SBC

ESP_A2D_CIE_LEN_M12

ESP_A2D_CIE_LEN_M24

ESP_A2D_CIE_LEN_ATRAC

Type Definitions

typedef uint8_t **esp_a2d_mct_t**

typedef void (*__esp_a2d_cb_t__) (*esp_a2d_cb_event_t* event, *esp_a2d_cb_param_t* *param)
A2DP profile callback function type.

Parameters

- `event`: : Event type
- `param`: : Pointer to callback parameter

typedef void (*__esp_a2d_sink_data_cb_t__) (**const** uint8_t *buf, uint32_t len)
A2DP profile data callback function.

Parameters

- `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

typedef int32_t (*__esp_a2d_source_data_cb_t__) (uint8_t *buf, int32_t len)

A2DP source data read callback function.

Return size of bytes read successfully, if the argumetn len is -1, this value is ignored.

Parameters

- `buf`: : buffer to be filled with PCM data stream from higer layer
- `len`: : size(in bytes) of data block to be copied to buf. -1 is an indication to user that data buffer shall be flushed

Enumerations

enum esp_a2d_connection_state_t

Bluetooth A2DP connection states.

Values:

ESP_A2D_CONNECTION_STATE_DISCONNECTED = 0

connection released

ESP_A2D_CONNECTION_STATE_CONNECTING

connecting remote device

ESP_A2D_CONNECTION_STATE_CONNECTED

connection established

ESP_A2D_CONNECTION_STATE_DISCONNECTING

disconnecting remote device

enum esp_a2d_disc_rsn_t

Bluetooth A2DP disconnection reason.

Values:

ESP_A2D_DISC_RSN_NORMAL = 0

Finished disconnection that is initiated by local or remote device

ESP_A2D_DISC_RSN_ABNORMAL

Abnormal disconnection caused by signal loss

enum esp_a2d_audio_state_t

Bluetooth A2DP datapath states.

Values:

ESP_A2D_AUDIO_STATE_REMOTE_SUSPEND = 0

audio stream datapath suspended by remote device

ESP_A2D_AUDIO_STATE_STOPPED

audio stream datapath stopped

ESP_A2D_AUDIO_STATE_STARTED

audio stream datapath started

enum esp_a2d_media_ctrl_ack_t

A2DP media control command acknowledgement code.

Values:

ESP_A2D_MEDIA_CTRL_ACK_SUCCESS = 0

media control command is acknowledged with success

ESP_A2D_MEDIA_CTRL_ACK_FAILURE

media control command is acknowledged with failure

ESP_A2D_MEDIA_CTRL_ACK_BUSY

media control command is rejected, as previous command is not yet acknowledged

enum esp_a2d_media_ctrl_t

A2DP media control commands.

Values:

ESP_A2D_MEDIA_CTRL_NONE = 0

dummy command

ESP_A2D_MEDIA_CTRL_CHECK_SRC_RDY

check whether AVDTP is connected, only used in A2DP source

ESP_A2D_MEDIA_CTRL_START

command to set up media transmission channel

ESP_A2D_MEDIA_CTRL_STOP

command to stop media transmission

ESP_A2D_MEDIA_CTRL_SUSPEND

command to suspend media transmission

enum esp_a2d_cb_event_t

A2DP callback events.

Values:

ESP_A2D_CONNECTION_STATE_EVT = 0

connection state changed event

ESP_A2D_AUDIO_STATE_EVT

audio stream transmission state changed event

ESP_A2D_AUDIO_CFG_EVT

audio codec is configured, only used for A2DP SINK

ESP_A2D_MEDIA_CTRL_ACK_EVT

acknowledge event in response to media control commands

BT AVRCP APIs

Overview

Bluetooth AVRCP reference APIs.

Instructions

Application Example

Instructions

API Reference

Header File

- bt/bluedroid/api/include/api/esp_avrc_api.h

Functions

esp_err_t **esp_avrc_ct_register_callback**(*esp_avrc_ct_cb_t callback*)

Register application callbacks to AVRCP module; 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> AVRC controller callback parameters.

Public Members

struct *esp_avrc_ct_cb_param_t*::*avrc_ct_conn_stat_param* **conn_stat**

AVRC 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**

AVRC 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 AVRC 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 **tl**

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 miliseconds

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

SPP API

Overview

Instructions

Application Example

Check bluetooth folder in ESP-IDF examples, which contains the following application:

- This is a SPP demo. This demo can discover the service, connect, send and recive SPP data bluetooth/bt_spp_acceptor, bluetooth/bt_spp_initiator

API Reference

Header File

- bt/bluedroid/api/include/api/esp_spp_api.h

Functions

esp_err_t **esp_spp_register_callback**(esp_spp_cb_t *callback*)

This function is called to init callbacks with SPP module.

Return

- ESP_OK: success
- other: failed

Parameters

- `callback`: pointer to the init callback function.

esp_err_t **esp_spp_init**(*esp_spp_mode_t mode*)

This function is called to init SPP.

Return

- ESP_OK: success

- other: failed

Parameters

- `mode`: Choose the mode of SPP, ESP_SPP_MODE_CB or ESP_SPP_MODE_VFS.

esp_err_t **esp_spp_deinit**()

This function is called to uninit SPP.

Return

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_start_discovery**(*esp_bd_addr_t bd_addr*)

This function is called to performs service discovery for the services provided by the given peer device. When the operation is complete the callback function will be called with a ESP_SPP_DISCOVERY_COMP_EVT.

Return

- ESP_OK: success
- other: failed

Parameters

- `bd_addr`: Remote device bluetooth device address.

esp_err_t **esp_spp_connect**(*esp_spp_sec_t sec_mask*, *esp_spp_role_t role*, uint8_t *remote_scn*, *esp_bd_addr_t peer_bd_addr*)

This function makes an SPP conection to a remote BD Address. When the connection is initiated or failed to initiate, the callback is called with ESP_SPP_CL_INIT_EVT. When the connection is established or failed, the callback is called with ESP_SPP_OPEN_EVT.

Return

- ESP_OK: success
- other: failed

Parameters

- `sec_mask`: Security Setting Mask. Suggest to use ESP_SPP_SEC_NONE, ESP_SPP_SEC_AUTHORIZE or ESP_SPP_SEC_AUTHENTICATE only.
- `role`: Master or slave.
- `remote_scn`: Remote device bluetooth device SCN.
- `peer_bd_addr`: Remote device bluetooth device address.

esp_err_t **esp_spp_disconnect**(uint32_t *handle*)

This function closes an SPP connection.

Return

- ESP_OK: success
- other: failed

Parameters

- `handle`: The connection handle.

esp_err_t **esp_spp_start_srv**(*esp_spp_sec_t sec_mask*, *esp_spp_role_t role*, uint8_t *local_scn*, **const** char **name*)

This function create a SPP server and starts listening for an SPP connection request from a remote Bluetooth device. When the server is started successfully, the callback is called with ESP_SPP_START_EVT. When the connection is established, the callback is called with ESP_SPP_SRV_OPEN_EVT.

Return

- ESP_OK: success
- other: failed

Parameters

- `sec_mask`: Security Setting Mask. Security Setting Mask. Suggest to use ESP_SPP_SEC_NONE, ESP_SPP_SEC_AUTHORIZE or ESP_SPP_SEC_AUTHENTICATE only.
- `role`: Master or slave.
- `local_scn`: The specific channel you want to get. If channel is 0, means get any channel.
- `name`: Server's name.

esp_err_t **esp_spp_write**(uint32_t *handle*, int *len*, uint8_t **p_data*)

This function is used to write data, olny for ESP_SPP_MODE_CB.

Return

- ESP_OK: success
- other: failed

Parameters

- `handle`: The connection handle.
- `len`: The length of the data written.
- `p_data`: The data written.

esp_err_t **esp_spp_vfs_register**(void)

This function is used to register VFS.

Return

- ESP_OK: success
- other: failed

Unions

union esp_spp_cb_param_t

#include <esp_spp_api.h> SPP callback parameters union.

Public Members

struct *esp_spp_cb_param_t*::*spp_init_evt_param* **init**

SPP callback param of SPP_INIT_EVT

struct *esp_spp_cb_param_t*::*spp_discovery_comp_evt_param* **disc_comp**
SPP callback param of SPP_DISCOVERY_COMP_EVT

struct *esp_spp_cb_param_t*::*spp_open_evt_param* **open**
SPP callback param of ESP_SPP_OPEN_EVT

struct *esp_spp_cb_param_t*::*spp_srv_open_evt_param* **srv_open**
SPP callback param of ESP_SPP_SRV_OPEN_EVT

struct *esp_spp_cb_param_t*::*spp_close_evt_param* **close**
SPP callback param of ESP_SPP_CLOSE_EVT

struct *esp_spp_cb_param_t*::*spp_start_evt_param* **start**
SPP callback param of ESP_SPP_START_EVT

struct *esp_spp_cb_param_t*::*spp_cl_init_evt_param* **cl_init**
SPP callback param of ESP_SPP_CL_INIT_EVT

struct *esp_spp_cb_param_t*::*spp_write_evt_param* **write**
SPP callback param of ESP_SPP_WRITE_EVT

struct *esp_spp_cb_param_t*::*spp_data_ind_evt_param* **data_ind**
SPP callback param of ESP_SPP_DATA_IND_EVT

struct *esp_spp_cb_param_t*::*spp_cong_evt_param* **cong**
SPP callback param of ESP_SPP_CONG_EVT

struct spp_cl_init_evt_param
#include <esp_spp_api.h> ESP_SPP_CL_INIT_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

uint8_t **sec_id**
security ID used by this server

bool **use_co**
TRUE to use co_rfc_data

struct spp_close_evt_param
#include <esp_spp_api.h> ESP_SPP_CLOSE_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **port_status**
PORT status

uint32_t **handle**
The connection handle

bool **async**
FALSE, if local initiates disconnect

struct spp_cong_evt_param
#include <*esp_spp_api.h*> ESP_SPP_CONG_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

bool **cong**
TRUE, congested. FALSE, uncongested

struct spp_data_ind_evt_param
#include <*esp_spp_api.h*> ESP_SPP_DATA_IND_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

uint16_t **len**
The length of data

uint8_t ***data**
The data recived

struct spp_discovery_comp_evt_param
#include <*esp_spp_api.h*> SPP_DISCOVERY_COMP_EVT.

Public Members

esp_spp_status_t **status**
status

uint8_t **scn_num**
The num of scn_num

uint8_t **scn**[**ESP_SPP_MAX_SCN**]
channel #

struct spp_init_evt_param
#include <*esp_spp_api.h*> SPP_INIT_EVT.

Public Members

esp_spp_status_t **status**
status

struct spp_open_evt_param
#include <*esp_spp_api.h*> ESP_SPP_OPEN_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

int **fd**
The file descriptor olny for ESP_SPP_MODE_VFS

esp_bd_addr_t **rem_bda**
The peer address

struct spp_srv_open_evt_param
#include <esp_spp_api.h> ESP_SPP_SRV_OPEN_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

uint32_t **new_listen_handle**
The new listen handle

int **fd**
The file descriptor olny for ESP_SPP_MODE_VFS

esp_bd_addr_t **rem_bda**
The peer address

struct spp_start_evt_param
#include <esp_spp_api.h> ESP_SPP_START_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

uint8_t **sec_id**
security ID used by this server

bool **use_co**
TRUE to use co_rfc_data

struct spp_write_evt_param
#include <esp_spp_api.h> ESP_SPP_WRITE_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t **handle**
The connection handle

int **len**
The length of the data written.

bool **cong**
congestion status

Macros

ESP_SPP_SEC_NONE
No security. relate to BTA_SEC_NONE in bta/bta_api.h

ESP_SPP_SEC_AUTHORIZE
Authorization required (only needed for out going connection) relate to BTA_SEC_AUTHORIZE in bta/bta_api.h

ESP_SPP_SEC_AUTHENTICATE
Authentication required. relate to BTA_SEC_AUTHENTICATE in bta/bta_api.h

ESP_SPP_SEC_ENCRYPT
Encryption required. relate to BTA_SEC_ENCRYPT in bta/bta_api.h

ESP_SPP_SEC_MODE4_LEVEL4
Mode 4 level 4 service, i.e. incoming/outgoing MITM and P-256 encryption relate to BTA_SEC_MODE4_LEVEL4 in bta/bta_api.h

ESP_SPP_SEC_MITM
Man-In-The_Middle protection relate to BTA_SEC_MITM in bta/bta_api.h

ESP_SPP_SEC_IN_16_DIGITS
Min 16 digit for pin code relate to BTA_SEC_IN_16_DIGITS in bta/bta_api.h

ESP_SPP_MAX_MTU
SPP max MTU

ESP_SPP_MAX_SCN
SPP max SCN

Type Definitions

typedef uint16_t **esp_spp_sec_t**

typedef void() esp_spp_cb_t(esp_spp_cb_event_t event, esp_spp_cb_param_t *param)
SPP callback function type.

Parameters

- `event`: Event type
- `param`: Point to callback parameter, currently is union type

Enumerations

enum esp_spp_status_t

Values:

ESP_SPP_SUCCESS = 0
Successful operation.

ESP_SPP_FAILURE
Generic failure.

ESP_SPP_BUSY
Temporarily can not handle this request.

ESP_SPP_NO_DATA
no data.

ESP_SPP_NO_RESOURCE
No more set pm control block

enum esp_spp_role_t

Values:

ESP_SPP_ROLE_MASTER = 0
Role: master

ESP_SPP_ROLE_SLAVE = 1
Role: slave

enum esp_spp_mode_t

Values:

ESP_SPP_MODE_CB = 0
When data is coming, a callback will come with data

ESP_SPP_MODE_VFS = 1
Use VFS to write/read data

enum esp_spp_cb_event_t

SPP callback function events.

Values:

ESP_SPP_INIT_EVT = 0
When SPP is inited, the event comes

ESP_SPP_DISCOVERY_COMP_EVT = 8
When SDP discovery complete, the event comes

ESP_SPP_OPEN_EVT = 26
When SPP Client connection open, the event comes

ESP_SPP_CLOSE_EVT = 27
When SPP connection closed, the event comes

ESP_SPP_START_EVT = 28
When SPP server started, the event comes

ESP_SPP_CL_INIT_EVT = 29
When SPP client initiated a connection, the event comes

ESP_SPP_DATA_IND_EVT = 30
When SPP connection received data, the event comes, olny for ESP_SPP_MODE_CB

ESP_SPP_CONG_EVT = 31
When SPP connection congestion status changed, the event comes, olny for ESP_SPP_MODE_CB

ESP_SPP_WRITE_EVT = 33
When SPP write operation completes, the event comes, olny for ESP_SPP_MODE_CB

ESP_SPP_SRV_OPEN_EVT = 34
When SPP Server connection open, the event comes

HFP DEFINES

Overview

Instructions

API Reference

Header File

- bt/bluedroid/api/include/api/esp_hf_defs.h

Enumerations

enum esp_hf_volume_control_target_t
Bluetooth HFP audio volume control target.

Values:

ESP_HF_VOLUME_CONTROL_TARGET_SPK = 0
speaker

ESP_HF_VOLUME_CONTROL_TARGET_MIC
microphone

enum esp_hf_roaming_status_t
+CIND roaming status indicator values

Values:

ESP_HF_ROAMING_STATUS_INACTIVE = 0
roaming is not active

ESP_HF_ROAMING_STATUS_ACTIVE
a roaming is active

enum esp_hf_call_status_t
+CIND call status indicator values

Values:

ESP_HF_CALL_STATUS_NO_CALLS = 0
no call in progress

ESP_HF_CALL_STATUS_CALL_IN_PROGRESS = 1
call is present(active or held)

enum esp_hf_call_setup_status_t
+CIND call setup status indicator values

Values:

ESP_HF_CALL_SETUP_STATUS_NONE = 0
no call setup in progress

ESP_HF_CALL_SETUP_STATUS_INCOMING = 1
incoming call setup in progress

ESP_HF_CALL_SETUP_STATUS_OUTGOING_DIALING = 2
outgoing call setup in dialing state

ESP_HF_CALL_SETUP_STATUS_OUTGOING_ALERTING = 3
outgoing call setup in alerting state

enum esp_hf_call_held_status_t
+CIND call held indicator values

Values:

ESP_HF_CALL_HELD_STATUS_NONE = 0
no calls held

ESP_HF_CALL_HELD_STATUS_HELD_AND_ACTIVE = 1
both active and held call

ESP_HF_CALL_HELD_STATUS_HELD = 2
call on hold, no active call

enum esp_hf_service_availability_status_t
+CIND network service availability status

Values:

ESP_HF_SERVICE_AVAILABILITY_STATUS_UNAVAILABLE = 0
service not available

ESP_HF_SERVICE_AVAILABILITY_STATUS_AVAILABLE
service available

enum esp_hf_current_call_status_t
+CLCC status of the call

Values:

ESP_HF_CURRENT_CALL_STATUS_ACTIVE = 0
active

ESP_HF_CURRENT_CALL_STATUS_HELD = 1
held

ESP_HF_CURRENT_CALL_STATUS_DIALING = 2
dialing (outgoing calls only)

ESP_HF_CURRENT_CALL_STATUS_ALERTING = 3
alerting (outgoing calls only)

ESP_HF_CURRENT_CALL_STATUS_INCOMING = 4
incoming (incoming calls only)

ESP_HF_CURRENT_CALL_STATUS_WAITING = 5
waiting (incoming calls only)

ESP_HF_CURRENT_CALL_STATUS_HELD_BY_RESP_HOLD = 6
call held by response and hold

enum esp_hf_current_call_direction_t
+CLCC direction of the call

Values:

ESP_HF_CURRENT_CALL_DIRECTION_OUTGOING = 0
outgoing

ESP_HF_CURRENT_CALL_DIRECTION_INCOMING = 1
incoming

enum esp_hf_current_call_mpty_type_t
+CLCC multi-party call flag

Values:

ESP_HF_CURRENT_CALL_MPTY_TYPE_SINGLE = 0
not a member of a multi-party call

ESP_HF_CURRENT_CALL_MPTY_TYPE_MULTI = 1
member of a multi-party call

enum esp_hf_current_call_mode_t
+CLCC call mode

Values:

ESP_HF_CURRENT_CALL_MODE_VOICE = 0

ESP_HF_CURRENT_CALL_MODE_DATA = 1

ESP_HF_CURRENT_CALL_MODE_FAX = 2

enum esp_hf_call_addr_type_t
+CLCC address type

Values:

ESP_HF_CALL_ADDR_TYPE_UNKNOWN = 0x81
unkown address type

ESP_HF_CALL_ADDR_TYPE_INTERNATIONAL = 0x91
international address

enum esp_hf_subscriber_service_type_t
+CNUM service type of the phone number

Values:

ESP_HF_SUBSCRIBER_SERVICE_TYPE_UNKNOWN = 0
unknown

ESP_HF_SUBSCRIBER_SERVICE_TYPE_VOICE
voice service

ESP_HF_SUBSCRIBER_SERVICE_TYPE_FAX
fax service

enum esp_hf_btrh_status_t
+BTRH response and hold result code

Values:

ESP_HF_BTRH_STATUS_HELD = 0
incoming call is put on held in AG

ESP_HF_BTRH_STATUS_ACCEPTED
held incoming call is accepted in AG

ESP_HF_BTRH_STATUS_REJECTED
held incoming call is rejected in AG

enum esp_hf_btrh_cmd_t
AT+BTRH response and hold action code.

Values:

ESP_HF_BTRH_CMD_HOLD = 0
put the incoming call on hold

ESP_HF_BTRH_CMD_ACCEPT = 1
accept a held incoming call

ESP_HF_BTRH_CMD_REJECT = 2
reject a held incoming call

enum esp_hf_at_response_code_t
response indication codes for AT commands

Values:

ESP_HF_AT_RESPONSE_CODE_OK = 0
acknoweledges execution of a command line

ESP_HF_AT_RESPONSE_CODE_ERR
command not accepted

ESP_HF_AT_RESPONSE_CODE_NO_CARRIER
connection terminated

ESP_HF_AT_RESPONSE_CODE_BUSY
busy signal detected

ESP_HF_AT_RESPONSE_CODE_NO_ANSWER
connection completion timeout

ESP_HF_AT_RESPONSE_CODE_DELAYED
delayed

ESP_HF_AT_RESPONSE_CODE_BLACKLISTED
blacklisted

ESP_HF_AT_RESPONSE_CODE_CME
CME error

enum esp_hf_vr_state_t
voice recognition state

Values:

ESP_HF_VR_STATE_DISABLED = 0
voice recognition disabled

ESP_HF_VR_STATE_ENABLED
voice recognition enabled

enum esp_hf_chld_type_t

AT+CHLD command values.

Values:

ESP_HF_CHLD_TYPE_REL = 0

<0>, Terminate all held or set UDUB("busy") to a waiting call

ESP_HF_CHLD_TYPE_REL_ACC

<1>, Terminate all active calls and accepts a waiting/held call

ESP_HF_CHLD_TYPE_HOLD_ACC

<2>, Hold all active calls and accepts a waiting/held call

ESP_HF_CHLD_TYPE_MERGE

<3>, Add all held calls to a conference

ESP_HF_CHLD_TYPE_MERGE_DETACH

<4>, connect the two calls and disconnects the subscriber from both calls

ESP_HF_CHLD_TYPE_REL_X

<1x>, releases specified calls only

ESP_HF_CHLD_TYPE_PRIV_X

<2x>, request private consultation mode with specified call

enum esp_hf_cme_err_t

Extended Audio Gateway Error Result Code Response.

Values:

ESP_HF_CME_AG_FAILURE = 0

ag failure

ESP_HF_CME_NO_CONNECTION_TO_PHONE = 1

no connection to phone

ESP_HF_CME_OPERATION_NOT_ALLOWED = 3

operation not allowed

ESP_HF_CME_OPERATION_NOT_SUPPORTED = 4

operation not supported

ESP_HF_CME_PH_SIM_PIN_REQUIRED = 5

PH-SIM PIN Required

ESP_HF_CME_SIM_NOT_INSERTED = 10

SIM not inserted

ESP_HF_CME_SIM_PIN_REQUIRED = 11

SIM PIN required

ESP_HF_CME_SIM_PUK_REQUIRED = 12

SIM PUK required

ESP_HF_CME_SIM_FAILURE = 13

SIM failure

ESP_HF_CME_SIM_BUSY = 14

SIM busy

ESP_HF_CME_INCORRECT_PASSWORD = 16

incorrect password

ESP_HF_CME_SIM_PIN2_REQUIRED = 17
SIM PIN2 required

ESP_HF_CME_SIM_PUK2_REQUIRED = 18
SIM PUK2 required

ESP_HF_CME_MEMEORY_FULL = 20
memory full

ESP_HF_CME_INVALID_INDEX = 21
invalid index

ESP_HF_CME_MEMEORY_FAILURE = 23
memory failure

ESP_HF_CME_TEXT_STRING_TOO_LONG = 24
test string too long

ESP_HF_CME_INVALID_CHARACTERS_IN_TEXT_STRING = 25
invalid characters in text string

ESP_HF_CME_DIAL_STRING_TOO_LONG = 26
dial string too long

ESP_HF_CME_INVALID_CHARACTERS_IN_DIAL_STRING = 27
invalid characters in dial string

ESP_HF_CME_NO_NETWORK_SERVICE = 30
no network service

ESP_HF_CME_NETWORK_TIMEOUT = 31
network timeout

ESP_HF_CME_NETWORK_NOT_ALLOWED = 32
network not allowed emergency calls only

HFP CLIENT API

Overview

Instructions

API Reference

Header File

- bt/bluedroid/api/include/api/esp_hf_client_api.h

Functions

esp_err_t **esp_hf_client_register_callback** (*esp_hf_client_cb_t callback*)
Register application callback function to HFP client module. This function should be called only after esp_bluedroid_enable() completes successfully, used by HFP client.

Return

- ESP_OK: success

- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

Parameters

- `callback`: HFP client event callback function

esp_err_t **esp_hf_client_init** (void)

Initialize the bluetooth HFP client module. This function should be called after esp_bluedroid_enable() completes successfully.

Return

- ESP_OK: if the initialization request is sent successfully
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_deinit** (void)

De-initialize for HFP client module. This function should be called only after esp_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_hf_client_connect** (*esp_bd_addr_t remote_bda*)

Connect to remote bluetooth HFP audio gateway(AG) device, must after esp_hf_client_init()

Return

- ESP_OK: connect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `remote_bda`: remote bluetooth device address

esp_err_t **esp_hf_client_disconnect** (*esp_bd_addr_t remote_bda*)

Disconnect from the remote HFP audio gateway.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `remote_bda`: remote bluetooth device address

esp_err_t **esp_hf_client_connect_audio**(*esp_bd_addr_t remote_bda*)

Create audio connection with remote HFP AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `remote_bda`: remote bluetooth device address

esp_err_t **esp_hf_client_disconnect_audio**(*esp_bd_addr_t remote_bda*)

Release the established audio connection with remote HFP AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `remote_bda`: remote bluetooth device address

esp_err_t **esp_hf_client_start_voice_recognition**(void)

Enable voice recognition in the AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_stop_voice_recognition**(void)

Disable voice recognition in the AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_volume_update**(*esp_hf_volume_control_target_t type*, int *volume*)

Volume synchronization with AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer

- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `type`: volume control target, speaker or microphone
- `volume`: gain of the speaker of microphone, ranges 0 to 15

esp_err_t **esp_hf_client_dial**(**const** char **number*)

Place a call with a specified number, if number is NULL, last called number is called. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `number`: number string of the call. If NULL, the last number is called(aka re-dial)

esp_err_t **esp_hf_client_dial_memory**(int *location*)

Place a call with number specified by location(speed dial). As a precondition, to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `location`: location of the number in the memory

esp_err_t **esp_hf_client_send_chld_cmd**(*esp_hf_chld_type_t chld*, int *idx*)

Send call hold and multiparty commands, or enhanced call control commands(Use AT+CHLD). As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `chld`: AT+CHLD call hold and multiparty handling AT command.
- `idx`: used in Enhanced Call Control Mechanisms, used if chld is ESP_HF_CHLD_TYPE_REL_X or ESP_HF_CHLD_TYPE_PRIV_X

esp_err_t **esp_hf_client_send_btrh_cmd**(*esp_hf_btrh_cmd_t btrh*)

Send response and hold action command(Send AT+BTRH command) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `btrh`: response and hold action to send

esp_err_t **esp_hf_client_answer_call** (void)

Answer an incoming call(send ATA command). As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_reject_call** (void)

Reject an incoming call(send AT+CHUP command), As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_query_current_calls** (void)

Query list of current calls in AG(send AT+CLCC command), As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_query_current_operator_name** (void)

Query the name of currently selected network operator in AG(use AT+COPS commands) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_retrieve_subscriber_info** (void)

Get subscriber information number from AG(send AT+CNUM command) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_send_dtmf** (char *code*)

Transmit DTMF codes during an ongoing call(use AT+VTS commands) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `code`: dtmf code, single ascii character in the set 0-9, #, *, A-D

esp_err_t **esp_hf_client_request_last_voice_tag_number** (void)

Request a phone number from AG corresponding to last voice tag recorded (send AT+BINP command). As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_register_data_callback** (*esp_hf_client_incoming_data_cb_t* *recv*, *esp_hf_client_outgoing_data_cb_t* *send*)

Register HFP client data output function; the callback is only used in the case that Voice Over HCI is enabled.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

Parameters

- `recv`: HFP client incoming data callback function
- `send`: HFP client outgoing data callback function

void **esp_hf_client_outgoing_data_ready** (void)

Trigger the lower-layer to fetch and send audio data. This function is only only used in the case that Voice Over HCI is enabled. Precondition is that the HFP audio connection is connected. After this function is called, lower layer will invoke esp_hf_client_outgoing_data_cb_t to fetch data.

void **esp_hf_client_pcm_resample_init** (uint32_t *src_sps*, uint32_t *bits*, uint32_t *channels*)

Initialize the down sampling converter. This is a utility function that can only be used in the case that Voice Over HCI is enabled.

Parameters

- `src_sps`: original samples per second(source audio data, i.e. 48000, 32000, 16000, 44100, 22050, 11025)
- `bits`: number of bits per pcm sample (16)
- `channels`: number of channels (i.e. mono(1), stereo(2)...)

int32_t **esp_hf_client_pcm_resample** (void **src*, uint32_t *in_bytes*, void **dst*)

Down sampling utility to convert high sampling rate into 8K/16bits 1-channel mode PCM samples. This can only be used in the case that Voice Over HCI is enabled.

Return number of samples converted

Parameters

- `src`: pointer to the buffer where the original smapling PCM are stored
- `in_bytes`: length of the input PCM sample buffer in byte
- `dst`: pointer to the buffer which is to be used to store the converted PCM samples

Unions

union esp_hf_client_cb_param_t

#include <esp_hf_client_api.h> HFP client callback parameters.

Public Members

struct *esp_hf_client_cb_param_t*::*hf_client_conn_stat_param* **conn_stat**

HF callback param of ESP_HF_CLIENT_CONNECTION_STATE_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_audio_stat_param* **audio_stat**

HF callback param of ESP_HF_CLIENT_AUDIO_STATE_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_bvra_param* **bvra**

HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_service_availability_param* **service_availability**

HF callback param of ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_network_roaming_param* **roaming**

HF callback param of ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_signal_strength_ind_param* **signal_strength**

HF callback param of ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_battery_level_ind_param* **battery_level**

HF callback param of ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_current_operator_param* **cops**

HF callback param of ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_call_ind_param* **call**

HF callback param of ESP_HF_CLIENT_CIND_CALL_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_call_setup_ind_param* **call_setup**
HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_call_held_ind_param* **call_held**
HF callback param of ESP_HF_CLIENT_CIND_CALL_HELD_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_btrh_param* **btrh**
HF callback param of ESP_HF_CLIENT_BRTH_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_clip_param* **clip**
HF callback param of ESP_HF_CLIENT_CLIP_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_ccwa_param* **ccwa**
HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_clcc_param* **clcc**
HF callback param of ESP_HF_CLIENT_CLCC_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_volume_control_param* **volume_control**
HF callback param of ESP_HF_CLIENT_VOLUME_CONTROL_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_at_response_param* **at_response**
HF callback param of ESP_HF_CLIENT_AT_RESPONSE_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_cnum_param* **cnum**
HF callback param of ESP_HF_CLIENT_CNUM_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_bsirparam* **bsir**
HF callback param of ESP_HF_CLIENT_BSIR_EVT

struct *esp_hf_client_cb_param_t*::*hf_client_binp_param* **binp**
HF callback param of ESP_HF_CLIENT_BINP_EVT

struct hf_client_at_response_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_AT_RESPONSE_EVT.

Public Members

esp_hf_at_response_code_t **code**
AT response code

esp_hf_cme_err_t **cme**
Extended Audio Gateway Error Result Code

struct hf_client_audio_stat_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_AUDIO_STATE_EVT.

Public Members

esp_hf_client_audio_state_t **state**
audio connection state

esp_bd_addr_t **remote_bda**
remote bluetooth device address

struct hf_client_battery_level_ind_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT.

Public Members

int **value**
battery charge value, ranges from 0 to 5

struct hf_client_binp_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_BINP_EVT.

Public Members

const char ***number**
phone number corresponding to the last voice tag in the HF

struct hf_client_bsirparam
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_BSIR_EVT.

Public Members

esp_hf_client_in_band_ring_state_t **state**
setting state of in-band ring tone

struct hf_client_btrh_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_BTRH_EVT.

Public Members

esp_hf_btrh_status_t **status**
call hold and response status result code

struct hf_client_bvra_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_BVRA_EVT.

Public Members

esp_hf_vr_state_t **value**
voice recognition state

struct hf_client_call_held_ind_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_CIND_CALL_HELD_EVT.

Public Members

esp_hf_call_held_status_t **status**
bluetooth proprietary call hold status indocator

struct hf_client_call_ind_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_CIND_CALL_EVT.

Public Members

esp_hf_call_status_t **status**
call status indicator

struct hf_client_call_setup_ind_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_CIND_CALL_SETUP_EVT.

Public Members

esp_hf_call_setup_status_t **status**
call setup status indicator

struct hf_client_ccwa_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_CCWA_EVT.

Public Members

const char ***number**
phone number string of waiting call

struct hf_client_clcc_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_CLCC_EVT.

Public Members

int **idx**
numbering(starting with 1) of the call

esp_hf_current_call_direction_t **dir**
direction of the call

esp_hf_current_call_status_t **status**
status of the call

esp_hf_current_call_mpty_type_t **mpty**
multi-party flag

char ***number**
phone number(optional)

struct hf_client_clip_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_CLIP_EVT.

Public Members

const char ***number**
phone number string of call

struct hf_client_cnum_param
#include <*esp_hf_client_api.h*> ESP_HF_CLIENT_CNUM_EVT.

Public Members

const char ***number**
phone number string

esp_hf_subscriber_service_type_t **type**
service type that the phone number relates to

struct hf_client_conn_stat_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CONNECTION_STATE_EVT.

Public Members

esp_hf_client_connection_state_t **state**
HF connection state

uint32_t **peer_feat**
AG supported features

uint32_t **chld_feat**
AG supported features on call hold and multiparty services

esp_bd_addr_t **remote_bda**
remote bluetooth device address

struct hf_client_current_operator_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT.

Public Members

const char ***name**
name of the network operator

struct hf_client_network_roaming_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT.

Public Members

esp_hf_roaming_status_t **status**
roaming status

struct hf_client_service_availability_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT.

Public Members

esp_hf_service_availability_status_t **status**
service availability status

struct hf_client_signal_strength_ind_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT.

Public Members

int **value**
singal strength value, ranges from 0 to 5

struct hf_client_volume_control_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_VOLUME_CONTROL_EVT.

Public Members

esp_hf_volume_control_target_t **type**
volume control target, speaker or microphone

int **volume**
gain, ranges from 0 to 15

Macros

ESP_BT_HF_CLIENT_NUMBER_LEN

ESP_BT_HF_CLIENT_OPERATOR_NAME_LEN

ESP_HF_CLIENT_PEER_FEAT_3WAY

ESP_HF_CLIENT_PEER_FEAT_ECNR

ESP_HF_CLIENT_PEER_FEAT_VREC

ESP_HF_CLIENT_PEER_FEAT_INBAND

ESP_HF_CLIENT_PEER_FEAT_VTAG

ESP_HF_CLIENT_PEER_FEAT_REJECT

ESP_HF_CLIENT_PEER_FEAT_ECS

ESP_HF_CLIENT_PEER_FEAT_ECC

ESP_HF_CLIENT_PEER_FEAT_EXTERR

ESP_HF_CLIENT_PEER_FEAT_CODEC

ESP_HF_CLIENT_CHLD_FEAT_REL

ESP_HF_CLIENT_CHLD_FEAT_REL_ACC

ESP_HF_CLIENT_CHLD_FEAT_REL_X

ESP_HF_CLIENT_CHLD_FEAT_HOLD_ACC

ESP_HF_CLIENT_CHLD_FEAT_PRIV_X

ESP_HF_CLIENT_CHLD_FEAT_MERGE

ESP_HF_CLIENT_CHLD_FEAT_MERGE_DETACH

Type Definitions

typedef void (*__esp_hf_client_incoming_data_cb_t__)(**const** uint8_t *buf, uint32_t len)
HFP client incoming data callback function, the callback is useful in case of Voice Over HCI.

Parameters

- `buf`: : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.
- `len`: : size(in bytes) in buf

typedef uint32_t (*__esp_hf_client_outgoing_data_cb_t__) (uint8_t *buf, uint32_t len)

HFP client outgoing data callback function, the callback is useful in case of Voice Over HCI. Once audio connection is set up and the application layer has prepared data to send, the lower layer will call this function to read data and then send. This callback is supposed to be implemented as non-blocking, and if data is not enough, return value 0 is supposed.

Parameters

- `buf`: : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.
- `len`: : size(in bytes) in buf
- `length`: of data successfully read

typedef void (*__esp_hf_client_cb_t__) (*esp_hf_client_cb_event_t* event, *esp_hf_client_cb_param_t* *param)

HFP client callback function type.

Parameters

- `event`: : Event type
- `param`: : Pointer to callback parameter

Enumerations

enum esp_hf_client_connection_state_t

Bluetooth HFP RFCOMM connection and service level connection status.

Values:

ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTED = 0

RFCOMM data link channel released

ESP_HF_CLIENT_CONNECTION_STATE_CONNECTING

connecting remote device on the RFCOMM data link

ESP_HF_CLIENT_CONNECTION_STATE_CONNECTED

RFCOMM connection established

ESP_HF_CLIENT_CONNECTION_STATE_SLC_CONNECTED

service level connection established

ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTING

disconnecting with remote device on the RFCOMM dat link

enum esp_hf_client_audio_state_t

Bluetooth HFP audio connection status.

Values:

ESP_HF_CLIENT_AUDIO_STATE_DISCONNECTED = 0

audio connection released

ESP_HF_CLIENT_AUDIO_STATE_CONNECTING
audio connection has been initiated

ESP_HF_CLIENT_AUDIO_STATE_CONNECTED
audio connection is established

ESP_HF_CLIENT_AUDIO_STATE_CONNECTED_MSBC
mSBC audio connection is estalibshed

enum esp_hf_client_in_band_ring_state_t
in-band ring tone state

Values:

ESP_HF_CLIENT_IN_BAND_RINGTONE_NOT_PROVIDED = 0

ESP_HF_CLIENT_IN_BAND_RINGTONE_PROVIDED

enum esp_hf_client_cb_event_t
HF CLIENT callback events.

Values:

ESP_HF_CLIENT_CONNECTION_STATE_EVT = 0
connection state changed event

ESP_HF_CLIENT_AUDIO_STATE_EVT
audio connection state change event

ESP_HF_CLIENT_BVRA_EVT
voice recognition state change event

ESP_HF_CLIENT_CIND_CALL_EVT
call indication

ESP_HF_CLIENT_CIND_CALL_SETUP_EVT
call setup indication

ESP_HF_CLIENT_CIND_CALL_HELD_EVT
call held indication

ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT
network service availability indication

ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT
signal strength indication

ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT
roaming status indication

ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT
battery level indication

ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT
current operator information

ESP_HF_CLIENT_BTRH_EVT
call response and hold event

ESP_HF_CLIENT_CLIP_EVT
Calling Line Identification notification

ESP_HF_CLIENT_CCWA_EVT
call waiting notification

ESP_HF_CLIENT_CLCC_EVT
list of current calls notification

ESP_HF_CLIENT_VOLUME_CONTROL_EVT
audio volume control command from AG, provided by +VGM or +VGS message

ESP_HF_CLIENT_AT_RESPONSE_EVT
AT command response event

ESP_HF_CLIENT_CNUM_EVT
subscriber information response from AG

ESP_HF_CLIENT_BSIR_EVT
setting of in-band ring tone

ESP_HF_CLIENT_BINP_EVT
requested number of last voice tag from AG

ESP_HF_CLIENT_RING_IND_EVT
ring indication event

To see the overview of the ESP32 Bluetooth stack architecture, follow links below:

- ESP32 Bluetooth Architecture (PDF) [English]
- ESP32 Bluetooth Architecture (PDF) []

Example code for this API section is provided in bluetooth directory of ESP-IDF examples.

Several examples contain detailed description. To see them please follow links below:

- GATT Client Example Walkthrough
- GATT Server Service Table Example Walkthrough
- GATT Server Example Walkthrough
- GATT Security Client Example Walkthrough
- GATT Security Server Example Walkthrough
- GATT Client Multi-connection Example Walkthrough

3.4 Ethernet API

3.4.1 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 overriden 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 recevie 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_FULLDUPLEX

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.

3.5 Peripherals API

3.5.1 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 (8 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 sesnor.

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;
```

(continues on next page)

(continued from previous page)

```
    adc2_config_channel_atten( ADC2_CHANNEL_7, ADC_ATTEN_0db );

    esp_err_t r = adc2_get_raw( ADC2_CHANNEL_7, ADC_WIDTH_12Bit, &read_raw);
    if ( r == ESP_OK ) {
        printf("%d\n", read_raw );
    } else if ( r == ESP_ERR_TIMEOUT ) {
        printf("ADC2 used by Wi-Fi.\n");
    }
```

The reading may fail due to collision with Wi-Fi, should check it. An example using the ADC2 driver to read the output of DAC is available in esp-idf: peripherals/adc2

Reading the internal hall effect sensor:

```
#include <driver/adc.h>

...

    adc1_config_width(ADC_WIDTH_BIT_12);
    int val = hall_sensor_read();
```

The value read in both these examples is 12 bits wide (range 0-4095).

Minimizing Noise

The ESP32 ADC can be sensitive to noise leading to large discrepancies in ADC readings. To minimize noise, users may connect a 0.1uF capacitor to the ADC input pad in use. Multisampling may also be used to further mitigate the effects of noise.

ADC Calibration

The esp_adc_cal/include/esp_adc_cal.h API provides functions to correct for differences in measured voltages caused by variation of ADC reference voltages (Vref) 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 = coeff_a * x + 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.

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.

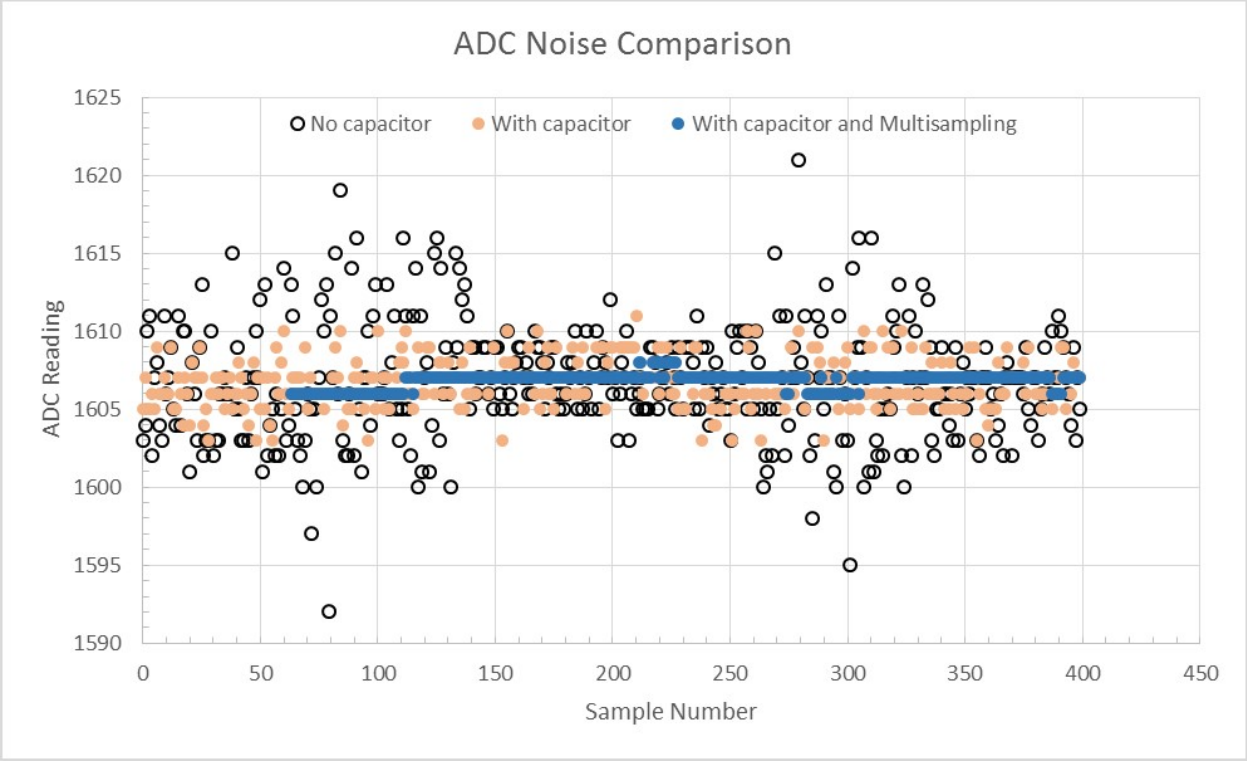


Fig. 3: Graph illustrating noise mitigation using capacitor and multisampling of 64 samples.

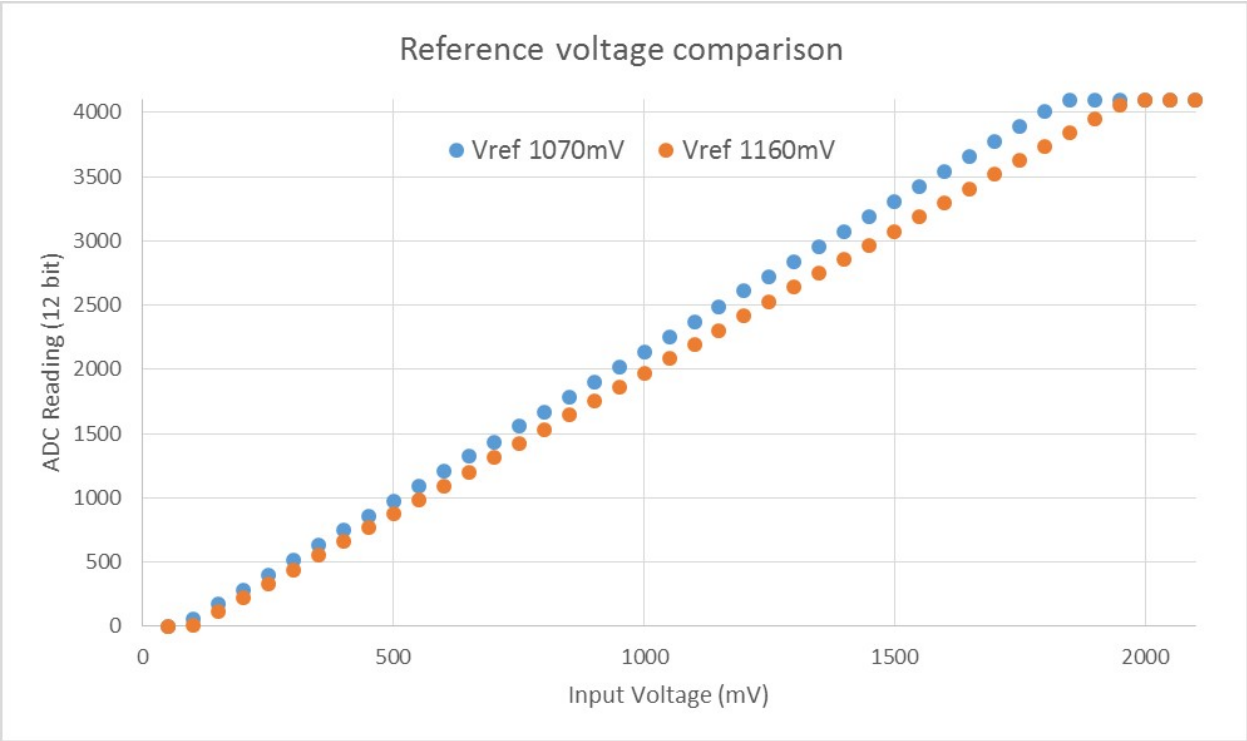


Fig. 4: Graph illustrating effect of differing reference voltages on the ADC voltage curve.

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));
    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 channal not valid

Parameters

- `channel`: Channel to get the gpio number
- `gpio_num`: output buffer to hold the gpio number

esp_err_t **adc1_config_width**(*adc_bits_width_t width_bit*)

Configure ADC1 capture width, meanwhile enable output invert for ADC1. The configuration is for all channels of ADC1.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `width_bit`: Bit capture width for ADC1

esp_err_t **adc_set_data_width**(*adc_unit_t adc_unit*, *adc_bits_width_t width_bit*)

Configure ADC capture width.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `adc_unit`: ADC unit index

- `width_bit`: Bit capture width for ADC unit.

esp_err_t **adc1_config_channel_atten**(*adc1_channel_t channel*, *adc_atten_t atten*)

Set the attenuation of a particular channel on ADC1, and configure its associated GPIO pin mux.

The default ADC full-scale voltage is 1.1V. To read higher voltages (up to the pin maximum voltage, usually 3.3V) requires setting >0dB signal attenuation for that ADC channel.

Note For any given channel, this function must be called before the first time adc1_get_raw() is called for that channel.

Note This function can be called multiple times to configure multiple ADC channels simultaneously. adc1_get_raw() can then be called for any configured channel.

When VDD_A is 3.3V:

- 0dB attenuaton (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)

Due to ADC characteristics, most accurate results are obtained within the following approximate voltage ranges:

Note The full-scale voltage is the voltage corresponding to a maximum reading (depending on ADC1 configured bit width, this value is: 4095 for 12-bits, 2047 for 11-bits, 1023 for 10-bits, 511 for 9 bits.)

Note At 11dB attenuation the maximum voltage is limited by VDD_A, not the full scale voltage.

- 0dB attenuaton (ADC_ATTEN_DB_0) between 100 and 950mV
- 2.5dB attenuation (ADC_ATTEN_DB_2_5) between 100 and 1250mV
- 6dB attenuation (ADC_ATTEN_DB_6) between 150 to 1750mV
- 11dB attenuation (ADC_ATTEN_DB_11) between 150 to 2450mV

For maximum accuracy, use the ADC calibration APIs and measure voltages within these recommended ranges.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel`: ADC1 channel to configure
- `atten`: Attenuation level

int **adc1_get_raw**(*adc1_channel_t channel*)

Take an ADC1 reading from a single channel.

Note When the power switch of SARADC1, SARADC2, HALL sensor and AMP sensor is turned on, the input of GPIO36 and GPIO39 will be pulled down for about 80ns. When enabling power for any of these peripherals, ignore input from GPIO36 and GPIO39. Please refer to section 3.11 of 'ECO_and_Workarounds_for_Bugs_in_ESP32' for the description of this issue.

Note Call adc1_config_width() before the first time this function is called.

Note For any given channel, adc1_config_channel_atten(channel) must be called before the first time this function is called. Configuring a new channel does not prevent a previously configured channel from being read.

Return

- -1: Parameter error
- Other: ADC1 channel reading.

Parameters

- `channel`: ADC1 channel to read

void **adc_power_on**()
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 When the power switch of SARADC1, SARADC2, HALL sensor and AMP sensor is turned on, the input of GPIO36 and GPIO39 will be pulled down for about 80ns. When enabling power for any of these peripherals, ignore input from GPIO36 and GPIO39. Please refer to section 3.11 of 'ECO_and_Workarounds_for_Bugs_in_ESP32' for the description of this issue.

Note The Hall Sensor uses channels 0 and 3 of ADC1. Do not configure these channels for use as ADC channels.

Note The ADC1 module must be enabled by calling adc1_config_width() before calling hall_sensor_read(). ADC1 should be configured for 12 bit readings, as the hall sensor readings are low values and do not cover the full range of the ADC.

Return The hall sensor reading.

esp_err_t **adc2_pad_get_io_num**(*adc2_channel_t channel*, *gpio_num_t *gpio_num*)

Get the gpio number of a specific ADC2 channel.

Return

- ESP_OK if success
- ESP_ERR_INVALID_ARG if channal 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 attenuaton (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 When the power switch of SARADC1, SARADC2, HALL sensor and AMP sensor is turned on, the input of GPIO36 and GPIO39 will be pulled down for about 80ns. When enabling power for any of these peripherals, ignore input from GPIO36 and GPIO39. Please refer to section 3.11 of 'ECO_and_Workarounds_for_Bugs_in_ESP32' for the description of this issue.

Note For a given channel, `adc2_config_channel_atten()` must be called before the first time this function is called. If Wi-Fi is started via `esp_wifi_start()`, this function will always fail with `ESP_ERR_TIMEOUT`.

Return

- ESP_OK if success
- ESP_ERR_TIMEOUT the WIFI is started, using the ADC2

Parameters

- `channel`: ADC2 channel to read
- `width_bit`: Bit capture width for ADC2
- `raw_out`: the variable to hold the output data.

esp_err_t **adc2_vref_to_gpio**(*gpio_num_t gpio*)

Output ADC2 reference voltage to gpio 25 or 26 or 27.

This function utilizes the testing mux exclusive to ADC 2 to route the reference voltage one of ADC2's channels. Supported gpios are gpios 25, 26, and 27. This refernce 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 = coeff_a * x + 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

3.5.2 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 channal 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

3.5.3 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_reset_pin**(*gpio_num_t gpio_num*)

Reset an gpio to default state (select gpio function, enable pullup and disable input and output).

Note This function also configures the IOMUX for this pin to the GPIO function, and disconnects any other peripheral output configured via GPIO Matrix.

Return Always return ESP_OK.

Parameters

- `gpio_num`: GPIO number.

esp_err_t **gpio_set_intr_type**(*gpio_num_t gpio_num*, *gpio_int_type_t intr_type*)

GPIO set interrupt trigger type.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to set the trigger type of e.g. of GPIO16, gpio_num should be GPIO_NUM_16 (16);
- `intr_type`: Interrupt type, select from gpio_int_type_t

esp_err_t **gpio_intr_enable**(*gpio_num_t gpio_num*)

Enable GPIO module interrupt signal.

Note Please do not use the interrupt of GPIO36 and GPIO39 when using ADC. Please refer to the comments of `adc1_get_raw`. Please refer to section 3.11 of 'ECO_and_Workarounds_for_Bugs_in_ESP32' for the description of this issue.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to enable an interrupt on e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);

esp_err_t **gpio_intr_disable**(*gpio_num_t gpio_num*)

Disable GPIO module interrupt signal.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to disable the interrupt of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);

esp_err_t **gpio_set_level** (*gpio_num_t gpio_num*, uint32_t *level*)
GPIO set output level.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO number error

Parameters

- `gpio_num`: GPIO number. If you want to set the output level of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
- `level`: Output level. 0: low ; 1: high

int **gpio_get_level** (*gpio_num_t gpio_num*)
GPIO get input level.

Warning If the pad is not configured for input (or input and output) the returned value is always 0.

Return

- 0 the GPIO input level is 0
- 1 the GPIO input level is 1

Parameters

- `gpio_num`: GPIO number. If you want to get the logic level of e.g. pin GPIO16, gpio_num should be GPIO_NUM_16 (16);

esp_err_t **gpio_set_direction** (*gpio_num_t gpio_num*, *gpio_mode_t mode*)
GPIO set direction.

Configure GPIO direction,such as output_only,input_only,output_and_input

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO error

Parameters

- `gpio_num`: Configure GPIO pins number, it should be GPIO number. If you want to set direction of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
- `mode`: GPIO direction

esp_err_t **gpio_set_pull_mode** (*gpio_num_t gpio_num*, *gpio_pull_mode_t pull*)
Configure GPIO pull-up/pull-down resistors.

Only pins that support both input & output have integrated pull-up and pull-down resistors. Input-only GPIOs 34-39 do not.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG : Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to set pull up or down mode for e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
- `pull`: GPIO pull up/down mode.

esp_err_t **gpio_wakeup_enable**(*gpio_num_t gpio_num*, *gpio_int_type_t intr_type*)

Enable GPIO wake-up function.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number.
- `intr_type`: GPIO wake-up type. Only GPIO_INTR_LOW_LEVEL or GPIO_INTR_HIGH_LEVEL can be used.

esp_err_t **gpio_wakeup_disable**(*gpio_num_t gpio_num*)

Disable GPIO wake-up function.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number

esp_err_t **gpio_isr_register**(void (**fn*)) void *

, void **arg*, int *intr_alloc_flags*, *gpio_isr_handle_t* **handle*Register GPIO interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

This ISR function is called whenever any GPIO interrupt occurs. See the alternative gpio_install_isr_service() and gpio_isr_handler_add() API in order to have the driver support per-GPIO ISRs.

To disable or remove the ISR, pass the returned handle to the *interrupt allocation functions*.

Parameters

- `fn`: Interrupt handler function.
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.
- `arg`: Parameter for handler function
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

Return

- ESP_OK Success ;
- ESP_ERR_INVALID_ARG GPIO error
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags

esp_err_t **gpio_pullup_en**(*gpio_num_t gpio_num*)

Enable pull-up on GPIO.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number

esp_err_t **gpio_pullup_dis** (*gpio_num_t gpio_num*)
Disable pull-up on GPIO.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number

esp_err_t **gpio_pulldown_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 **gpio_pulldown_dis** (*gpio_num_t gpio_num*)
Disable pull-down on GPIO.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number

esp_err_t **gpio_install_isr_service** (int *intr_alloc_flags*)
Install the driver's GPIO ISR handler service, which allows per-pin GPIO interrupt handlers.

This function is incompatible with gpio_isr_register() - if that function is used, a single global ISR is registered for all GPIO interrupts. If this function is used, the ISR service provides a global GPIO ISR and individual pin handlers are registered via the gpio_isr_handler_add() function.

Return

- ESP_OK Success
- ESP_ERR_NO_MEM No memory to install this service
- ESP_ERR_INVALID_STATE ISR service already installed.

- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags
- ESP_ERR_INVALID_ARG GPIO error

Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.

void **gpio_uninstall_isr_service**()

Uninstall the driver's GPIO ISR service, freeing related resources.

esp_err_t **gpio_isr_handler_add**(*gpio_num_t gpio_num*, *gpio_isr_t isr_handler*, void **args*)

Add ISR handler for the corresponding GPIO pin.

Call this function after using gpio_install_isr_service() to install the driver's GPIO ISR handler service.

The pin ISR handlers no longer need to be declared with IRAM_ATTR, unless you pass the ESP_INTR_FLAG_IRAM flag when allocating the ISR in gpio_install_isr_service().

This ISR handler will be called from an ISR. So there is a stack size limit (configurable as "ISR stack size" in menuconfig). This limit is smaller compared to a global GPIO interrupt handler due to the additional level of indirection.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number
- `isr_handler`: ISR handler function for the corresponding GPIO number.
- `args`: parameter for ISR handler.

esp_err_t **gpio_isr_handler_remove**(*gpio_num_t gpio_num*)

Remove ISR handler for the corresponding GPIO pin.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number

esp_err_t **gpio_set_drive_capability**(*gpio_num_t gpio_num*, *gpio_drive_cap_t strength*)

Set GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Drive capability of the pad

esp_err_t **gpio_get_drive_capability**(*gpio_num_t gpio_num*, *gpio_drive_cap_t *strength*)

Get GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Pointer to accept drive capability of the pad

esp_err_t **gpio_hold_en**(*gpio_num_t gpio_num*)

Enable gpio pad hold function.

The gpio pad hold function works in both input and output modes, but must be output-capable gpios. If pad hold enabled: in output mode: the output level of the pad will be force locked and can not be changed. in input mode: the input value read will not change, regardless the changes of input signal.

The state of digital gpio cannot be held during Deep-sleep, and it will resume the hold function when the chip wakes up from Deep-sleep. If the digital gpio also needs to be held during Deep-sleep, `gpio_deep_sleep_hold_en` should also be called.

Power down or call gpio_hold_dis will disable this function.

Return

- ESP_OK Success
- ESP_ERR_NOT_SUPPORTED Not support pad hold function

Parameters

- `gpio_num`: GPIO number, only support output-capable GPIOs

esp_err_t **gpio_hold_dis**(*gpio_num_t gpio_num*)

Disable gpio pad hold function.

When the chip is woken up from Deep-sleep, the gpio will be set to the default mode, so, the gpio will output the default level if this function is called. If you dont't want the level changes, the gpio should be configured to a known state before this function is called. e.g. If you hold gpio18 high during Deep-sleep, after the chip is woken up and `gpio_hold_dis` is called, gpio18 will output low level(because gpio18 is input mode by default). If you don't want this behavior, you should configure gpio18 as output mode and set it to hight level before calling `gpio_hold_dis`.

Return

- ESP_OK Success
- ESP_ERR_NOT_SUPPORTED Not support pad hold function

Parameters

- `gpio_num`: GPIO number, only support output-capable GPIOs

void **gpio_deep_sleep_hold_en** (void)
Enable all digital gpio pad hold function during Deep-sleep.

When the chip is in Deep-sleep mode, all digital gpio will hold the state before sleep, and when the chip is woken up, the status of digital gpio will not be held. Note that the pad hold feature only works when the chip is in Deep-sleep mode, when not in sleep mode, the digital gpio state can be changed even you have called this function.

Power down or call gpio_hold_dis will disable this function, otherwise, the digital gpio hold feature works as long as the chip enter Deep-sleep.

void **gpio_deep_sleep_hold_dis** (void)
Disable all digital gpio pad hold function during Deep-sleep.

void **gpio_iomux_in** (uint32_t *gpio_num*, uint32_t *signal_idx*)
Set pad input to a peripheral signal through the IOMUX.

Parameters

- `gpio_num`: GPIO number of the pad.
- `signal_idx`: Peripheral signal id to input. One of the `*_IN_IDX` signals in `soc/gpio_sig_map.h`.

void **gpio_iomux_out** (uint8_t *gpio_num*, int *func*, bool *oen_inv*)
Set peripheral output to an GPIO pad through the IOMUX.

Parameters

- `gpio_num`: gpio_num GPIO number of the pad.
- `func`: The function number of the peripheral pin to output pin. One of the `FUNC_X_*` of specified pin (X) in `soc/io_mux_reg.h`.
- `oen_inv`: True if the output enable needs to be inversed, otherwise False.

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_O
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. talse 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_gpio_pulldown_dis** (*gpio_num_t gpio_num*)

RTC GPIO pulldown disable.

This function only works for RTC IOs. In general, call gpio_pulldown_dis, which will work both for normal GPIOs and RTC IOs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- `gpio_num`: GPIO number (e.g. GPIO_NUM_12)

esp_err_t **rtc_gpio_hold_en** (*gpio_num_t gpio_num*)

Enable hold function on an RTC IO pad.

Enabling HOLD function will cause the pad to latch current values of input enable, output enable, output value, function, drive strength values. This function is useful when going into light or deep sleep mode to prevent the pin configuration from changing.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- `gpio_num`: GPIO number (e.g. GPIO_NUM_12)

esp_err_t **rtc_gpio_hold_dis** (*gpio_num_t gpio_num*)

Disable hold function on an RTC IO pad.

Disabling hold function will allow the pad receive the values of input enable, output enable, output value, function, drive strength from RTC_IO peripheral.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- `gpio_num`: GPIO number (e.g. GPIO_NUM_12)

esp_err_t **rtc_gpio_isolate** (*gpio_num_t gpio_num*)

Helper function to disconnect internal circuits from an RTC IO This function disables input, output, pullup, pulldown, and enables hold feature for an RTC IO. Use this function if an RTC IO needs to be disconnected from internal circuits in deep sleep, to minimize leakage current.

In particular, for ESP32-WROVER module, call rtc_gpio_isolate(GPIO_NUM_12) before entering deep sleep, to reduce deep sleep current.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if GPIO is not an RTC IO

Parameters

- `gpio_num`: GPIO number (e.g. GPIO_NUM_12).

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

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

3.5.4 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 made 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 commend link for I2C master sending n bytes to slave is shown below:

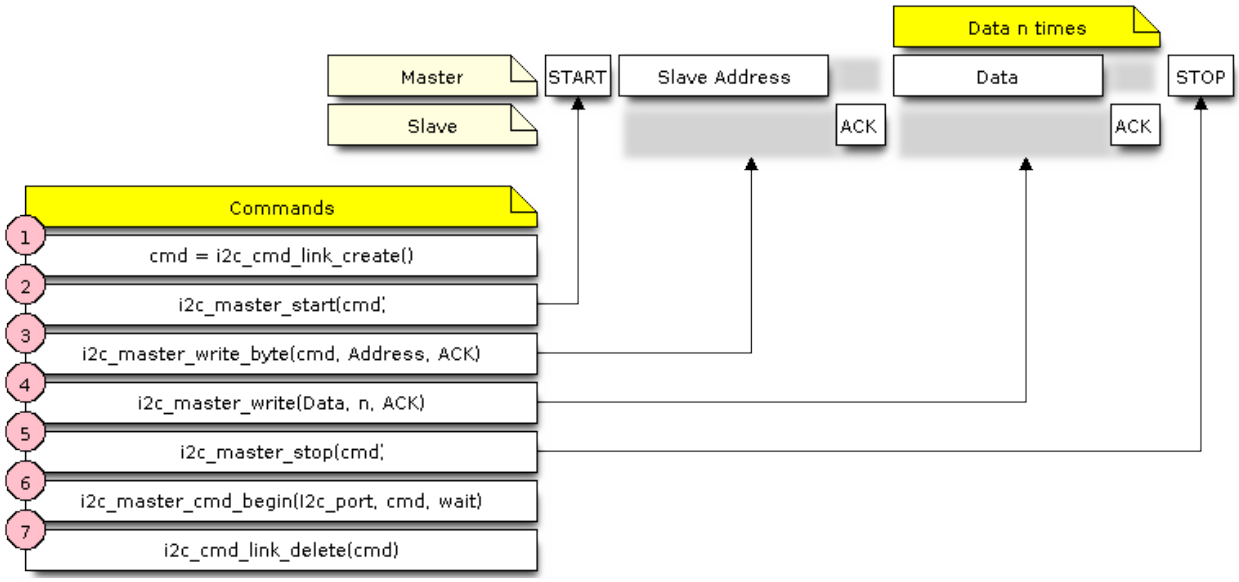


Fig. 5: I2C command link - master write example

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()`.
7. 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.

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:

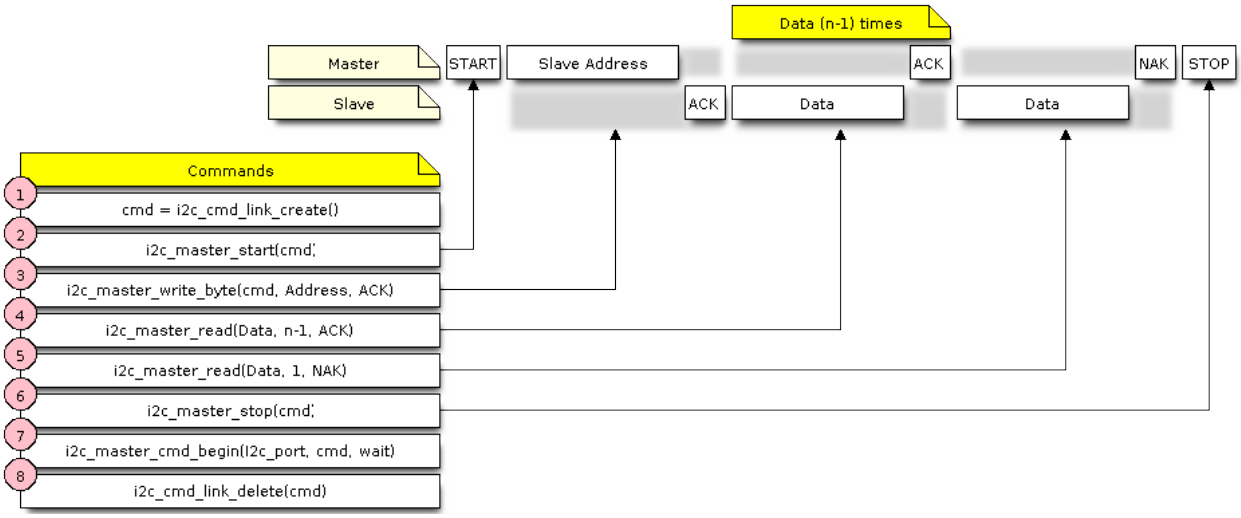


Fig. 6: I2C command link - master read example

```
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<= cyc_num <=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 receving 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

3.5.5 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 = true* and ESP32 will automatically calculate APLL parameter.

Note: If *use_apll = true* and *fixed_mclk > 0*, then the Master clock output for I2S is fixed and equal to the fixed_mclk value. The audio clock rate (LRCK) is always the MCLK divisor and 0 < MCLK/LRCK/channels/bits_per_sample <

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 = false
};

static const i2s_pin_config_t pin_config = {
    .bck_io_num = 26,
    .ws_io_num = 25,
    .data_out_num = 22,
    .data_in_num = I2S_PIN_NO_CHANGE
};

...

    i2s_driver_install(i2s_num, &i2s_config, 0, NULL);   //install and start i2s␣
↪driver

    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 = {
     .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,
```

(continues on next page)

(continued from previous page)

```
    .intr_alloc_flags = 0, // default interrupt priority
    .dma_buf_count = 8,
    .dma_buf_len = 64,
    .use_apll = false
};

...

    i2s_driver_install(i2s_num, &i2s_config, 0, NULL);   //install and start i2s␣
↪driver

    i2s_set_pin(i2s_num, NULL); //for internal DAC, this will enable both of the␣
↪internal channels

    //You can call i2s_set_dac_mode to set built-in DAC output mode.
    //i2s_set_dac_mode(I2S_DAC_CHANNEL_BOTH_EN);

    i2s_set_sample_rates(i2s_num, 22050); //set sample rates

    i2s_driver_uninstall(i2s_num); //stop & destroy i2s driver
```

API Reference

Header File

- driver/include/driver/i2s.h

Functions

esp_err_t **i2s_set_pin**(*i2s_port_t i2s_num*, **const** *i2s_pin_config_t *pin*)

Set I2S pin number.

Inside the pin configuration structure, set I2S_PIN_NO_CHANGE for any pin where the current configuration should not be changed.

Note The I2S peripheral output signals can be connected to multiple GPIO pads. However, the I2S peripheral input signal can only be connected to one GPIO pad.

Parameters

- `i2s_num`: I2S_NUM_0 or I2S_NUM_1
- `pin`: I2S Pin structure, or NULL to set 2-channel 8-bit internal DAC pin configuration (GPIO25 & GPIO26)

Note if *pin is set as NULL, this function will initialize both of the built-in DAC channels by default. if you don't want this to happen and you want to initialize only one of the DAC channels, you can call i2s_set_dac_mode instead.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL IO error

esp_err_t **i2s_set_dac_mode**(*i2s_dac_mode_t dac_mode*)

Set I2S dac mode, I2S built-in DAC is disabled by default.

Note Built-in DAC functions are only supported on I2S0 for current ESP32 chip. If either of the built-in DAC channel are enabled, the other one can not be used as RTC DAC function at the same time.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `dac_mode`: DAC mode configurations - see i2s_dac_mode_t

esp_err_t **i2s_driver_install**(*i2s_port_t i2s_num*, **const** *i2s_config_t *i2s_config*, int *queue_size*, void **i2s_queue*)

Install and start I2S driver.

This function must be called before any I2S driver read/write operations.

Parameters

- `i2s_num`: I2S_NUM_0, I2S_NUM_1
- `i2s_config`: I2S configurations - see *i2s_config_t* struct
- `queue_size`: I2S event queue size/depth.
- `i2s_queue`: I2S event queue handle, if set NULL, driver will not use an event queue.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

esp_err_t **i2s_driver_uninstall**(*i2s_port_t i2s_num*)

Uninstall I2S driver.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2s_num`: I2S_NUM_0, I2S_NUM_1

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.

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

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

bool **use_apll**

I2S using APLL as main I2S clock, enable it to get accurate clock

int **fixed_mclk**

I2S using fixed MCLK output. If use_apll = true and fixed_mclk > 0, then the clock output for i2s is fixed and equal to the fixed_mclk value.

struct i2s_event_t
Event structure used in I2S event queue.

Public Members

i2s_event_type_t **type**
I2S event type

size_t **size**
I2S data size for I2S_DATA event

struct i2s_pin_config_t
I2S pin number for i2s_set_pin.

Public Members

int **bck_io_num**
BCK in out pin

int **ws_io_num**
WS in out pin

int **data_out_num**
DATA out pin

int **data_in_num**
DATA in pin

Macros

I2S_PIN_NO_CHANGE
Use in *i2s_pin_config_t* for pins which should not be changed

Type Definitions

typedef *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, defaut 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

3.5.6 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.

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 became important, if you are planning to use this API for purposes other that changing intensity of LEDs. Check section *Supported Range of Frequency and Duty Resolution* for more details.

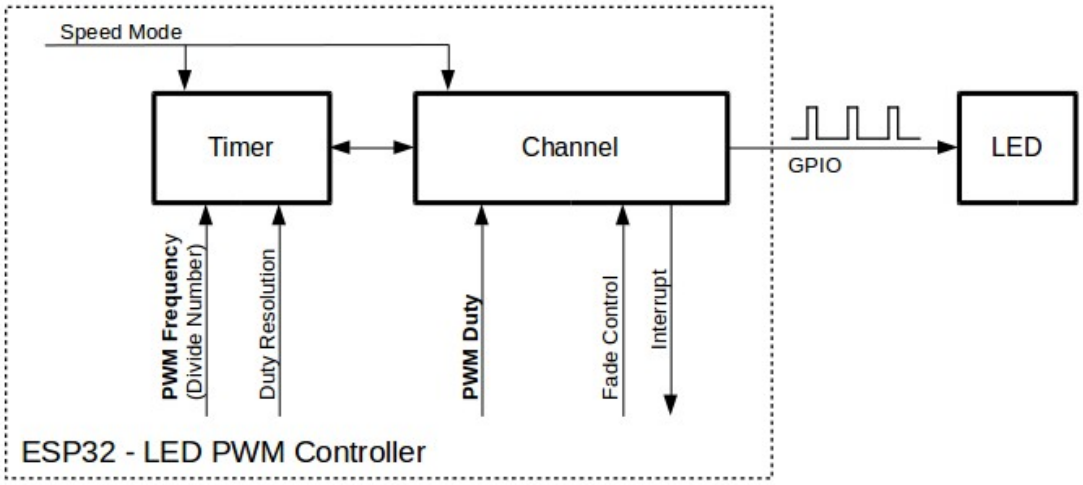


Fig. 7: Key Settings of LED PWM Controller's API

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 became 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 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 ** 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 ** 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.

Note Call this function to activate the LEDC updated parameters. After ledc_set_duty, we need to call this function to update the settings.

Note ledc_set_duty, ledc_set_duty_with_hpoint and ledc_update_duty are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_duty_and_update

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- `channel`: LEDC channel (0-7), select from ledc_channel_t

esp_err_t **ledc_stop**(*ledc_mode_t speed_mode*, *ledc_channel_t channel*, uint32_t *idle_level*)

LEDC stop. Disable LEDC output, and set idle level.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from ledc_channel_t
- `idle_level`: Set output idle level after LEDC stops.

esp_err_t **ledc_set_freq**(*ledc_mode_t speed_mode*, *ledc_timer_t timer_num*, uint32_t *freq_hz*)

LEDC set channel frequency (Hz)

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Can not find a proper pre-divider number base on the given frequency and the current duty_resolution.

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `timer_num`: LEDC timer index (0-3), select from ledc_timer_t
- `freq_hz`: Set the LEDC frequency

uint32_t **ledc_get_freq**(*ledc_mode_t speed_mode*, *ledc_timer_t timer_num*)

LEDC get channel frequency (Hz)

Return

- 0 error
- Others Current LEDC frequency

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `timer_num`: LEDC timer index (0-3), select from ledc_timer_t

esp_err_t **ledc_set_duty_with_hpoint**(*ledc_mode_t speed_mode*, *ledc_channel_t channel*, uint32_t *duty*, uint32_t *hpoint*)

LEDC set duty and hpoint value Only after calling ledc_update_duty will the duty update.

Note ledc_set_duty, ledc_set_duty_with_hpoint and ledc_update_duty are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_duty_and_update

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from ledc_channel_t

- `duty`: Set the LEDC duty, the range of duty setting is [0, (2**duty_resolution)]
- `hpoint`: Set the LEDC hpoint value(max: 0xfffff)

int **ledc_get_hpoint** (*ledc_mode_t speed_mode*, *ledc_channel_t channel*)

LEDC get hpoint value, the counter value when the output is set high level.

Return

- LEDC_ERR_VAL if parameter error
- Others Current hpoint value of LEDC channel

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from ledc_channel_t

esp_err_t **ledc_set_duty** (*ledc_mode_t speed_mode*, *ledc_channel_t channel*, uint32_t *duty*)

LEDC set duty This function do not change the hpoint value of this channel. if needed, please call ledc_set_duty_with_hpoint. only after calling ledc_update_duty will the duty update.

Note ledc_set_duty, ledc_set_duty_with_hpoint and ledc_update_duty are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_duty_and_update.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from ledc_channel_t
- `duty`: Set the LEDC duty, the range of duty setting is [0, (2**duty_resolution)]

uint32_t **ledc_get_duty** (*ledc_mode_t speed_mode*, *ledc_channel_t channel*)

LEDC get duty.

Return

- LEDC_ERR_DUTY if parameter error
- Others Current LEDC duty

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from ledc_channel_t

esp_err_t **ledc_set_fade** (*ledc_mode_t speed_mode*, *ledc_channel_t channel*, uint32_t *duty*, *ledc_duty_direction_t fade_direction*, uint32_t *step_num*, uint32_t *duty_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.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from ledc_channel_t
- `duty`: Set the start of the gradient duty, the range of duty setting is [0, (2**duty_resolution)]
- `fade_direction`: Set the direction of the gradient
- `step_num`: Set the number of the gradient
- `duty_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)]
- `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*, uint32_t *scale*, uint32_t *cycle_num*)

Set LEDC fade function.

Note Call ledc_fade_func_install() once before calling this function. Call ledc_fade_start() after this to start fading.

Note ledc_set_fade_with_step, ledc_set_fade_with_time and ledc_fade_start are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_fade_step_and_start

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- `channel`: LEDC channel index (0-7), select from ledc_channel_t
- `target_duty`: Target duty of fading [0, (2**duty_resolution) - 1]
- `scale`: Controls the increase or decrease step scale.
- `cycle_num`: increase or decrease the duty every cycle_num cycles

esp_err_t **ledc_set_fade_with_time** (*ledc_mode_t speed_mode*, *ledc_channel_t channel*, uint32_t *target_duty*, int *max_fade_time_ms*)

Set LEDC fade function, with a limited time.

Note Call ledc_fade_func_install() once before calling this function. Call ledc_fade_start() after this to start fading.

Note ledc_set_fade_with_step, ledc_set_fade_with_time and ledc_fade_start are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_fade_step_and_start

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode,

- `channel`: LEDC channel index (0-7), select from ledc_channel_t
- `target_duty`: Target duty of fading.(0 - (2 ** duty_resolution - 1)))
- `max_fade_time_ms`: The maximum time of the fading (ms).

esp_err_t **ledc_fade_func_install**(int *intr_alloc_flags*)

Install LEDC fade function. This function will occupy interrupt of LEDC module.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function already installed.

Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.

void **ledc_fade_func_uninstall**()

Uninstall LEDC fade function.

esp_err_t **ledc_fade_start**(*ledc_mode_t* *speed_mode*, *ledc_channel_t* *channel*, *ledc_fade_mode_t* *fade_mode*)

Start LEDC fading.

Note Call ledc_fade_func_install() once before calling this function. Call this API right after ledc_set_fade_with_time or ledc_set_fade_with_step before to start fading.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_ERR_INVALID_ARG Parameter error.

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel number
- `fade_mode`: Whether to block until fading done.

esp_err_t **ledc_set_duty_and_update**(*ledc_mode_t* *speed_mode*, *ledc_channel_t* *channel*, uint32_t *duty*, uint32_t *hpoint*)

A thread-safe API to set duty for LEDC channel and return when duty updated.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from ledc_channel_t
- `duty`: Set the LEDC duty, the range of duty setting is [0, (2**duty_resolution)]
- `hpoint`: Set the LEDC hpoint value(max: 0xfffff)

esp_err_t **ledc_set_fade_time_and_start**(*ledc_mode_t speed_mode*, *ledc_channel_t channel*, uint32_t *target_duty*, uint32_t *max_fade_time_ms*, *ledc_fade_mode_t fade_mode*)

A thread-safe API to set and start LEDC fade function, with a limited time.

Note Call ledc_fade_func_install() once, before calling this function.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- `channel`: LEDC channel index (0-7), select from ledc_channel_t
- `target_duty`: Target duty of fading.(0 - (2 ** duty_resolution - 1)))
- `max_fade_time_ms`: The maximum time of the fading (ms).
- `fade_mode`: choose blocking or non-blocking mode

esp_err_t **ledc_set_fade_step_and_start**(*ledc_mode_t speed_mode*, *ledc_channel_t channel*, uint32_t *target_duty*, uint32_t *scale*, uint32_t *cycle_num*, *ledc_fade_mode_t fade_mode*)

A thread-safe API to set and start LEDC fade function.

Note Call ledc_fade_func_install() once before calling this function.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- `channel`: LEDC channel index (0-7), select from ledc_channel_t
- `target_duty`: Target duty of fading [0, (2**duty_resolution) - 1]
- `scale`: Controls the increase or decrease step scale.
- `cycle_num`: increase or decrease the duty every cycle_num cycles
- `fade_mode`: choose blocking or non-blocking mode

Structures

struct ledc_channel_config_t

Configuration parameters of LEDC channel for ledc_channel_config function.

Public Members

int **gpio_num**

the LEDC output gpio_num, if you want to use gpio16, gpio_num = 16

ledc_mode_t **speed_mode**

LEDC speed speed_mode, high-speed mode or low-speed mode

ledc_channel_t **channel**

LEDC channel (0 - 7)

ledc_intr_type_t **intr_type**

configure interrupt, Fade interrupt enable or Fade interrupt disable

ledc_timer_t **timer_sel**

Select the timer source of channel (0 - 3)

uint32_t **duty**

LEDC channel duty, the range of duty setting is [0, (2**duty_resolution)]

int **hpoint**

LEDC channel hpoint value, the max value is 0xfffff

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

LEDC_ERR_VAL

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

LEDC_DUTY_DIR_MAX

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

LEDC_TIMER_MAX

enum ledc_channel_t

Values:

LEDC_CHANNEL_0 = 0
LEDC channel 0

LEDC_CHANNEL_1
LEDC channel 1

LEDC_CHANNEL_2
LEDC channel 2

LEDC_CHANNEL_3
LEDC channel 3

LEDC_CHANNEL_4
LEDC channel 4

LEDC_CHANNEL_5
LEDC channel 5

LEDC_CHANNEL_6
LEDC channel 6

LEDC_CHANNEL_7
LEDC channel 7

LEDC_CHANNEL_MAX

enum ledc_timer_bit_t

Values:

LEDC_TIMER_1_BIT = 1
LEDC PWM duty resolution of 1 bits

LEDC_TIMER_2_BIT
LEDC PWM duty resolution of 2 bits

LEDC_TIMER_3_BIT
LEDC PWM duty resolution of 3 bits

LEDC_TIMER_4_BIT
LEDC PWM duty resolution of 4 bits

LEDC_TIMER_5_BIT
LEDC PWM duty resolution of 5 bits

LEDC_TIMER_6_BIT
LEDC PWM duty resolution of 6 bits

LEDC_TIMER_7_BIT
LEDC PWM duty resolution of 7 bits

LEDC_TIMER_8_BIT
LEDC PWM duty resolution of 8 bits

LEDC_TIMER_9_BIT
LEDC PWM duty resolution of 9 bits

LEDC_TIMER_10_BIT
LEDC PWM duty resolution of 10 bits

LEDC_TIMER_11_BIT
LEDC PWM duty resolution of 11 bits

LEDC_TIMER_12_BIT
LEDC PWM duty resolution of 12 bits

LEDC_TIMER_13_BIT
LEDC PWM duty resolution of 13 bits

LEDC_TIMER_14_BIT
LEDC PWM duty resolution of 14 bits

LEDC_TIMER_15_BIT
LEDC PWM duty resolution of 15 bits

LEDC_TIMER_16_BIT
LEDC PWM duty resolution of 16 bits

LEDC_TIMER_17_BIT
LEDC PWM duty resolution of 17 bits

LEDC_TIMER_18_BIT
LEDC PWM duty resolution of 18 bits

LEDC_TIMER_19_BIT
LEDC PWM duty resolution of 19 bits

LEDC_TIMER_20_BIT
LEDC PWM duty resolution of 20 bits

LEDC_TIMER_BIT_MAX

enum ledc_fade_mode_t
Values:

LEDC_FADE_NO_WAIT = 0
LEDC fade function will return immediately

LEDC_FADE_WAIT_DONE
LEDC fade function will block until fading to the target duty

LEDC_FADE_MAX

3.5.7 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.

```
                 ____________________________________________________________________________
↪__
                 |     SYNCSIG                FAULT SIG             CAPTURE SIG                 ␣
↪   |
                 |    0   1   2               0   1   2         0   1   2                       ␣
↪   |
```

(continues on next page)

(continued from previous page)

```
                |_____________________________________________________________________________
↪G  |
INTERRUPTS<-----+ |   |   |             |   |   |         |   |   |                |
↪P  |
                | |   |   |             |   |   |         |   |   |                |
↪I  |
        _________|_|___|___|_____________|___|___|_________|___|___|_________       |
↪O  |
        |         |   |   |             |   |   |         |   |   |         |      |
↪  |
        |         |   |   |             |   |   |         |   |   |         |      |
↪M  |
        |         |   |   |           __v___v___v__     __v___v___v__       |      |
↪A  |
        |         |   |   |           |           |     |           |       |      |
↪T  |
        |         |   |   |           |   FAULT   |     |  CAPTURE  |       |      |
↪R  |
        |         |   |   |           |  HANDLER  |     |           |       |      |
↪I  |
        |         |   |   |           |           |     |___________|       |      |
↪X  |
        |         |   |   |           |___________|                         |      |
↪  |
        |         |   |   |                                                 |      |
↪  |
        |      ____v___v___v____                   _____________________    |      |
↪  |
        |      |  +---------+  |                   |  +------------+  |--------->
↪|PWM0A|
        |      |  | Timer 0 |  |                   |  | Operator 0 |  |    |      |
↪  |
        |      |  +---------+  |                   |  +------------+  |--------->
↪|PWM0B|
        |      |               |                   |                  |    |      |
↪  |
        |      |  +---------+  |                   |  +------------+  |--------->
↪|PWM1A|
        |      |  | Timer 1 |  |------------------->|  | Operator 1 |  |    |      |
↪  |
        |      |  +---------+  |                   |  +------------+  |--------->
↪|PWM1B|
        |      |               |                   |                  |    |      |
↪  |
        |      |  +---------+  |                   |  +------------+  |--------->
↪|PWM2A|
        |      |  | Timer 2 |  |                   |  | Operator 2 |  |    |      |
↪  |
        |      |  +---------+  |                   |  +------------+  |--------->
↪|PWM2B|
        |      |_______________|                   |__________________|    |      |___
↪__|
        |                                                                 |
        | MCPWM-UNIT 0/1                                                  |
        |_________________________________________________________________|
```

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/MCPWMXB)

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `op_num`: set the operator(MCPWMXA/MCPWMXB), '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/MCPWMXB) in us.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `op_num`: set the operator(MCPWMXA/MCPWMXB), '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/MCPWMXB), '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 intput_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)
- `intput_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 - negtive 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 **cmpr_a**
Set % duty cycle for operator a(MCPWMXA), i.e for 62.3% duty cycle, duty_a = 62.3

float **cmpr_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

3.5.8 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.

Note This function will disable three events: PCNT_EVT_L_LIM, PCNT_EVT_H_LIM, PCNT_EVT_ZERO.

Return

- ESP_OK Success
- ESP_ERR_INVALID_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. Please do not use pcnt_isr_service_install if this function was called.

Return

- ESP_OK Success
- ESP_ERR_NOT_FOUND Can not find the interrupt that matches the flags.
- ESP_ERR_INVALID_ARG Function pointer error.

Parameters

- `fn`: Interrupt handler function.
- `arg`: Parameter for handler function
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here. Calling esp_intr_free to unregister this ISR service if needed, but only if the handle is not NULL.

esp_err_t **pcnt_set_pin** (*pcnt_unit_t unit*, *pcnt_channel_t channel*, int *pulse_io*, int *ctrl_io*)

Configure PCNT pulse signal input pin and control input pin.

Note Set the signal input to PCNT_PIN_NOT_USED if unused.

Return

- ESP_OK Success
- ESP_ERR_INVALID_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

esp_err_t **pcnt_isr_handler_add** (*pcnt_unit_t unit*, void (**isr_handler*)) void *
, void **args*Add ISR handler for specified unit.

Call this function after using pcnt_isr_service_install() to install the PCNT driver's ISR handler service.

The ISR handlers do not need to be declared with IRAM_ATTR, unless you pass the ESP_INTR_FLAG_IRAM flag when allocating the ISR in pcnt_isr_service_install().

This ISR handler will be called from an ISR. So there is a stack size limit (configurable as "ISR stack size" in menuconfig). This limit is smaller compared to a global PCNT interrupt handler due to the additional level of indirection.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `unit`: PCNT unit number
- `isr_handler`: Interrupt handler function.
- `args`: Parameter for handler function

esp_err_t **pcnt_isr_service_install** (int *intr_alloc_flags*)

Install PCNT ISR service.

Note We can manage different interrupt service for each unit. This function will use the default ISR handle service, Calling pcnt_isr_service_uninstall to uninstall the default service if needed. Please do not use pcnt_isr_register if this function was called.

Return

- ESP_OK Success
- ESP_ERR_NO_MEM No memory to install this service
- ESP_ERR_INVALID_STATE ISR service already installed

Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.

void **pcnt_isr_service_uninstall**(void)

Uninstall PCNT ISR service, freeing related resources.

esp_err_t **pcnt_isr_handler_remove**(*pcnt_unit_t unit*)

Delete ISR handler for specified unit.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `unit`: PCNT unit number

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

3.5.9 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.

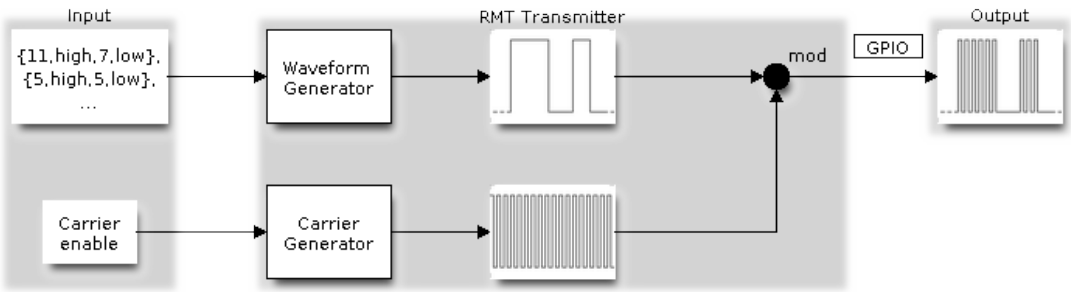


Fig. 8: RMT Transmitter Overview

The reverse operation is performed by the receiver, where a series of pulses is decoded into a list of values containing the pulse duration and binary level. A filter may be applied to remove high frequency noise from the input signal.

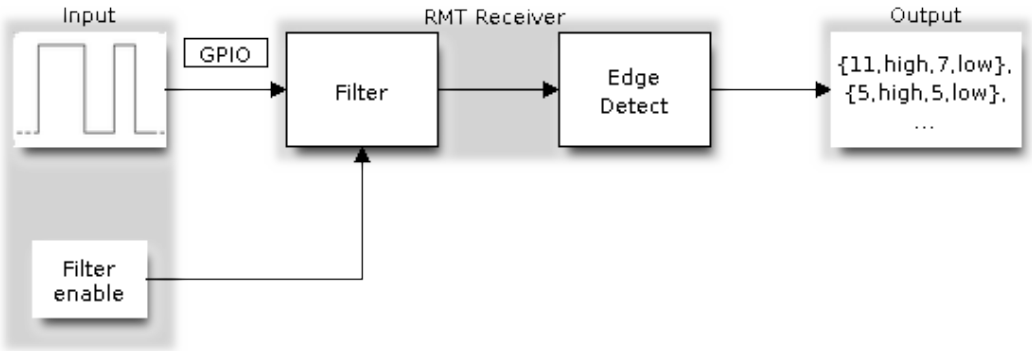


Fig. 9: RMT Receiver Overview

There couple of typical steps to setup and operate the RMT and they are discussed in the following sections:

1. *Configure Driver*
2. *Transmit Data* or *Receive Data*
3. *Change Operation Parameters*
4. *Use Interrupts*

The RMT has eight channels numbered from zero to seven. Each channel is able to independently transmit or receive data. They are referred to using indexes defined in structure `rmt_channel_t`.

Configure Driver

There are several parameters that define how particular channel operates. Most of these parameters are configured by setting specific members of `rmt_config_t` structure. Some of the parameters are common to both transmit or receive mode, and some are mode specific. They are all discussed below.

Common Parameters

- The **channel** to be configured, select one from the `rmt_channel_t` enumerator.
- The RMT **operation mode** - whether this channel is used to transmit or receive data, selected by setting a **rmt_mode** members to one of the values from `rmt_mode_t`.
- What is the **pin number** to transmit or receive RMT signals, selected by setting **gpio_num**.
- How many **memory blocks** will be used by the channel, set with **mem_block_num**.
- A **clock divider**, that will determine the range of pulse length generated by the RMT transmitter or discriminated by the receiver. Selected by setting **clk_div** to a value within [1 .. 255] range. The RMT source clock is typically APB CLK, 80Mhz by default.

Note: The period of a square wave after the clock divider is called a 'tick'. The length of the pulses generated by the RMT transmitter or discriminated by the receiver is configured in number of 'ticks'.

There are also couple of specific parameters that should be set up depending if selected channel is configured in *Transmit Mode* or *Receive Mode*:

Transmit Mode

When configuring channel in transmit mode, set **tx_config** and the following members of `rmt_tx_config_t`:

- Transmit the currently configured data items in a loop - **loop_en**
- Enable the RMT carrier signal - **carrier_en**
- Frequency of the carrier in Hz - **carrier_freq_hz**
- Duty cycle of the carrier signal in percent (%) - **carrier_duty_percent**
- Level of the RMT output, when the carrier is applied - **carrier_level**
- Enable the RMT output if idle - **idle_output_en**
- Set the signal level on the RMT output if idle - **idle_level**

Receive Mode

In receive mode, set **rx_config** and the following members of `rmt_rx_config_t`:

- Enable a filter on the input of the RMT receiver - **filter_en**
- A threshold of the filter, set in the number of ticks - **filter_ticks_thresh**. Pulses shorter than this setting will be filtered out. Note, that the range of entered tick values is [0..255].
- A pulse length threshold that will turn the RMT receiver idle, set in number of ticks - **idle_threshold**. The receiver will ignore pulses longer than this setting.

Finalize Configuration

Once the `rmt_config_t` structure is populated with parameters, it should be then invoked with `rmt_config()` to make the configuration effective.

The last configuration step is installation of the driver in memory by calling `rmt_driver_install()`. If `rx_buf_size` parameter of this function is > 0, then a ring buffer for incoming data will be allocated. A default ISR handler will be installed, see a note in *Use Interrupts*.

Now, depending on how the channel is configured, we are ready to either *Transmit Data* or *Receive Data*. This is described in next two sections.

Transmit Data

Before being able to transmit some RMT pulses, we need to define the pulse pattern. The minimum pattern recognized by the RMT controller, later called an 'item', is provided in a structure `rmt_item32_t`, see soc/esp32/include/soc/rmt_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.

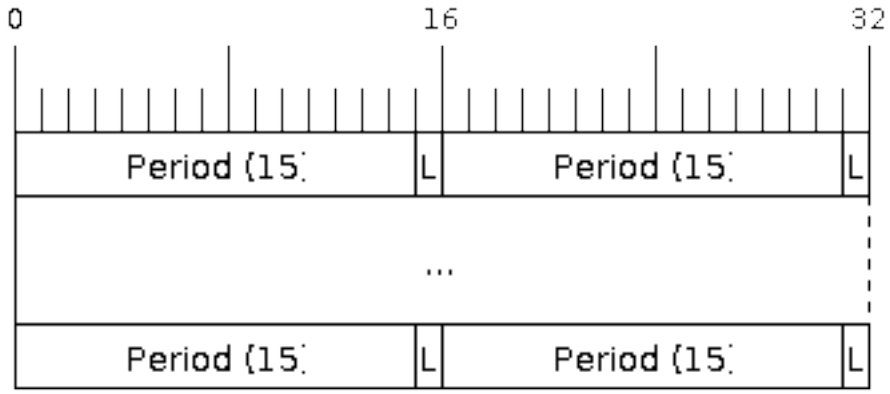


Fig. 10: Structure of RMT items (L - signal level)

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 be 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 describes 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 4 n, and end_addr_chn = RMT base address + 0x800 + 64 4 n + 64 4 RMT_MEM_SIZE_CHn mod 512 4

Note If memory block number of one channel is set to a value greater than 1, this channel will occupy the memory block of the next channel. Channel 0 can use at most 8 blocks of memory, accordingly channel 7 can only use one memory block.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel (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_idle_level** (*rmt_channel_t channel*, bool **idle_out_en*, *rmt_idle_level_t *level*)

Get RMT idle output level for transmitter.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel (0-7)
- `idle_out_en`: Pointer to accept value of enable idle.
- `level`: Pointer to accept value of output signal's level in idle state for specified channel.

esp_err_t **rmt_get_status** (*rmt_channel_t channel*, uint32_t **status*)

Get RMT status.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel (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. If ESP_INTR_FLAG_IRAM is used, please do not use the memory allocated from psram when calling rmt_write_items.

esp_err_t **rmt_driver_uninstall** (*rmt_channel_t channel*)

Uninstall RMT driver.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel (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. If ESP_INTR_FLAG_IRAM is used, please do not use the memory allocated from psram when calling rmt_write_items.
- `item_num`: RMT data item number.
- `wait_tx_done`:
  - If set 1, it will block the task and wait for sending done.
  - If set 0, it will not wait and return immediately.

esp_err_t **rmt_wait_tx_done** (*rmt_channel_t channel*, TickType_t *wait_time*)

Wait RMT TX finished.

Return

- ESP_OK RMT Tx done successfully
- ESP_ERR_TIMEOUT Exceeded the 'wait_time' given
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Driver not installed

Parameters

- `channel`: RMT channel (0 - 7)
- `wait_time`: Maximum time in ticks to wait for transmission to be complete. If set 0, return immediately with ESP_ERR_TIMEOUT if TX is busy (polling).

esp_err_t **rmt_get_ringbuf_handle**(*rmt_channel_t channel*, *RingbufHandle_t *buf_handle*)

Get ringbuffer from RMT.

Users can get the RMT RX ringbuffer handle, and process the RX data.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel (0 - 7)
- `buf_handle`: Pointer to buffer handle to accept RX ringbuffer handle.

esp_err_t **rmt_translator_init**(*rmt_channel_t channel*, *sample_to_rmt_t fn*)

Init rmt translator and register user callback. The callback will convert the raw data that needs to be sent to rmt format. If a channel is initialized more than once, tha user callback will be replaced by the later.

Return

- ESP_FAIL Init fail.
- ESP_OK Init success.

Parameters

- `channel`: RMT channel (0 - 7).
- `fn`: Point to the data conversion function.

esp_err_t **rmt_write_sample**(*rmt_channel_t channel*, **const** uint8_t **src*, size_t *src_size*, bool *wait_tx_done*)

Translate uint8_t type of data into rmt format and send it out. Requires rmt_translator_init to init the translator first.

Return

- ESP_FAIL Send fail
- ESP_OK Send success

Parameters

- `channel`: RMT channel (0 - 7).
- `src`: Pointer to the raw data.
- `src_size`: The size of the raw data.
- `wait_tx_done`: Set true to wait all data send done.

rmt_tx_end_callback_t **rmt_register_tx_end_callback**(*rmt_tx_end_fn_t function*, void **arg*)

Registers a callback that will be called when transmission ends.

Called by rmt_driver_isr_default in interrupt context.

Note Requires rmt_driver_install to install the default ISR handler.

Return the previous callback settings (members will be set to NULL if there was none)

Parameters

- `function`: Function to be called from the default interrupt handler or NULL.
- `arg`: Argument which will be provided to the callback when it is called.

Structures

struct rmt_tx_config_t

Data struct of RMT TX configure parameters.

Public Members

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

struct rmt_tx_end_callback_t
Structure encapsulating a RMT TX end callback.

Public Members

rmt_tx_end_fn_t **function**
Function which is called on RMT TX end

void ***arg**
Optional argument passed to function

Macros

RMT_MEM_BLOCK_BYTE_NUM

RMT_MEM_ITEM_NUM

Type Definitions

typedef *intr_handle_t* **rmt_isr_handle_t**

typedef void (***rmt_tx_end_fn_t**)(*rmt_channel_t* channel, void *arg)

typedef void (***sample_to_rmt_t**)(**const** void *src, rmt_item32_t *dest, size_t src_size, size_t wanted_num, size_t *translated_size, size_t *item_num)
User callback function to convert uint8_t type data to rmt format(rmt_item32_t).

This function may be called from an ISR, so, the code should be short and efficient.

Note In fact, item_num should be a multiple of translated_size, e.g. : When we convert each byte of uint8_t type data to rmt format data, the relation between item_num and translated_size should be `item_num = translated_size*8`.

Parameters

- `src`: Pointer to the buffer storing the raw data that needs to be converted to rmt format.
- `dest`: Pointer to the buffer storing the rmt format data.
- `src_size`: The raw data size.
- `wanted_num`: The number of rmt format data that wanted to get.
- `translated_size`: The size of the raw data that has been converted to rmt format, it should return 0 if no data is converted in user callback.

- `item_num`: The number of the rmt format data that actually converted to, it can be less than wanted_num if there is not enough raw data, but cannot exceed wanted_num. it should return 0 if no data was converted.

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

3.5.10 SDMMC Host Driver

Overview

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.

Pin mappings of these slots are given in the following table:

Signal	Slot 0	Slot 1
CMD	GPIO11	GPIO15
CLK	GPIO6	GPIO14
D0	GPIO7	GPIO2
D1	GPIO8	GPIO4
D2	GPIO9	GPIO12
D3	GPIO10	GPIO13
D4	GPIO16	
D5	GPIO17	
D6	GPIO5	
D7	GPIO18	
CD	any input via GPIO matrix	
WP	any input via GPIO matrix	

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()*. Note that it is not advised to specify Card Detect pin when working with SDIO cards, because in ESP32 card detect signal can also trigger SDIO slave interrupt.

Warning: Pins used by slot 0 (`HS1_*`) are also used to connect SPI flash chip in ESP-WROOM32 and ESP32-WROVER modules. These pins can not be shared between SD card and SPI flash. If you need to use Slot 0, connect SPI flash to different pins and set Efuses accordingly.

Using the SDMMC Host driver

Of all the funtions 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.

See also

See *SD/SDIO/MMC Driver* for the higher level driver which implements the protocol layer.

See *SD SPI Host Driver* for a similar driver which uses SPI controller and is limited to SPI mode of SD protocol.

API Reference

Header File

- driver/include/driver/sdmmc_host.h

Functions

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

esp_err_t **sdmmc_host_init_slot**(int *slot*, **const** *sdmmc_slot_config_t* **slot_config*)

Initialize given slot of SDMMC peripheral.

On the ESP32, SDMMC peripheral has two slots:

- Slot 0: 8-bit wide, maps to HS1_* signals in PIN MUX
- Slot 1: 4-bit wide, maps to HS2_* signals in PIN MUX

Card detect and write protect signals can be routed to arbitrary GPIOs using GPIO matrix.

Note This function is not thread safe

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if host has not been initialized using sdmmc_host_init

Parameters

- `slot`: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- `slot_config`: additional configuration for the slot

esp_err_t **sdmmc_host_set_bus_width** (int *slot*, size_t *width*)

Select bus width to be used for data transfer.

SD/MMC card must be initialized prior to this command, and a command to set bus width has to be sent to the card (e.g. SD_APP_SET_BUS_WIDTH)

Note This function is not thread safe

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if slot number or width is not valid

Parameters

- `slot`: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- `width`: bus width (1, 4, or 8 for slot 0; 1 or 4 for slot 1)

size_t **sdmmc_host_get_slot_width** (int *slot*)

Get bus width configured in `sdmmc_host_init_slot` to be used for data transfer.

Return configured bus width of the specified slot.

Parameters

- `slot`: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)

esp_err_t **sdmmc_host_set_card_clk** (int *slot*, uint32_t *freq_khz*)

Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

Note This function is not thread safe

Return

- ESP_OK on success
- other error codes may be returned in the future

Parameters

- `slot`: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- `freq_khz`: card clock frequency, in kHz

esp_err_t **sdmmc_host_do_transaction** (int *slot*, *sdmmc_command_t* **cmdinfo*)

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

Note This function is not thread safe w.r.t. init/deinit functions, and bus width/clock speed configuration functions. Multiple tasks can call sdmmc_host_do_transaction as long as other sdmmc_host_* functions are not called.

Attention Data buffer passed in cmdinfo->data must be in DMA capable memory

Return

- ESP_OK on success
- ESP_ERR_TIMEOUT if response or data transfer has timed out
- ESP_ERR_INVALID_CRC if response or data transfer CRC check has failed
- ESP_ERR_INVALID_RESPONSE if the card has sent an invalid response
- ESP_ERR_INVALID_SIZE if the size of data transfer is not valid in SD protocol
- ESP_ERR_INVALID_ARG if the data buffer is not in DMA capable memory

Parameters

- `slot`: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- `cmdinfo`: pointer to structure describing command and data to transfer

esp_err_t **sdmmc_host_io_int_enable** (int *slot*)

Enable IO interrupts.

This function configures the host to accept SDIO interrupts.

Return returns ESP_OK, other errors possible in the future

Parameters

- `slot`: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)

esp_err_t **sdmmc_host_io_int_wait** (int *slot*, TickType_t *timeout_ticks*)

Block until an SDIO interrupt is received, or timeout occurs.

Return

- ESP_OK on success (interrupt received)
- ESP_ERR_TIMEOUT if the interrupt did not occur within timeout_ticks

Parameters

- `slot`: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- `timeout_ticks`: number of RTOS ticks to wait for the interrupt

esp_err_t **sdmmc_host_deinit** ()

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

Structures

struct sdmmc_slot_config_t
Extra configuration for SDMMC peripheral slot

Public Members

gpio_num_t **gpio_cd**
GPIO number of card detect signal.

gpio_num_t **gpio_wp**
GPIO number of write protect signal.

uint8_t **width**
Bus width used by the slot (might be less than the max width supported)

Macros

SDMMC_HOST_SLOT_0
SDMMC slot 0.

SDMMC_HOST_SLOT_1
SDMMC slot 1.

SDMMC_HOST_DEFAULT()
Default *sdmmc_host_t* structure initializer for SDMMC peripheral.

Uses SDMMC peripheral, with 4-bit mode enabled, and max frequency set to 20MHz

SDMMC_SLOT_NO_CD
indicates that card detect line is not used

SDMMC_SLOT_NO_WP
indicates that write protect line is not used

SDMMC_SLOT_WIDTH_DEFAULT
use the default width for the slot (8 for slot 0, 4 for slot 1)

SDMMC_SLOT_CONFIG_DEFAULT()
Macro defining default configuration of SDMMC host slot

3.5.11 SD SPI Host Driver

Overview

SPI controllers accessible via spi_master driver (HSPI, VSPI) can be used to work with SD cards. The driver which provides this capability is called "SD SPI Host", due to its similarity with the *SDMMC Host* driver.

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.

See *SD/SDIO/MMC Driver* for the higher level driver which implements the protocol layer.

API Reference

Header File

- driver/include/driver/sdspi_host.h

Functions

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

esp_err_t **sdspi_host_init_slot**(int *slot*, **const** *sdspi_slot_config_t* **slot_config*)

Initialize SD SPI driver for the specific SPI controller.

Note This function is not thread safe

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if sdspi_init_slot has invalid arguments
- ESP_ERR_NO_MEM if memory can not be allocated
- other errors from the underlying spi_master and gpio drivers

Parameters

- `slot`: SPI controller to use (HSPI_HOST or VSPI_HOST)
- `slot_config`: pointer to slot configuration structure

esp_err_t **sdspi_host_do_transaction**(int *slot*, *sdmmc_command_t* **cmdinfo*)

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

Note This function is not thread safe w.r.t. init/deinit functions, and bus width/clock speed configuration functions. Multiple tasks can call sdspi_host_do_transaction as long as other sdspi_host_* functions are not called.

Return

- ESP_OK on success
- ESP_ERR_TIMEOUT if response or data transfer has timed out
- ESP_ERR_INVALID_CRC if response or data transfer CRC check has failed
- ESP_ERR_INVALID_RESPONSE if the card has sent an invalid response

Parameters

- `slot`: SPI controller (HSPI_HOST or VSPI_HOST)
- `cmdinfo`: pointer to structure describing command and data to transfer

esp_err_t **sdspi_host_set_card_clk** (int *slot*, uint32_t *freq_khz*)

Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

Note This function is not thread safe

Return

- ESP_OK on success
- other error codes may be returned in the future

Parameters

- `slot`: SPI controller (HSPI_HOST or VSPI_HOST)
- `freq_khz`: card clock frequency, in kHz

esp_err_t **sdspi_host_deinit** ()

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

Structures

struct sdspi_slot_config_t

Extra configuration for SPI host

Public Members

gpio_num_t **gpio_miso**

GPIO number of MISO signal.

gpio_num_t **gpio_mosi**

GPIO number of MOSI signal.

gpio_num_t **gpio_sck**

GPIO number of SCK signal.

gpio_num_t **gpio_cs**
GPIO number of CS signal.

gpio_num_t **gpio_cd**
GPIO number of card detect signal.

gpio_num_t **gpio_wp**
GPIO number of write protect signal.

int **dma_channel**
DMA channel to be used by SPI driver (1 or 2)

Macros

SDSPI_HOST_DEFAULT()
Default *sdmmc_host_t* structure initializer for SD over SPI driver.

Uses SPI mode and max frequency set to 20MHz

'slot' can be set to one of HSPI_HOST, VSPI_HOST.

SDSPI_SLOT_NO_CD
indicates that card detect line is not used

SDSPI_SLOT_NO_WP
indicates that write protect line is not used

SDSPI_SLOT_CONFIG_DEFAULT()
Macro defining default configuration of SPI host

3.5.12 SDIO Card Slave Driver

Overview

Note: At the moment, this code has been proven to work on the Wrover-Kit V3. Earlier versions of the Wrover-Kit and other development kits are electrically incompatible with this code. Functionality on other devboards is untested.

The ESP32 SDIO Card peripherals (Host, Slave) shares two sets of pins as below table. The first set is usually occupied by SPI0 bus which is responsible for the SPI flash holding the code to run. This means SDIO slave driver can only runs on the second set of pins while SDIO host is not using it.

Pin Name	Slot1	Slot2
	GPIO Number	
CLK	6	14
CMD	11	15
DAT0	7	2
DAT1	8	4
DAT2	9	12
DAT3	10	13

The SDIO slave can run under 3 modes: SPI, 1-bit SD and 4-bit SD modes, which is detected automatically by the hardware. According to the SDIO specification, the host initialize the slave into SD mode by first sending CMD0 with DAT3 pin high, while initialize the slave into SPI mode by sending CMD0 with CS pin (the same pin as DAT3) low.

After the initialization, the host can enable the 4-bit SD mode by writing CCCR register 0x07 by CMD52. All the bus detection process are handled by the slave peripheral.

The host has to communicate with the slave by an ESP-slave-specific protocol. The slave driver offers 3 services over Function 1 access by CMD52 and CMD53: (1) a sending FIFO and a receiving FIFO, (2) 52 8-bit R/W registers shared by host and slave, (3) 16 interrupt sources (8 from host to slave, and 8 from slave to host).

Terminology

The SDIO slave driver uses the following terms:

- Transfer: a transfer is always started by a command token from the host, and may contain a reply and several data blocks. ESP32 slave software is based on transfers.
- Sending: slave to host transfers.
- Receiving: host to slave transfers.

Note: Register names in ESP Rechnical Reference Manual are oriented from the point of view of the host, i.e. 'rx' registers refer to sending, while 'tx' registers refer to receiving. We're not using *tx* or *rx* in the driver to avoid ambiguities.

- FIFO: specific address in Function 1 that can be access by CMD53 to read/write large amount of data. The address is related to the length requested to read from/write to the slave in a single transfer: *requested length* = 0x1F800-address.
- **Ownership: When the driver takes ownership of a buffer, it means the driver can randomly read/write the buffer** (mostly by the hardware). The application should not read/write the buffer until the ownership is returned to the application. If the application reads from a buffer owned by a receiving driver, the data read can be random; if the application writes to a buffer owned by a sending driver, the data sent may be corrupted.
- Requested length: The length requested in one transfer determined by the FIFO address.
- Transfer length: The length requested in one transfer determined by the CMD53 byte/block count field.

Note: Requested length is different from the transfer length. ESP32 slave DMA base on the *requested length* rather than the *transfer length*. The *transfer length* should be no shorter than the *requested length*, and the rest part will be filled with 0 (sending) or discard (receiving).

- Receiving buffer size: The buffer size is pre-defined between the host and the slave before communication starts. Slave application has to set the buffer size during initialization by the `recv_buffer_size` member of `sdio_slave_config_t`.
- Interrupts: the esp32 slave support interrupts in two directions: from host to slave (called slave interrupts below) and from slave to host (called host interrupts below). See more in *Interrupts*.
- Registers: specific address in Function 1 access by CMD52 or CMD53.

ESP SDIO Slave Protocol

The communication protocol slave used to communicate with the host is ESP32 specific, please refer to *ESP SDIO slave protocol*, or example peripherals/sdio for designing a host.

ESP SDIO slave protocol

The protocol is based on Function 1 access by CMD52 and CMD53, offering 3 services: (1) sending and receiving FIFO, (2) 52 8-bit R/W register shared by host and slave, (3) 8 general purpose interrupt sources from host to slave and 8 in the oppsite direction.

The host should access the registers below as described to communicate with slave.

Slave register table

32-bit

- 0x044 (TOKEN_RDATA): in which bit 27-16 holds the receiving buffer number.
- 0x058 (INT_ST): holds the interrupt source bits from slave to host.
- 0x060 (PKT_LEN): holds the accumulated length (by byte) to be sent from slave to host.
- 0x0D4 (INT_CLR): write 1 to clear interrupt bits corresponding to INT_ST.
- 0x0DC (INT_ENA): mask bits for interrupts from slave to host.

8-bit

Shared general purpose registers:

- 0x06C-0x077: R/W registers 0-11 shared by slave and host.
- 0x07A-0x07B: R/W registers 14-15 shared by slave and host.
- 0x07E-0x07F: R/W registers 18-19 shared by slave and host.
- 0x088-0x08B: R/W registers 24-27 shared by slave and host.
- 0x09C-0x0BB: R/W registers 32-63 shared by slave and host.

Interrupt Registers: - 0x08D (SLAVE_INT): bits for host to interrupt slave. auto clear.

FIFO (sending and receiving)

0x090 - 0x1F7FF are reserved for FIFOs.

Note: This includes the CMD52 and CMD53 (block mode or byte mode).

The function number should be set to 1, OP Code should be set to 1 (for CMD53).

The slave will respond with the length according to the length field in CMD53 (1 of CMD52), with the data longer than *requested length* filled with 0 (sending) or discard (receiving).

Interrupts

For the host interrupts, the slave raise the interrupt by pulling DAT1 line down at a proper time (level sensitive). The host detect this and read the INT_ST register to see the source. Then the host can clear it by writing the INT_CLR reg-

ister and do something with the interrupt. The host can also mask unneeded sources by clearing the bits in INT_ENA register corresponding to the sources. If all the sources are cleared (or masked), the DAT1 line goes inactive.

`sdio_slave_hostint_t` (*SDIO Card Slave Driver*) shows the bit definition corresponding to host interrupt sources.

For the slave interrupts, the host send transfers to write the SLAVE_INT register. Once a bit is written from 0 to 1, the slave hardware and driver will detect it and inform the app.

Receiving FIFO

To write the receiving FIFO in the slave, host should work in the following steps:

1. Read the TOKEN1 field (bits 27-16) of TOKEN_RDATA (0x044) register. The buffer number remaining is TOKEN1 minus the number of buffers used by host.
2. Make sure the buffer number is sufficient (*buffer_size* * *buffer_num* is greater than data to write, *buffer_size* is pre-defined between the host and the slave before the communication starts). Or go back to step 1 until the buffer is enough.
3. Write to the FIFO address with CMD53. Note that the *requested length* should not be larger than calculated in step 2, and the FIFO address is related to *rquested length*.
4. Calculate used buffers, note that non-full buffer at the tail should be seen as one that is used.

Sending FIFO

To read the sending FIFO in the slave, host should work in the following steps:

1. Wait for the interrupt line to be active (optional, low by default).
2. Read (poll) the interrupt bits in INT_ST register to see whether new packets exists.
3. If new packets are ready, reads the PKT_LEN reg. The data length to read from slave is PKT_LEN minuses the length that has been read from the host. If the PKT_LEN is not larger than used, wait and poll until the slave is ready and update the PKT_LEN.
4. Read from the FIFO with CMD53. Note that the *requested length* should not be larger than calculated in step3, and the FIFO address is related to *requested length*.
5. Recored read length.

Interrupts

There are interrupts from host to slave, and from slave to host to help communicating conveniently.

Slave Interrupts

The host can interrupt the slave by writing any one bit in the register 0x08D. Once any bit of the register is set, an interrupt is raised and the SDIO slave driver calls the callback function defined in the `slave_intr_cb` member in the `sdio_slave_config_t` structure.

Note: The callback function is called in the ISR, do not use any delay, loop or spinlock in the callback.

There's another set of functions can be used. You can call `sdio_slave_wait_int` to wait for an interrupt within a certain time, or call `sdio_slave_clear_int` to clear interrupts from host. The callback function can work with the wait functions perfectly.

Host Interrupts

The slave can interrupt the host by an interrupt line (at certain time) which is level sensitive. When the host see the interrupt line pulled down, it may read the slave interrupt status register, to see the interrupt source. Host can clear interrupt bits, or choose to disable a interrupt source. The interrupt line will hold active until all the sources are cleared or disabled.

There are several dedicated interrupt sources as well as general purpose sources. see `sdio_slave_hostint_t` for more information.

Shared Registers

There are 52 8-bit R/W shared registers to share information between host and slave. The slave can write or read the registers at any time by `sdio_slave_read_reg` and `sdio_slave_write_reg`. The host can access (R/W) the register by CMD52 or CMD53.

Receiving FIFO

When the host is going to send the slave some packets, it has to check whether the slave is ready to receive by reading the buffer number of slave.

To allow the host sending data to the slave, the application has to load buffers to the slave driver by the following steps:

1. Register the buffer by calling `sdio_slave_recv_register_buf`, and get the handle of the registered buffer. The driver will allocate memory for the linked-list descriptor needed to link the buffer onto the hardware.
2. Load buffers onto the driver by passing the buffer handle to `sdio_slave_recv_load_buf`.
3. Call `sdio_slave_recv` to get the received data. If non-blocking call is needed, set `wait=0`.
4. Pass the handle of processed buffer back to the driver by `sdio_recv_load_buf` again.

Note: To avoid overhead from copying data, the driver itself doesn't have any buffer inside, the application is responsible to offer new buffers in time. The DMA will automatically store received data to the buffer.

Sending FIFO

Each time the slave has data to send, it raises an interrupt and the host will request for the packet length. There are two sending modes:

- Stream Mode: when a buffer is loaded to the driver, the buffer length will be counted into the packet length requested by host in the incoming communications. Regardless previous packets are sent or not. This means the host can get data of several buffers in one transfer.
- Packet Mode: the packet length is updated packet by packet, and only when previous packet is sent. This means that the host can only get data of one buffer in one transfer.

Note: To avoid overhead from copying data, the driver itself doesn't have any buffer inside. Namely, the DMA takes data directly from the buffer provided by the application. The application should not touch the buffer until the sending is finished.

The sending mode can be set in the `sending_mode` member of `sdio_slave_config_t`, and the buffer numbers can be set in the `send_queue_size`. All the buffers are restricted to be no larger than 4092 bytes. Though in the stream mode several buffers can be sent in one transfer, each buffer is still counted as one in the queue.

The application can call `sdio_slave_transmit` to send packets. In this case the function returns when the transfer is sucessfully done, so the queue is not fully used. When higher effeciency is required, the application can use the following functions instead:

1. Pass buffer information (address, length, as well as an `arg` indicating the buffer) to `sdio_slave_send_queue`. If non-blocking call is needed, set `wait=0`. If the `wait` is not `portMAX_DELAY` (wait until success), application has to check the result to know whether the data is put in to the queue or discard.
2. Call `sdio_slave_send_get_finished` to get and deal with a finished transfer. A buffer should be keep unmodified until returned from `sdio_slave_send_get_finished`. This means the buffer is actually sent to the host, rather than just staying in the queue.

There are several ways to use the `arg` in the queue parameter:

1. Directly point `arg` to a dynamic-allocated buffer, and use the `arg` to free it when transfer finished.
2. Wrap transfer informations in a transfer structure, and point `arg` to the structure. You can use the structure to do more things like:

```
typedef struct {
    uint8_t* buffer;
    size_t   size;
    int      id;
}sdio_transfer_t;

//and send as:
sdio_transfer_t trans = {
    .buffer = ADDRESS_TO_SEND,
    .size = 8,
    .id = 3,  //the 3rd transfer so far
};
sdio_slave_send_queue(trans.buffer, trans.size, &trans, portMAX_DELAY);

//... maybe more transfers are sent here

//and deal with finished transfer as:
sdio_transfer_t* arg = NULL;
sdio_slave_send_get_finished((void**)&arg, portMAX_DELAY);
ESP_LOGI("tag", "(%d) successfully send %d bytes of %p", arg->id, arg->size, arg->
↪buffer);
some_post_callback(arg); //do more things
```

3. Working with the receiving part of this driver, point `arg` to the receive buffer handle of this buffer. So that we can directly use the buffer to receive data when it's sent:

```
uint8_t buffer[256]={1,2,3,4,5,6,7,8};
sdio_slave_buf_handle_t handle = sdio_slave_recv_register_buf(buffer);
sdio_slave_send_queue(buffer, 8, handle, portMAX_DELAY);
```

(continues on next page)

(continued from previous page)

```
//... maybe more transfers are sent here

//and load finished buffer to receive as
sdio_slave_buf_handle_t handle = NULL;
sdio_slave_send_get_finished((void**)&handle, portMAX_DELAY);
sdio_slave_recv_load_buf(handle);
```

More about this, see peripherals/sdio.

Application Example

Slave/master communication: peripherals/sdio.

API Reference

Header File

- driver/include/driver/sdio_slave.h

Functions

esp_err_t **sdio_slave_initialize**(*sdio_slave_config_t *config*)

Initialize the sdio slave driver

Return

- ESP_ERR_NOT_FOUND if no free interrupt found.
- ESP_ERR_INVALID_STATE if already initialized.
- ESP_ERR_NO_MEM if fail due to memory allocation failed.
- ESP_OK if success

Parameters

- `config`: Configuration of the sdio slave driver.

void **sdio_slave_deinit**()

De-initialize the sdio slave driver to release the resources.

esp_err_t **sdio_slave_start**()

Start hardware for sending and receiving, as well as set the IOREADY1 to 1.

Note The driver will continue sending from previous data and PKT_LEN counting, keep data received as well as start receiving from current TOKEN1 counting. See `sdio_slave_reset`.

Return

- ESP_ERR_INVALID_STATE if already started.
- ESP_OK otherwise.

void **sdio_slave_stop**()

Stop hardware from sending and receiving, also set IOREADY1 to 0.

Note this will not clear the data already in the driver, and also not reset the PKT_LEN and TOKEN1 counting. Call `sdio_slave_reset` to do that.

esp_err_t **sdio_slave_reset**()

Clear the data still in the driver, as well as reset the PKT_LEN and TOKEN1 counting.

Return always return ESP_OK.

sdio_slave_buf_handle_t **sdio_slave_recv_register_buf**(uint8_t **start*)

Register buffer used for receiving. All buffers should be registered before used, and then can be used (again) in the driver by the handle returned.

Note The driver will use and only use the amount of space specified in the `recv_buffer_size` member set in the *`sdio_slave_config_t`*. All buffers should be larger than that. The buffer is used by the DMA, so it should be DMA capable and 32-bit aligned.

Return The buffer handle if success, otherwise NULL.

Parameters

- `start`: The start address of the buffer.

esp_err_t **sdio_slave_recv_unregister_buf**(*sdio_slave_buf_handle_t handle*)

Unregister buffer from driver, and free the space used by the descriptor pointing to the buffer.

Return ESP_OK if success, ESP_ERR_INVALID_ARG if the handle is NULL or the buffer is being used.

Parameters

- `handle`: Handle to the buffer to release.

esp_err_t **sdio_slave_recv_load_buf**(*sdio_slave_buf_handle_t handle*)

Load buffer to the queue waiting to receive data. The driver takes ownership of the buffer until the buffer is returned by `sdio_slave_send_get_finished` after the transaction is finished.

Return

- ESP_ERR_INVALID_ARG if invalid handle or the buffer is already in the queue. Only after the buffer is returened by `sdio_slave_recv` can you load it again.
- ESP_OK if success

Parameters

- `handle`: Handle to the buffer ready to receive data.

esp_err_t **sdio_slave_recv**(*sdio_slave_buf_handle_t **handle_ret*, uint8_t ***start_o*, size_t **len_o*, TickType_t *wait*)

Get received data if exist. The driver returns the ownership of the buffer to the app.

Note Call `sdio_slave_load_buf` with the handle to re-load the buffer onto the link list, and receive with the same buffer again. The address and length of the buffer got here is the same as got from `sdio_slave_get_buffer`.

Return

- ESP_ERR_INVALID_ARG if handle_ret is NULL
- ESP_ERR_TIMEOUT if timeout before receiving new data
- ESP_OK if success

Parameters

- `handle_ret`: Handle to the buffer holding received data. Use this handle in `sdio_slave_recv_load_buf` to receive in the same buffer again.
- `start_o`: Start address output, set to NULL if not needed.
- `len_o`: Actual length of the data in the buffer, set to NULL if not needed.
- `wait`: Time to wait before data received.

uint8_t ***sdio_slave_recv_get_buf** (*sdio_slave_buf_handle_t handle*, size_t **len_o*)

Retrieve the buffer corresponding to a handle.

Return buffer address if success, otherwise NULL.

Parameters

- `handle`: Handle to get the buffer.
- `len_o`: Output of buffer length

esp_err_t **sdio_slave_send_queue** (uint8_t **addr*, size_t *len*, void **arg*, TickType_t *wait*)

Put a new sending transfer into the send queue. The driver takes ownership of the buffer until the buffer is returned by `sdio_slave_send_get_finished` after the transaction is finished.

Return

- ESP_ERR_INVALID_ARG if the length is not greater than 0.
- ESP_ERR_TIMEOUT if the queue is still full until timeout.
- ESP_OK if success.

Parameters

- `addr`: Address for data to be sent. The buffer should be DMA capable and 32-bit aligned.
- `len`: Length of the data, should not be longer than 4092 bytes (may support longer in the future).
- `arg`: Argument to returned in `sdio_slave_send_get_finished`. The argument can be used to indicate which transaction is done, or as a parameter for a callback. Set to NULL if not needed.
- `wait`: Time to wait if the buffer is full.

esp_err_t **sdio_slave_send_get_finished** (void ***out_arg*, TickType_t *wait*)

Return the ownership of a finished transaction.

Return ESP_ERR_TIMEOUT if no transaction finished, or ESP_OK if succeed.

Parameters

- `out_arg`: Argument of the finished transaction. Set to NULL if unused.
- `wait`: Time to wait if there's no finished sending transaction.

esp_err_t **sdio_slave_transmit** (uint8_t **addr*, size_t *len*)

Start a new sending transfer, and wait for it (blocked) to be finished.

Return

- ESP_ERR_INVALID_ARG if the length of descriptor is not greater than 0.
- ESP_ERR_TIMEOUT if the queue is full or host do not start a transfer before timeout.

- ESP_OK if success.

Parameters

- `addr`: Start address of the buffer to send
- `len`: Length of buffer to send.

uint8_t **sdio_slave_read_reg**(int *pos*)
Read the spi slave register shared with host.

Note register 28 to 31 are reserved for interrupt vector.

Return value of the register.

Parameters

- `pos`: register address, 0-27 or 32-63.

esp_err_t **sdio_slave_write_reg**(int *pos*, uint8_t *reg*)
Write the spi slave register shared with host.

Note register 29 and 31 are used for interrupt vector.

Return ESP_ERR_INVALID_ARG if address wrong, otherwise ESP_OK.

Parameters

- `pos`: register address, 0-11, 14-15, 18-19, 24-27 and 32-63, other address are reserved.
- `reg`: the value to write.

sdio_slave_hostint_t **sdio_slave_get_host_intena**()
Get the interrupt enable for host.

Return the interrupt mask.

void **sdio_slave_set_host_intena**(*sdio_slave_hostint_t ena*)
Set the interrupt enable for host.

Parameters

- `ena`: Enable mask for host interrupt.

esp_err_t **sdio_slave_send_host_int**(uint8_t *pos*)
Interrupt the host by general purpose interrupt.

Return

- ESP_ERR_INVALID_ARG if interrupt num error
- ESP_OK otherwise

Parameters

- `pos`: Interrupt num, 0-7.

void **sdio_slave_clear_host_int**(uint8_t *mask*)
Clear general purpose interrupt to host.

Parameters

- `mask`: Interrupt bits to clear, by bit mask.

esp_err_t **sdio_slave_wait_int** (int *pos*, TickType_t *wait*)
Wait for general purpose interrupt from host.

Note this clears the interrupt at the same time.

Return ESP_OK if success, ESP_ERR_TIMEOUT if timeout.

Parameters

- `pos`: Interrupt source number to wait for. is set.
- `wait`: Time to wait before interrupt triggered.

Structures

struct sdio_slave_config_t
Configuration of SDIO slave.

Public Members

sdio_slave_timing_t **timing**
timing of sdio_slave. see `sdio_slave_timing_t`.

sdio_slave_sending_mode_t **sending_mode**
mode of sdio_slave. `SDIO_SLAVE_MODE_STREAM` if the data needs to be sent as much as possible; `SDIO_SLAVE_MODE_PACKET` if the data should be sent in packets.

int **send_queue_size**
max buffers that can be queued before sending.

size_t **recv_buffer_size**
If buffer_size is too small, it costs more CPU time to handle larger number of buffers. If buffer_size is too large, the space larger than the transaction length is left blank but still counts a buffer, and the buffers are easily run out. Should be set according to length of data really transferred. All data that do not fully fill a buffer is still counted as one buffer. E.g. 10 bytes data costs 2 buffers if the size is 8 bytes per buffer. Buffer size of the slave pre-defined between host and slave before communication. All receive buffer given to the driver should be larger than this.

sdio_event_cb_t **event_cb**
when the host interrupts slave, this callback will be called with interrupt number (0-7).

Macros

SDIO_SLAVE_RECV_MAX_BUFFER

Type Definitions

typedef void (*__sdio_event_cb_t__) (uint8_t event)

typedef void *__sdio_slave_buf_handle_t__
Handle of a receive buffer, register a handle by calling `sdio_slave_recv_register_buf`. Use the handle to load the buffer to the driver, or call `sdio_slave_recv_unregister_buf` if it is no longer used.

Enumerations

enum sdio_slave_hostint_t

Mask of interrupts sending to the host.

Values:

SDIO_SLAVE_HOSTINT_SEND_NEW_PACKET = HOST_SLC0_RX_NEW_PACKET_INT_ENA
New packet available.

SDIO_SLAVE_HOSTINT_RECV_OVF = HOST_SLC0_TX_OVF_INT_ENA
Slave receive buffer overflow.

SDIO_SLAVE_HOSTINT_SEND_UDF = HOST_SLC0_RX_UDF_INT_ENA
Slave sending buffer underflow (this case only happen when the host do not request for packet according to the packet len).

SDIO_SLAVE_HOSTINT_BIT7 = HOST_SLC0_TOHOST_BIT7_INT_ENA
General purpose interrupt bits that can be used by the user.

SDIO_SLAVE_HOSTINT_BIT6 = HOST_SLC0_TOHOST_BIT6_INT_ENA

SDIO_SLAVE_HOSTINT_BIT5 = HOST_SLC0_TOHOST_BIT5_INT_ENA

SDIO_SLAVE_HOSTINT_BIT4 = HOST_SLC0_TOHOST_BIT4_INT_ENA

SDIO_SLAVE_HOSTINT_BIT3 = HOST_SLC0_TOHOST_BIT3_INT_ENA

SDIO_SLAVE_HOSTINT_BIT2 = HOST_SLC0_TOHOST_BIT2_INT_ENA

SDIO_SLAVE_HOSTINT_BIT1 = HOST_SLC0_TOHOST_BIT1_INT_ENA

SDIO_SLAVE_HOSTINT_BIT0 = HOST_SLC0_TOHOST_BIT0_INT_ENA

enum sdio_slave_timing_t

Timing of SDIO slave.

Values:

SDIO_SLAVE_TIMING_PSEND_PSAMPLE = 0
Send at posedge, and sample at posedge. Default value for HS mode. Normally there's no problem using this to work in DS mode.

SDIO_SLAVE_TIMING_NSEND_PSAMPLE
Send at negedge, and sample at posedge. Default value for DS mode and below.

SDIO_SLAVE_TIMING_PSEND_NSAMPLE
Send at posedge, and sample at negedge.

SDIO_SLAVE_TIMING_NSEND_NSAMPLE
Send at negedge, and sample at negedge.

enum sdio_slave_sending_mode_t

Configuration of SDIO slave mode.

Values:

SDIO_SLAVE_SEND_STREAM = 0
Stream mode, all packets to send will be combined as one if possible.

SDIO_SLAVE_SEND_PACKET = 1
Packet mode, one packets will be sent one after another (only increase packet_len if last packet sent).

3.5.13 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 Vdc = 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

3.5.14 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 mode, the read and write phases are combined, and the SPI host reads and writes data simultaneously. The total transaction length is decided by `command_bits + address_bits + trans_conf.length`, while the `trans_conf.rx_length` only determins length of data received into the buffer.

While in half duplex mode, the host have independent write and read phases. The length of write phase and read phase are decided by `trans_conf.length` and `trans_conf.rx_length` respectively.

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.

GPIO matrix and IOMUX

Most peripheral signals in ESP32 can connect directly to a specific GPIO, which is called its IOMUX pin. When a peripheral signal is routed to a pin other than its IOMUX pin, ESP32 uses the less direct GPIO matrix to make this connection.

If the driver is configured with all SPI signals set to their specific IOMUX pins (or left unconnected), it will bypass the GPIO matrix. If any SPI signal is configured to a pin other than its IOMUx pin, the driver will automatically route all the signals via the GPIO Matrix. The GPIO matrix samples all signals at 80MHz and sends them between the GPIO and the peripheral.

When the GPIO matrix is used, signals faster than 40MHz cannot propagate and the setup time of MISO is more easily violated, since the input delay of MISO signal is increased. The maximum clock frequency with GPIO Matrix is 40MHz or less, whereas using all IOMUX pins allows 80MHz.

Note: More details about influence of input delay on the maximum clock frequency, see *Timing considerations* below.

IOMUX pins for SPI controllers are as below:

Pin Name	HSPI	VSPI
	GPIO Number	
CS0*	15	5
SCLK	14	18
MISO	12	19
MOSI	13	23
QUADWP	2	22
QUADHD	4	21

note * Only the first device attaching to the bus can use CS0 pin.

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 synchroneously 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 below:

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 memcpy of temporary buffers.

Note: Half duplex transactions with both read and write phases are not supported when using DMA. See *Known Issues* for details and workarounds.

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`.

Speed and Timing Considerations

Transferring speed

There're three factors limiting the transferring speed: (1) The transaction interval, (2) The SPI clock frequency used. (3) The cache miss of SPI functions including callbacks. When large transactions are used, the clock frequency determines the transferring speed; while the interval effects the speed a lot if small transactions are used.

1. Transaction interval: The interval mainly comes from the cost of FreeRTOS queues and the time switching between tasks and the ISR. It also takes time for the software to setup spi peripheral registers as well as copy data to FIFOs, or setup DMA links. Depending on whether the DMA is used, the interval of an one-byte transaction is around 25us typically.
    1. The CPU is blocked and switched to other tasks when the transaction is in flight. This save the cpu time but increase the interval.
    2. When the DMA is enabled, it needs about 2us per transaction to setup the linked list. When the master is transferring, it automatically read data from the linked list. If the DMA is not enabled, CPU has to write/read each byte to/from the FIFO by itself. Usually this is faster than 2us, but the transaction length is limited to 32 bytes for both write and read.

    Typical transaction interval with one byte data is as below:

	Transaction Time
	Typical (us)
DMA	24
No DMA	22

2. SPI clock frequency: Each byte transferred takes 8 times of the clock period *8/fspi*. If the clock frequency is too high, some functions may be limited to use. See *Timing considerations*.
3. The cache miss: the default config puts only the ISR into the IRAM. Other SPI related functions including the driver itself and the callback may suffer from the cache miss and wait for some time while reading code from the flash. Select `CONFIG_SPI_MASTER_IN_IRAM` to put the whole SPI driver into IRAM, and put the entire callback(s) and its callee functions into IRAM to prevent this.

For a normal transaction, the overall cost is *20+8n/Fspi[MHz]* [us] for n bytes tranferred in one transaction. Hence the transferring speed is : *n/(20+8n/Fspi)*. Example of transferring speed under 8MHz clock speed:

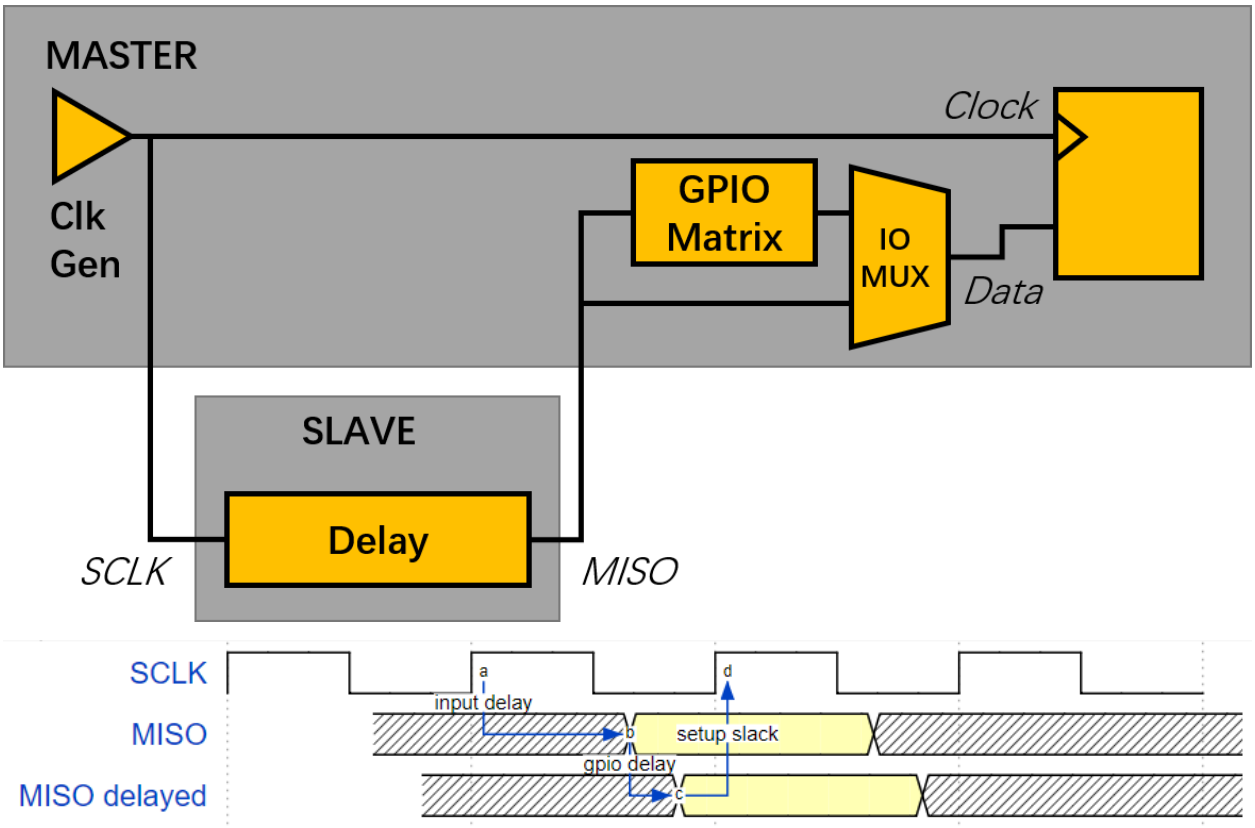
Frequency (MHz)	Transaction Interval (us)	Transaction Length (bytes)	Total Time (us)	Total Speed (kBps)
8	25	1	26	38.5
8	25	8	33	242.4
8	25	16	41	490.2
8	25	64	89	719.1
8	25	128	153	836.6

When the length of transaction is short, the cost of transaction interval is really high. Please try to squash data into one transaction if possible to get higher transfer speed.

BTW, the ISR is disabled during flash operation by default. To keep sending transactions during flash operations, enable `CONFIG_SPI_MASTER_ISR_IN_IRAM` and set `ESP_INTR_FLAG_IRAM` in the `intr_flags` member of `spi_bus_config_t`. Then all the transactions queued before the flash operations will be handled by the ISR continuously during flash operation. Note that the callback of each devices, and their callee functions, should be in the IRAM in this case, or your callback will crash due to cache miss.

Timing considerations

As shown in the figure below, there is a delay on the MISO signal after SCLK launch edge and before it's latched by the internal register. As a result, the MISO pin setup time is the limiting factor for SPI clock speed. When the delay is too large, setup slack is < 0 and the setup timing requirement is violated, leads to the failure of reading correctly.



The maximum frequency allowed is related to the *input delay* (maximum valid time after SCLK on the MISO bus), as well as the usage of GPIO matrix. The maximum frequency allowed is reduced to about 33~77% (related to existing *input delay*) when the GPIO matrix is used. To work at higher frequency, you have to use the IOMUX pins or the *dummy bit workaround*. You can get the maximum reading frequency of the master by `spi_get_freq_limit`.

Dummy bit workaround: We can insert dummy clocks (during which the host does not read data) before the read phase actually begins. The slave still sees the dummy clocks and gives out data, but the host does not read until the read phase. This compensates the lack of setup time of MISO required by the host, allowing the host reading at higher frequency.

In the ideal case (the slave is so fast that the input delay is shorter than an apb clock, 12.5ns), the maximum frequency host can read (or read and write) under different conditions is as below:

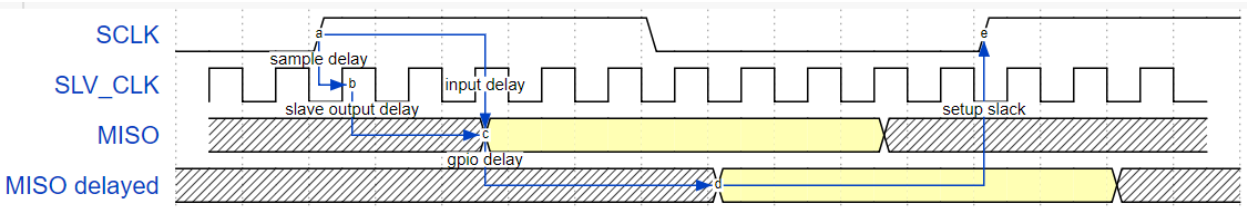
Frequency Limit (MHz)		Dummy Bits Used By Driver	Comments
GPIO matrix	IOMUX pins		
26.6	80	No	
40	–	Yes	Half Duplex, no DMA allowed

And if the host only writes, the *dummy bit workaround* is not used and the frequency limit is as below:

GPIO matrix (MHz)	IOMUX pins (MHz)
40	80

The spi master driver can work even if the *input delay* in the `spi_device_interface_config_t` is set to 0. However, setting a accurate value helps to: (1) calculate the frequency limit in full duplex mode, and (2) compensate the timing correctly by dummy bits in half duplex mode. You may find the maximum data valid time after the launch

edge of SPI clocks in the AC characteristics chapter of the device specifications, or measure the time on a oscilloscope or logic analyzer.



As shown in the figure above, the input delay is usually:

[input delay] = [sample delay] + [slave output delay]

1. The sample delay is the maximum random delay due to the asynchronization of SCLK and peripheral clock of the slave. It's usually 1 slave peripheral clock if the clock is asynchronize with SCLK, or 0 if the slave just use the SCLK to latch the SCLK and launch MISO data. e.g. for ESP32 slaves, the delay is 12.5ns (1 apb clock), while it is reduced to 0 if the slave is in the same chip as the master.
2. The slave output delay is the time for the MOSI to be stable after the launch edge. e.g. for ESP32 slaves, the output delay is 37.5ns (3 apb clocks) when IOMUX pins in the slave is used, or 62.5ns (5 apb clocks) if through the GPIO matrix.

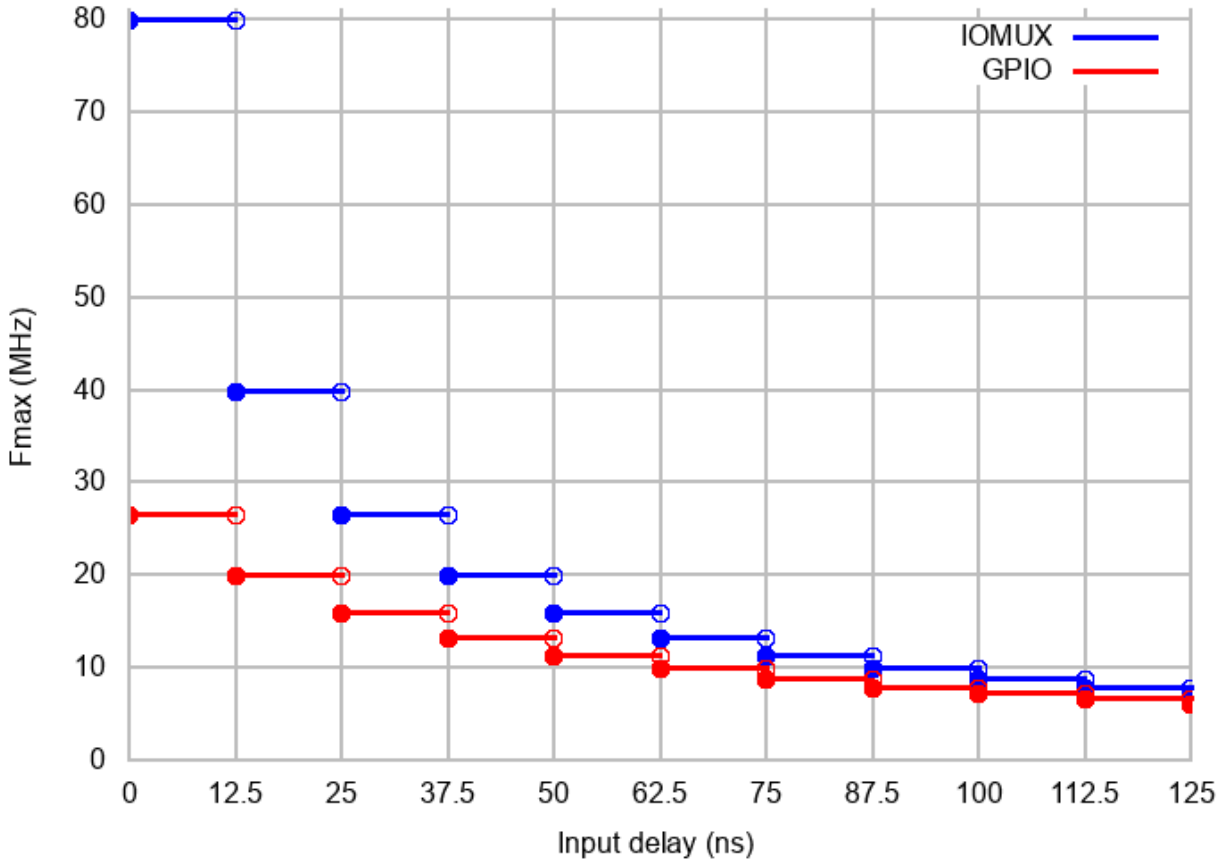
Some typical delays are shown in the following table:

Device	Input delay (ns)
Ideal device	0
ESP32 slave IOMUX*	50
ESP32 slave GPIO*	75
ESP32 slave is on an independent chip, 12.5ns sample delay included.	

The MISO path delay(tv), consists of slave *input delay* and master *GPIO matrix delay*, finally determines the frequency limit, above which the full duplex mode will not work, or dummy bits are used in the half duplex mode. The frequency limit is:

Freq limit[MHz] = 80 / (floor(MISO delay[ns]/12.5) + 1)

The figure below shows the relations of frequency limit against the input delay. 2 extra apb clocks should be counted into the MISO delay if the GPIO matrix in the master is used.



Corresponding frequency limit for different devices with different *input delay* are shown in the following table:

Master	Input delay (ns)	MISO path delay (ns)	Freq. limit (MHz)
IOMUX (0ns)	0	0	80
	50	50	16
	75	75	11.43
GPIO (25ns)	0	25	26.67
	50	75	11.43
	75	100	8.89

Thread Safety

The SPI driver API is thread safe when multiple SPI devices on the same bus are accessed from different tasks. However, the driver is not thread safe if the same SPI device is accessed from multiple tasks.

In this case, it is recommended to either refactor your application so only a single task accesses each SPI device, or to add mutex locking around access of the shared device.

Known Issues

1. Half duplex mode is not compatible with DMA when both writing and reading phases exist.

   If such transactions are required, you have to use one of the alternative solutions:

   1. use full-duplex mode instead.

2. disable the DMA by setting the last parameter to 0 in bus initialization function just as below: `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.

2. Full duplex mode is not compatible with the *dummy bit workaround*, hence the frequency is limited. See *dummy bit speed-up workaround*.

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 channnel.

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*, uint32_t *flags*, uint32_t *flags_o*)

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, set to ensure the pins set are capable with some functions:
  - `SPICOMMON_BUSFLAG_MASTER`: Initialize I/O in master mode
  - `SPICOMMON_BUSFLAG_SLAVE`: Initialize I/O in slave mode
  - `SPICOMMON_BUSFLAG_NATIVE_PINS`: Pins set should match the iomux pins of the controller.
  - `SPICOMMON_BUSFLAG_SCLK`, `SPICOMMON_BUSFLAG_MISO`, `SPICOMMON_BUSFLAG_MOSI`: Make sure SCLK/MISO/MOSI is/are set to a valid GPIO. Also check output capability according to the mode.
  - `SPICOMMON_BUSFLAG_DUAL`: Make sure both MISO and MOSI are output capable so that DIO mode is capable.
  - `SPICOMMON_BUSFLAG_WPHD` Make sure WP and HD are set to valid output GPIOs.
  - `SPICOMMON_BUSFLAG_QUAD`: Combination of `SPICOMMON_BUSFLAG_DUAL` and `SPICOMMON_BUSFLAG_WPHD`.
- `flags_o`: A SPICOMMON_BUSFLAG_* flag combination of bus abilities will be written to this address. Leave to NULL if not needed.
  - `SPICOMMON_BUSFLAG_NATIVE_PINS`: The bus is connected to iomux pins.
  - `SPICOMMON_BUSFLAG_SCLK`, `SPICOMMON_BUSFLAG_MISO`, `SPICOMMON_BUSFLAG_MOSI`: The bus has CLK/MISO/MOSI connected.
  - `SPICOMMON_BUSFLAG_DUAL`: The bus is capable with DIO mode.
  - `SPICOMMON_BUSFLAG_WPHD` The bus has WP and HD connected.
  - `SPICOMMON_BUSFLAG_QUAD`: Combination of `SPICOMMON_BUSFLAG_DUAL` and `SPICOMMON_BUSFLAG_WPHD`.

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

esp_err_t **spicommon_bus_free_io_cfg**(**const** *spi_bus_config_t* **bus_cfg*)

Free the IO used by a SPI peripheral.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Parameters

- `bus_cfg`: Bus config struct which defines which pins to be used.

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_cs_free_io**(int *cs_gpio_num*)

Free a chip select line.

Parameters

- `cs_gpio_num`: CS gpio num to free

void **spicommon_setup_dma_desc_links**(lldesc_t **dmadesc*, 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 `dmadesc`. 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 `dmadesc[0]` into DMA hardware results in the entirety `len` bytes of `data` being read or written.

Parameters

- `dmadesc`: 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 stuct pointer, pointed at the hardware registers

Parameters

- `host`: The SPI host

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 thsi will reset *both* DMA channels, and as such can only 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* refering to these lines will be ignored and can thus safely be left uninitialized.

Public Members

int **mosi_io_num**
GPIO pin for Master Out Slave In (=spi_d) signal, or -1 if not used.

int **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.

uint32_t **flags**
Abilities of bus to be checked by the driver. Or-ed value of `SPICOMMON_BUSFLAG_*` flags.

int **intr_flags**
Interrupt flag for the bus to set the priority, and IRAM attribute, see `esp_intr_alloc.h`. Note that the EDGE, INTRDISABLED attribute are ignored by the driver. Note that if ESP_INTR_FLAG_IRAM is set, ALL the callbacks of the driver, and their callee functions, should be put in the IRAM.

Macros

SPI_MAX_DMA_LEN

SPICOMMON_BUSFLAG_SLAVE
Initialize I/O in slave mode.

SPICOMMON_BUSFLAG_MASTER
Initialize I/O in master mode.

SPICOMMON_BUSFLAG_NATIVE_PINS
Check using iomux pins. Or indicates the pins are configured through the IO mux rather than GPIO matrix.

SPICOMMON_BUSFLAG_SCLK
Check existing of SCLK pin. Or indicates CLK line initialized.

SPICOMMON_BUSFLAG_MISO
Check existing of MISO pin. Or indicates MISO line initialized.

SPICOMMON_BUSFLAG_MOSI
Check existing of MOSI pin. Or indicates CLK line initialized.

SPICOMMON_BUSFLAG_DUAL
Check MOSI and MISO pins can output. Or indicates bus able to work under DIO mode.

SPICOMMON_BUSFLAG_WPHD
Check existing of WP and HD pins. Or indicates WP & HD pins initialized.

SPICOMMON_BUSFLAG_QUAD
Check existing of MOSI/MISO/WP/HD pins as output. Or indicates bus able to work under QIO mode.

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 transfered 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*, **const** *spi_device_interface_config_t *dev_config*, *spi_device_handle_t *handle*)

Allocate a device on a SPI bus.

This initializes the internal structures for a device, plus allocates a CS pin on the indicated SPI master peripheral and routes it to the indicated GPIO. All SPI master devices have three CS pins and can thus control up to three devices.

Note While in general, speeds up to 80MHz on the dedicated SPI pins and 40MHz on GPIO-matrix-routed pins are supported, full-duplex transfers routed over the GPIO matrix only support speeds up to 26MHz.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid

- ESP_ERR_NOT_FOUND if host doesn't have any free CS slots
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

Parameters

- `host`: SPI peripheral to allocate device on
- `dev_config`: SPI interface protocol config for the device
- `handle`: Pointer to variable to hold the device handle

esp_err_t **spi_bus_remove_device**(*spi_device_handle_t handle*)

Remove a device from the SPI bus.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if device already is freed
- ESP_OK on success

Parameters

- `handle`: Device handle to free

esp_err_t **spi_device_queue_trans**(*spi_device_handle_t handle*, *spi_transaction_t *trans_desc*, TickType_t *ticks_to_wait*)

Queue a SPI transaction for 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 succesfully completed. It will then return the description of the completed transaction so software can inspect the result and e.g. free the memory or re-use the buffers.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_TIMEOUT if there was no completed transaction before ticks_to_wait expired

- ESP_OK on success

Parameters

- `handle`: Device handle obtained using spi_host_add_dev
- `trans_desc`: Pointer to variable able to contain a pointer to the description of the transaction that is executed. The descriptor should not be modified until the descriptor is returned by spi_device_get_trans_result.
- `ticks_to_wait`: Ticks to wait until there's a returned item; use portMAX_DELAY to never time out.

esp_err_t **spi_device_transmit** (*spi_device_handle_t handle*, *spi_transaction_t *trans_desc*)

Send a SPI transaction, wait for it to complete, and return the result.

This function is the equivalent of calling spi_device_queue_trans() followed by spi_device_get_trans_result(). Do not use this when there is still a transaction separately queued from spi_device_queue_trans() that hasn't been finalized using spi_device_get_trans_result().

Note This function is not thread safe when multiple tasks access the same SPI device.

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

int **spi_cal_clock** (int *fapb*, int *hz*, int *duty_cycle*, uint32_t **reg_o*)

Calculate the working frequency that is most close to desired frequency, and also the register value.

Return Actual working frequency that most fit.

Parameters

- `fapb`: The frequency of apb clock, should be `APB_CLK_FREQ`.
- `hz`: Desired working frequency
- `duty_cycle`: Duty cycle of the spi clock
- `reg_o`: Output of value to be set in clock register, or NULL if not needed.

void **spi_get_timing** (bool *gpio_is_used*, int *input_delay_ns*, int *eff_clk*, int **dummy_o*, int **cycles_remain_o*)

Calculate the timing settings of specified frequency and settings.

Note If **dummy_o* is not zero, it means dummy bits should be applied in half duplex mode, and full duplex mode may not work.

Parameters

- `gpio_is_used`: True if using GPIO matrix, or False if iomux pins are used.
- `input_delay_ns`: Input delay from SCLK launch edge to MISO data valid.
- `eff_clk`: Effective clock frequency (in Hz) from spi_cal_clock.
- `dummy_o`: Address of dummy bits used output. Set to NULL if not needed.

- `cycles_remain_o`: Address of cycles remaining (after dummy bits are used) output.
  - -1 If too many cycles remaining, suggest to compensate half a clock.
  - 0 If no remaining cycles or dummy bits are not used.
  - positive value: cycles suggest to compensate.

int **spi_get_freq_limit** (bool *gpio_is_used*, int *input_delay_ns*)

Get the frequency limit of current configurations. SPI master working at this limit is OK, while above the limit, full duplex mode and DMA will not work, and dummy bits will be aplied in the half duplex mode.

Return Frequency limit of current configurations.

Parameters

- `gpio_is_used`: True if using GPIO matrix, or False if native pins are used.
- `input_delay_ns`: Input delay from SCLK launch edge to MISO data valid.

Structures

struct spi_device_interface_config_t

This is a configuration for a SPI slave device that is connected to one of the SPI buses.

Public Members

uint8_t **command_bits**

Default amount of bits in command phase (0-16), used when `SPI_TRANS_VARIABLE_CMD` is not used, otherwise ignored.

uint8_t **address_bits**

Default amount of bits in address phase (0-64), used when `SPI_TRANS_VARIABLE_ADDR` is not used, otherwise ignored.

uint8_t **dummy_bits**

Amount of dummy bits to insert between address and data phase.

uint8_t **mode**

SPI mode (0-3)

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, divisors of 80MHz, in Hz. See `SPI_MASTER_FREQ_*`.

int **input_delay_ns**

Maximum data valid time of slave. The time required between SCLK and MISO valid, including the possible clock delay from slave to master. The driver uses this value to give an extra delay before the

MISO is ready on the line. Leave at 0 unless you know you need a delay. For better timing performance at high frequency (over 8MHz), it's suggest to have the right value.

int **spics_io_num**
CS GPIO pin for this device, or -1 if not used.

uint32_t **flags**
Bitwise OR of SPI_DEVICE_* flags.

int **queue_size**
Transaction queue size. This sets how many transactions can be 'in the air' (queued using spi_device_queue_trans but not yet finished using spi_device_get_trans_result) at the same time.

transaction_cb_t **pre_cb**
Callback to be called before a transmission is started.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with ESP_INTR_FLAG_IRAM.

transaction_cb_t **post_cb**
Callback to be called after a transmission has completed.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with ESP_INTR_FLAG_IRAM.

struct spi_transaction_t
This structure describes one SPI transaction. The descriptor should not be modified until the transaction finishes.

Public Members

uint32_t **flags**
Bitwise OR of SPI_TRANS_* flags.

uint16_t **cmd**
Command data, of which the length is set in the `command_bits` of *spi_device_interface_config_t*.

NOTE: this field, used to be "command" in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF 3.0.

Example: write 0x0123 and command_bits=12 to send command 0x12, 0x3_ (in previous version, you may have to write 0x3_12).

uint64_t **addr**
Address data, of which the length is set in the `address_bits` of *spi_device_interface_config_t*.

NOTE: this field, used to be "address" in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF3.0.

Example: write 0x123400 and address_bits=24 to send address of 0x12, 0x34, 0x00 (in previous version, you may have to write 0x12340000).

size_t **length**
Total data length, in bits.

size_t **rxlength**
Total data length received, should be not greater than `length` in full-duplex mode (0 defaults this to the value of `length`).

void ***user**
User-defined variable. Can be used to store eg transaction ID.

const void ***tx_buffer**
Pointer to transmit buffer, or NULL for no MOSI phase.

uint8_t **tx_data**[4]
If SPI_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_MASTER_FREQ_8M
SPI master clock is divided by 80MHz apb clock. Below defines are example frequencies, and are accurate. Be free to specify a random frequency, it will be rounded to closest frequency (to macros below if above 8MHz). 8MHz

SPI_MASTER_FREQ_9M
8.89MHz

SPI_MASTER_FREQ_10M
10MHz

SPI_MASTER_FREQ_11M
11.43MHz

SPI_MASTER_FREQ_13M
13.33MHz

SPI_MASTER_FREQ_16M
16MHz

SPI_MASTER_FREQ_20M
20MHz

SPI_MASTER_FREQ_26M
26.67MHz

SPI_MASTER_FREQ_40M
40MHz

SPI_MASTER_FREQ_80M
80MHz

SPI_DEVICE_TXBIT_LSBFIRST
Transmit command/address/data LSB first instead of the default MSB first.

SPI_DEVICE_RXBIT_LSBFIRST
Receive data LSB first instead of the default MSB first.

SPI_DEVICE_BIT_LSBFIRST
Transmit and receive LSB first.

SPI_DEVICE_3WIRE
Use MOSI (=spid) for both sending and receiving data.

SPI_DEVICE_POSITIVE_CS
Make CS positive during a transaction instead of negative.

SPI_DEVICE_HALFDUPLEX
Transmit data before receiving it, instead of simultaneously.

SPI_DEVICE_CLK_AS_CS
Output clock on CS line if CS is active.

SPI_DEVICE_NO_DUMMY
There are timing issue when reading at high frequency (the frequency is related to whether iomux pins are used, valid time after slave sees the clock).

- In half-duplex mode, the driver automatically inserts dummy bits before reading phase to fix the timing issue. Set this flag to disable this feature.
- In full-duplex mode, however, the hardware cannot use dummy bits, so there is no way to prevent data being read from getting corrupted. Set this flag to confirm that you're going to work with output only, or read without dummy bits at your own risk.

SPI_TRANS_MODE_DIO
Transmit/receive data in 2-bit mode.

SPI_TRANS_MODE_QIO
Transmit/receive data in 4-bit mode.

SPI_TRANS_USE_RXDATA
Receive into rx_data member of *spi_transaction_t* instead into memory at rx_buffer.

SPI_TRANS_USE_TXDATA
Transmit tx_data member of *spi_transaction_t* instead of data at tx_buffer. Do not set tx_buffer when using this.

SPI_TRANS_MODE_DIOQIO_ADDR
Also transmit address in mode selected by SPI_MODE_DIO/SPI_MODE_QIO.

SPI_TRANS_VARIABLE_CMD
Use the `command_bits` in *`spi_transaction_ext_t`* rather than default value in *`spi_device_interface_config_t`*.

SPI_TRANS_VARIABLE_ADDR
Use the `address_bits` in *`spi_transaction_ext_t`* rather than default value in *`spi_device_interface_config_t`*.

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.

3.5.15 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
  - sclk - 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 quering the result using `spi_slave_get_trans_result`, or handle all requests synchroneously 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 succesfully 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.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with ESP_INTR_FLAG_IRAM.

slave_transaction_cb_t **post_trans_cb**
Callback called after a transaction is done.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with ESP_INTR_FLAG_IRAM.

struct spi_slave_transaction_t
This structure describes one SPI transaction

Public Members

size_t **length**
Total data length, in bits.

size_t **trans_len**
Transaction data length, in bits.

const void ***tx_buffer**
Pointer to transmit buffer, or NULL for no MOSI phase.

void ***rx_buffer**
Pointer to receive buffer, or NULL for no MISO phase.

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)

3.5.16 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.
- Once triggered the alarm will be disabled automatically and needs to be re-armed to trigger again.

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 be 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*)

Enable or disable generation of timer alarm events.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `group_num`: Timer group, 0 for TIMERG0 or 1 for TIMERG1

- `timer_num`: Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- `alarm_en`: To enable or disable timer alarm function.

esp_err_t **timer_isr_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.high|cnt.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

3.5.17 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.

In depth details of design of touch sensors and firmware development guidelines for the ESP32 are available in Touch Sensor Application Note. If you would like to test touch sensors in various configurations without building them on your own, check Guide for ESP32-Sense Development Kit.

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 rage 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.

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()`.

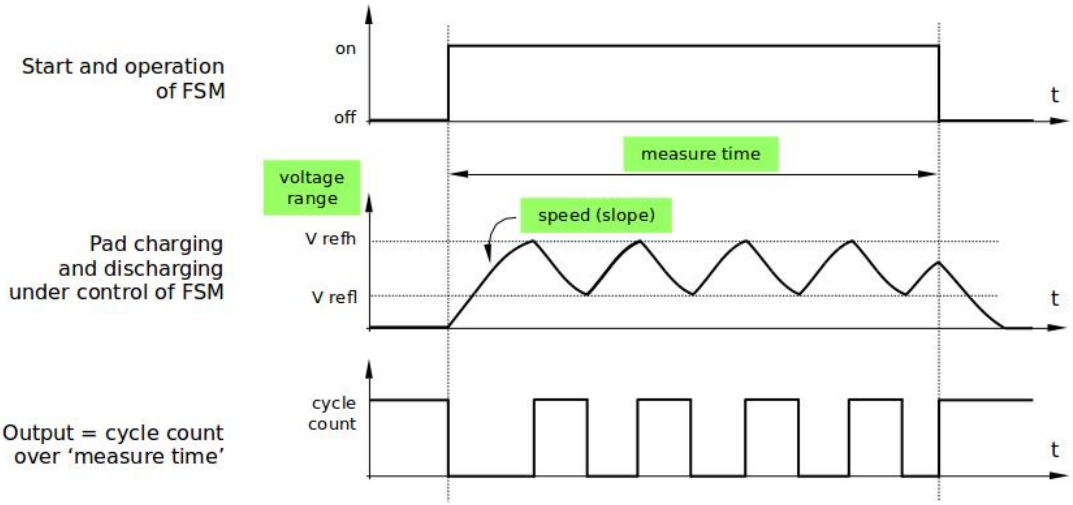


Fig. 11: Touch Pad - relationship between measurement parameters

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.

Note The default FSM mode is 'TOUCH_FSM_MODE_SW'. If you want to use interrupt trigger mode, then set it using function 'touch_pad_set_fsm_mode' to 'TOUCH_FSM_MODE_TIMER' after calling 'touch_pad_init'.

Return

- ESP_OK Success
- ESP_FAIL Touch pad init error

esp_err_t **touch_pad_deinit**()

Un-install touch pad driver.

Note After this function is called, other touch functions are prohibited from being called.

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.

Note If FSM mode is set to TOUCH_FSM_MODE_TIMER, this function will be blocked for one measurement cycle and wait for data to be valid.

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.

Note This API requests hardware measurement once. If IIR filter mode is enabled, please use 'touch_pad_read_raw_data' interface instead.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch pad parameter error
- ESP_ERR_INVALID_STATE This touch pad hardware connection is error, the value of "touch_value" is 0.
- 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 parameter error

- ESP_ERR_INVALID_STATE This touch pad hardware connection is error, the value of "touch_value" is 0.
- 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_raw_data** (*touch_pad_t touch_num*, uint16_t **touch_value*)

get raw data (touch sensor counter value) from IIR filter process. Need not request hardware measurements.

Note touch_pad_filter_start has to be called before calling touch_pad_read_raw_data. This function can be called from ISR

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch pad parameter error
- ESP_ERR_INVALID_STATE This touch pad hardware connection is error, the value of "touch_value" is 0.
- 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_set_filter_read_cb** (*filter_cb_t read_cb*)

Register the callback function that is called after each IIR filter calculation.

Note The 'read_cb' callback is called in timer task in each filtering cycle.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG set error

Parameters

- `read_cb`: Pointer to filtered callback function. If the argument passed in is NULL, the callback will stop.

esp_err_t **touch_pad_isr_handler_register** (void (**fn*)) void *

, void **arg*, int *unused*, *intr_handle_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
- ESP_ERR_NO_MEM No memory

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
- ESP_ERR_NO_MEM No memory

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 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 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

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

esp_err_t **touch_pad_get_wakeup_status**(*touch_pad_t* **pad_num*)

Get the touch pad which caused wakeup from sleep.

Return

- ESP_OK Success
- ESP_FAIL get status err

Parameters

- `pad_num`: pointer to touch pad which caused wakeup

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 0x7fff

TOUCH_PAD_MEASURE_WAIT_DEFAULT
The timer frequency is 8Mhz, the max value is 0xff

TOUCH_FSM_MODE_DEFAULT
The touch FSM my 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**

typedef void (*__filter_cb_t__) (uint16_t *raw_value, uint16_t *filtered_value)
Callback function that is called after each IIR filter calculation.

Note This callback is called in timer task in each filtering cycle.

Note This callback should not be blocked.

Parameters

- `raw_value`: The latest raw data(touch sensor counter value) that points to all channels(raw_value[0..TOUCH_PAD_MAX-1]).
- `filtered_value`: The latest IIR filtered data(calculated from raw data) that points to all channels(filtered_value[0..TOUCH_PAD_MAX-1]).

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

3.5.18 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 tree 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 cpp:func:*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 patern detection interrupts: peripherals/uart_events.

Transmitting and receiveing with the same UART in two separate FreeRTOS tasks: peripherals/uart_async_rxtxtasks.

Using synchronous I/O multiplexing for UART file descriptors: peripherals/uart_select.

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*, int *tx_io_num*, int *rx_io_num*, int *rts_io_num*, int *cts_io_num*)

Set UART pin number.

Note Internal signal can be output to multiple GPIO pads. Only one GPIO pad can connect with input signal.

Note Instead of GPIO number a macro 'UART_PIN_NO_CHANGE' may be provided to keep the currently allocated pin.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART_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 (>=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 (>=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 (>=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 duration(unit: the time it takes to send one bit at current baudrate)

int **`uart_read_bytes`** (*uart_port_t uart_num*, uint8_t **buf*, uint32_t *length*, TickType_t *ticks_to_wait*)

UART read bytes from UART buffer.

Return

- (-1) Error
- OTHERS (>=0) The number of bytes read from UART FIFO

Parameters

- `uart_num`: UART_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.

3.6 Protocols API

3.6.1 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"}
```

(continues on next page)

(continued from previous page)

```
    };
    //set txt data for service (will free and replace current data)
    mdns_service_txt_set("_http", "_tcp", serviceTxtData, 3);

    //change service port
    mdns_service_port_set("_myservice", "_udp", 4321);
}
```

mDNS Query

mDNS provides methods for browsing for services and resolving host's IP/IPv6 addresses.

Results for services are returned as a linked list of `mdns_result_t` objects.

Example method to resolve host IPs:

```
void resolve_mdns_host(const char * host_name)
{
    printf("Query A: %s.local", host_name);

    struct ip4_addr addr;
    addr.addr = 0;

    esp_err_t err = mdns_query_a(host_name, 2000,  &addr);
    if(err){
        if(err == ESP_ERR_NOT_FOUND){
            printf("Host was not found!");
            return;
        }
        printf("Query Failed");
        return;
    }

    printf(IPSTR, IP2STR(&addr));
}
```

Example method to resolve local services:

```
static const char * if_str[] = {"STA", "AP", "ETH", "MAX"};
static const char * ip_protocol_str[] = {"V4", "V6", "MAX"};

void mdns_print_results(mdns_result_t * results){
    mdns_result_t * r = results;
    mdns_ip_addr_t * a = NULL;
    int i = 1, t;
    while(r){
        printf("%d: Interface: %s, Type: %s\n", i++, if_str[r->tcpip_if], ip_protocol_
↪str[r->ip_protocol]);
        if(r->instance_name){
            printf("  PTR : %s\n", r->instance_name);
        }
        if(r->hostname){
            printf("  SRV : %s.local:%u\n", r->hostname, r->port);
        }
        if(r->txt_count){
```

(continues on next page)

(continued from previous page)

```
            printf("  TXT : [%u] ", r->txt_count);
            for(t=0; t<r->txt_count; t++){
                printf("%s=%s; ", r->txt[t].key, r->txt[t].value);
            }
            printf("\n");
        }
        a = r->addr;
        while(a){
            if(a->addr.type == 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 A record.

Return

- ESP_OK success
- ESP_ERR_INVALID_STATE mDNS is not running
- ESP_ERR_NO_MEM memory error
- ESP_ERR_INVALID_ARG parameter error

Parameters

- `host_name`: host name to look for
- `timeout`: time in milliseconds to wait for answer. If 0, max_results needs to be defined
- `addr`: pointer to the resulting IP6 address

esp_err_t **mdns_handle_system_event**(void **ctx*, system_event_t **event*)

System event handler This method controls the service state on all active interfaces and applications are required to call it from the system event handler for normal operation of mDNS service.

Parameters

- `ctx`: The system event context
- `event`: The system event

Structures

struct mdns_txt_item_t

mDNS basic text item structure Used in mdns_service_add()

Public Members

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 addreses 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

3.6.2 ESP-TLS

Overview

The ESP-TLS component provides a simplified API interface for accessing the commonly used TLS functionality. It supports common scenarios like CA certification validation, SNI, ALPN negotiation, non-blocking connection among others. All the configuration can be specified in the esp_tls_cfg_t data structure. Once done, TLS communication can be conducted using the following APIs: * esp_tls_conn_new(): for opening a new TLS connection * esp_tls_conn_read/write(): for reading/writing from the connection * esp_tls_conn_delete(): for freeing up the connection Any application layer protocol like HTTP1, HTTP2 etc can be executed on top of this layer.

Application Example

Simple HTTPS example that uses ESP-TLS to establish a secure socket connection: protocols/https_request.

API Reference

Header File

- esp-tls/esp_tls.h

Functions

esp_tls_t ***esp_tls_conn_new*** (**const** char **hostname*, int *hostlen*, int *port*, **const** *esp_tls_cfg_t* **cfg*)

Create a new TLS/SSL connection.

This function establishes a TLS/SSL connection with the specified host.

Return pointer to esp_tls_t, or NULL if connection couldn't be opened.

Parameters

- `hostname`: Hostname of the host.
- `hostlen`: Length of hostname.
- `port`: Port number of the host.
- `cfg`: TLS configuration as esp_tls_cfg_t. If you wish to open non-TLS connection, keep this NULL. For TLS connection, a pass pointer to esp_tls_cfg_t. At a minimum, this structure should be zero-initialized.

esp_tls_t ***esp_tls_conn_http_new*** (**const** char **url*, **const** *esp_tls_cfg_t* **cfg*)

Create a new TLS/SSL connection with a given "HTTP" url.

The behaviour is same as esp_tls_conn_new() API. However this API accepts host's url.

Return pointer to esp_tls_t, or NULL if connection couldn't be opened.

Parameters

- `url`: url of host.
- `cfg`: TLS configuration as esp_tls_cfg_t. If you wish to open non-TLS connection, keep this NULL. For TLS connection, a pass pointer to 'esp_tls_cfg_t'. At a minimum, this structure should be zero-initialized.

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

Write from buffer 'data' into specified tls connection.

Return

- >0 if write operation was successful, the return value is the number of bytes actually written to the TLS/SSL connection.
- 0 if write operation was not successful. The underlying connection was closed.
- <0 if write operation was not successful, because either an error occured or an action must be taken by the calling process.

Parameters

- `tls`: pointer to esp-tls as esp-tls handle.
- `data`: Buffer from which data will be written.

- `datalen`: Length of data buffer.

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

Read from specified tls connection into the buffer 'data'.

Return

- >0 if read operation was successful, the return value is the number of bytes actually read from the TLS/SSL connection.
- 0 if read operation was not successful. The underlying connection was closed.
- <0 if read operation was not successful, because either an error occured or an action must be taken by the calling process.

Parameters

- `tls`: pointer to esp-tls as esp-tls handle.
- `data`: Buffer to hold read data.
- `datalen`: Length of data buffer.

void **esp_tls_conn_delete** (*esp_tls_t* **tls*)

Close the TLS/SSL connection and free any allocated resources.

This function should be called to close each tls connection opened with esp_tls_conn_new() or esp_tls_conn_http_new() APIs.

Parameters

- `tls`: pointer to esp-tls as esp-tls handle.

Structures

struct esp_tls_cfg

ESP-TLS configuration parameters.

Public Members

const char ****alpn_protos**

Application protocols required for HTTP2. If HTTP2/ALPN support is required, a list of protocols that should be negotiated. The format is length followed by protocol name. For the most common cases the following is ok: "\x02h2"

- where the first '2' is the length of the protocol and
- the subsequent 'h2' is the protocol name

const unsigned char ***cacert_pem_buf**

Certificate Authority's certificate in a buffer

unsigned int **cacert_pem_bytes**

Size of Certificate Authority certificate pointed to by cacert_pem_buf

bool **non_block**

Configure non-blocking mode. If set to true the underneath socket will be configured in non blocking mode after tls session is established

struct esp_tls
ESP-TLS Connection Handle.

Public Members

mbedtls_ssl_context **ssl**
TLS/SSL context

mbedtls_entropy_context **entropy**
mbedTLS entropy context structure

mbedtls_ctr_drbg_context **ctr_drbg**
mbedTLS ctr drbg context structure. CTR_DRBG is deterministic random bit generation based on AES-256

mbedtls_ssl_config **conf**
TLS/SSL configuration to be shared between mbedtls_ssl_context structures

mbedtls_net_context **server_fd**
mbedTLS wrapper type for sockets

mbedtls_x509_crt **cacert**
Container for an X.509 certificate

int **sockfd**
Underlying socket file descriptor.

ssize_t (***read**) (**struct** *esp_tls* *tls, char *data, size_t datalen)
Callback function for reading data from TLS/SSL connection.

ssize_t (***write**) (**struct** *esp_tls* *tls, **const** char *data, size_t datalen)
Callback function for writing data to TLS/SSL connection.

Type Definitions

typedef struct *esp_tls_cfg* **esp_tls_cfg_t**
ESP-TLS configuration parameters.

typedef struct *esp_tls* **esp_tls_t**
ESP-TLS Connection Handle.

3.6.3 ESP HTTP Client

Overview

`esp_http_client` provides an API for making HTTP/S requests from ESP-IDF programs. The steps to use this API for an HTTP request are:

- *`esp_http_client_init()`*: To use the HTTP client, the first thing we must do is create an `esp_http_client` by pass into this function with the *`esp_http_client_config_t`* configurations. Which configuration values we do not define, the library will use default.
- *`esp_http_client_perform()`*: The `esp_http_client` argument created from the init function is needed. This function performs all operations of the esp_http_client, from opening the connection, sending data, downloading data and closing the connection if necessary. All related events will be invoked in the event_handle (defined by *`esp_http_client_config_t`*). This function performs its job and blocks the current task until it's done

- *esp_http_client_cleanup()*: After completing our **esp_http_client's** task, this is the last function to be called. It will close the connection (if any) and free up all the memory allocated to the HTTP client

Application Example

```
esp_err_t _http_event_handle(esp_http_client_event_t *evt)
{
    switch(evt->event_id) {
        case HTTP_EVENT_ERROR:
            ESP_LOGI(TAG, "HTTP_EVENT_ERROR");
            break;
        case HTTP_EVENT_ON_CONNECTED:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_CONNECTED");
            break;
        case HTTP_EVENT_HEADER_SENT:
            ESP_LOGI(TAG, "HTTP_EVENT_HEADER_SENT");
            break;
        case HTTP_EVENT_ON_HEADER:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_HEADER");
            printf("%.*s", evt->data_len, (char*)evt->data);
            break;
        case HTTP_EVENT_ON_DATA:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_DATA, len=%d", evt->data_len);
            if (!esp_http_client_is_chunked_response(evt->client)) {
                printf("%.*s", evt->data_len, (char*)evt->data);
            }

            break;
        case HTTP_EVENT_ON_FINISH:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_FINISH");
            break;
        case HTTP_EVENT_DISCONNECTED:
            ESP_LOGI(TAG, "HTTP_EVENT_DISCONNECTED");
            break;
    }
    return ESP_OK;
}

esp_http_client_config_t config = {
   .url = "http://httpbin.org/redirect/2",
   .event_handle = _http_event_handle,
};
esp_http_client_handle_t client = esp_http_client_init(&config);
esp_err_t err = esp_http_client_perform(client);

if (err == ESP_OK) {
   ESP_LOGI(TAG, "Status = %d, content_length = %d",
           esp_http_client_get_status_code(client),
           esp_http_client_get_content_length(client));
}
esp_http_client_cleanup(client);
```

Persistent Connections

Persistent connections means that the HTTP client can re-use the same connection for several transfers. If the server does not request to close the connection with the `Connection: close` header, the new transfer with sample ip

address, port, and protocol.

To allow the HTTP client to take full advantage of persistent connections, you should do as many of your file transfers as possible using the same handle.

Persistent Connections example

```
esp_err_t err;
esp_http_client_config_t config = {
    .url = "http://httpbin.org/get",
};
esp_http_client_handle_t client = esp_http_client_init(&config);
// first request
err = esp_http_client_perform(client);

// second request
esp_http_client_set_url(client, "http://httpbin.org/anything")
esp_http_client_set_method(client, HTTP_METHOD_DELETE);
esp_http_client_set_header(client, "HeaderKey", "HeaderValue");
err = esp_http_client_perform(client);

esp_http_client_cleanup(client);
```

HTTPS

The HTTP client supports SSL connections using **mbedtls**, with the **url** configuration starting with `https` scheme (or `transport_type = HTTP_TRANSPORT_OVER_SSL`). HTTPS support can be configured via :ref:CONFIG_ENABLE_HTTPS (enabled by default)..

Note: By providing information using HTTPS, the library will use the SSL transport type to connect to the server. If you want to verify server, then need to provide additional certificate in PEM format, and provide to `cert_pem` in `esp_http_client_config_t`

HTTPS example

```
static void https()
{
    esp_http_client_config_t config = {
        .url = "https://www.howsmyssl.com",
        .cert_pem = howsmyssl_com_root_cert_pem_start,
    };
    esp_http_client_handle_t client = esp_http_client_init(&config);
    esp_err_t err = esp_http_client_perform(client);

    if (err == ESP_OK) {
        ESP_LOGI(TAG, "Status = %d, content_length = %d",
                esp_http_client_get_status_code(client),
                esp_http_client_get_content_length(client));
    }
    esp_http_client_cleanup(client);
}
```

HTTP Stream

Some applications need to open the connection and control the reading of the data in an active manner. the HTTP client supports some functions to make this easier, of course, once you use these functions you should not use the *esp_http_client_perform()* function with that handle, and *esp_http_client_init()* alway to called first to get the handle. Perform that functions in the order below:

- *esp_http_client_init()*: to create and handle
- esp_http_client_set_* or esp_http_client_delete_*: to modify the http connection information (optional)
- *esp_http_client_open()*: Open the http connection with write_len parameter, write_len=0 if we only need read
- *esp_http_client_write()*: Upload data, max length equal to write_len of *esp_http_client_open()* function. We may not need to call it if write_len=0
- *esp_http_client_fetch_headers()*: After sending the headers and write data (if any) to the server, this function will read the HTTP Server response headers. Calling this function will return the content-length from the Server, and we can call *esp_http_client_get_status_code()* for the HTTP status of the connection.
- *esp_http_client_read()*: Now, we can read the HTTP stream by this function.
- *esp_http_client_close()*: We should the connection after finish
- *esp_http_client_cleanup()*: And release the resources

Perform HTTP request as Stream reader

Check the example function http_perform_as_stream_reader at protocols/esp_http_client.

HTTP Authentication

The HTTP client supports both **Basic** and **Digest** Authentication. By providing usernames and passwords in url or in the username, password of config entry. And with auth_type = HTTP_AUTH_TYPE_BASIC, the HTTP client takes only 1 perform to pass the authentication process. If auth_type = HTTP_AUTH_TYPE_NONE, but there are username and password in the configuration, the HTTP client takes 2 performs. The first time it connects to the server and receives the UNAUTHORIZED header. Based on this information, it will know which authentication method to choose, and perform it on the second.

Config authentication example with URI

```
esp_http_client_config_t config = {
    .url = "http://user:passwd@httpbin.org/basic-auth/user/passwd",
    .auth_type = HTTP_AUTH_TYPE_BASIC,
};
```

Config authentication example with username, password entry

```
esp_http_client_config_t config = {
    .url = "http://httpbin.org/basic-auth/user/passwd",
    .username = "user",
    .password = "passwd",
    .auth_type = HTTP_AUTH_TYPE_BASIC,
};
```

HTTP Client example: protocols/esp_http_client.

API Reference

Header File

- esp_http_client/include/esp_http_client.h

Functions

esp_http_client_handle_t **esp_http_client_init**(**const** *esp_http_client_config_t* **config*)

Start a HTTP session This function must be the first function to call, and it returns a esp_http_client_handle_t that you must use as input to other functions in the interface. This call MUST have a corresponding call to esp_http_client_cleanup when the operation is complete.

Return

- `esp_http_client_handle_t`
- NULL if any errors

Parameters

- `config`: The configurations, see `http_client_config_t`

esp_err_t **esp_http_client_perform**(*esp_http_client_handle_t client*)

Invoke this function after `esp_http_client_init` and all the options calls are made, and will perform the transfer as described in the options. It must be called with the same esp_http_client_handle_t as input as the esp_http_client_init call returned. esp_http_client_perform performs the entire request in a blocking manner and returns when done, or if it failed. You can do any amount of calls to esp_http_client_perform while using the same esp_http_client_handle_t. The underlying connection may be kept open if the server allows it. If you intend to transfer more than one file, you are even encouraged to do so. esp_http_client will then attempt to re-use the same connection for the following transfers, thus making the operations faster, less CPU intense and using less network resources. Just note that you will have to use `esp_http_client_set_**` between the invokes to set options for the following esp_http_client_perform.

Note You must never call this function simultaneously from two places using the same client handle. Let the function return first before invoking it another time. If you want parallel transfers, you must use several esp_http_client_handle_t. This function include `esp_http_client_open` -> `esp_http_client_write` -> `esp_http_client_fetch_headers` -> `esp_http_client_read` (and option) `esp_http_client_close`.

Return

- ESP_OK on successful
- ESP_FAIL on error

Parameters

- `client`: The esp_http_client handle

esp_err_t **esp_http_client_set_url**(*esp_http_client_handle_t client*, **const** char **url*)

Set URL for client, when performing this behavior, the options in the URL will replace the old ones.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `client`: The esp_http_client handle
- `url`: The url

esp_err_t **esp_http_client_set_post_field**(*esp_http_client_handle_t client*, **const** char **data*, int *len*)

Set post data, this function must be called before `esp_http_client_finalize_open` or perform Note: The data parameter passed to this function is a pointer and this function will not copy the data.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `client`: The esp_http_client handle
- `data`: post data pointer
- `len`: post length

int **esp_http_client_get_post_field**(*esp_http_client_handle_t client*, char ***data*)

Get current post field information.

Return Size of post data

Parameters

- `client`: The client
- `data`: Point to post data pointer

esp_err_t **esp_http_client_set_header**(*esp_http_client_handle_t client*, **const** char **key*, **const** char **value*)

Set http request header, this function must be called after esp_http_client_init and before any perform function.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `client`: The esp_http_client handle
- `key`: The header key
- `value`: The header value

esp_err_t **esp_http_client_set_method**(*esp_http_client_handle_t client*, *esp_http_client_method_t method*)

Set http request method.

Return ESP_OK

Parameters

- `client`: The esp_http_client handle
- `method`: The method

esp_err_t **esp_http_client_delete_header**(*esp_http_client_handle_t client*, **const** char **key*)

Delete http request header.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `client`: The esp_http_client handle
- `key`: The key

esp_err_t **esp_http_client_open**(*esp_http_client_handle_t client*, int *write_len*)

This function will be open the connection, write all header strings and return.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `client`: The esp_http_client handle
- `write_len`: HTTP Content length need to write to the server

int **esp_http_client_write**(*esp_http_client_handle_t client*, **const** char **buffer*, int *len*)

This function will write data to the HTTP connection previously opened by esp_http_client_open()

Return

- (-1) if any errors
- Length of data written

Parameters

- `client`: The esp_http_client handle
- `buffer`: The buffer
- `len`: This value must not be larger than the write_len parameter provided to esp_http_client_open()

int **esp_http_client_fetch_headers**(*esp_http_client_handle_t client*)

This function need to call after esp_http_client_open, it will read from http stream, process all receive headers.

Return

- (0) if stream doesn't contain content-length header, or chunked encoding (checked by `esp_http_client_is_chunked` response)
- (-1: ESP_FAIL) if any errors
- Download data length defined by content-length header

Parameters

- `client`: The esp_http_client handle

bool **`esp_http_client_is_chunked_response`**(*esp_http_client_handle_t client*)

Check response data is chunked, must call after `esp_http_client_finalize_open`

Return true or false

Parameters

- `client`: The esp_http_client handle

int **`esp_http_client_read`**(*esp_http_client_handle_t client*, char **buffer*, int *len*)

Read data from http stream.

Return

- (-1) if any errors
- Length of data was read

Parameters

- `client`: The esp_http_client handle
- `buffer`: The buffer
- `len`: The length

int **`esp_http_client_get_status_code`**(*esp_http_client_handle_t client*)

Get http response status code, the valid value if this function invoke after `esp_http_client_perform` or `esp_http_client_finalize_open`

Return Status code

Parameters

- `client`: The esp_http_client handle

int **`esp_http_client_get_content_length`**(*esp_http_client_handle_t client*)

Get http response content length (from header Content-Length) the valid value if this function invoke after `esp_http_client_perform` or `esp_http_client_finalize_open`

Return

- (-1) Chunked transfer
- Content-Length value as bytes

Parameters

- `client`: The esp_http_client handle

esp_err_t **`esp_http_client_close`**(*esp_http_client_handle_t client*)

Close http connection, still kept all http request resources.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `client`: The esp_http_client handle

esp_err_t **esp_http_client_cleanup**(*esp_http_client_handle_t client*)

This function must be the last function to call for an session. It is the opposite of the esp_http_client_init function and must be called with the same handle as input that a esp_http_client_init call returned. This might close all connections this handle has used and possibly has kept open until now. Don't call this function if you intend to transfer more files, re-using handles is a key to good performance with esp_http_client.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `client`: The esp_http_client handle

esp_http_client_transport_t **esp_http_client_get_transport_type**(*esp_http_client_handle_t client*)

Get transport type.

Return

- HTTP_TRANSPORT_UNKNOWN
- HTTP_TRANSPORT_OVER_TCP
- HTTP_TRANSPORT_OVER_SSL

Parameters

- `client`: The esp_http_client handle

Structures

struct esp_http_client_event

HTTP Client events data.

Public Members

esp_http_client_event_id_t **event_id**

event_id, to know the cause of the event

esp_http_client_handle_t **client**

esp_http_client_handle_t context

void ***data**

data of the event

int **data_len**

data length of data

void ***user_data**
user_data context, from *esp_http_client_config_t* user_data

char ***header_key**
For HTTP_EVENT_ON_HEADER event_id, it's store current http header key

char ***header_value**
For HTTP_EVENT_ON_HEADER event_id, it's store current http header value

struct esp_http_client_config_t
HTTP configuration.

Public Members

const char ***url**
HTTP URL, the information on the URL is most important, it overrides the other fields below, if any

const char ***host**
Domain or IP as string

int **port**
Port to connect, default depend on esp_http_client_transport_t (80 or 443)

const char ***username**
Using for Http authentication

const char ***password**
Using for Http authentication

esp_http_client_auth_type_t **auth_type**
Http authentication type, see `esp_http_client_auth_type_t`

const char ***path**
HTTP Path, if not set, default is `/`

const char ***query**
HTTP query

const char ***cert_pem**
SSL Certification, PEM format as string, if the client requires to verify server

esp_http_client_method_t **method**
HTTP Method

int **timeout_ms**
Network timeout in milliseconds

bool **disable_auto_redirect**
Disable HTTP automatic redirects

int **max_redirection_count**
Max redirection number, using default value if zero

http_event_handle_cb **event_handler**
HTTP Event Handle

esp_http_client_transport_t **transport_type**
HTTP transport type, see `esp_http_client_transport_t`

int **buffer_size**
HTTP buffer size (both send and receive)

void ***user_data**
HTTP user_data context

Macros

DEFAULT_HTTP_BUF_SIZE

ESP_ERR_HTTP_BASE
Starting number of HTTP error codes

ESP_ERR_HTTP_MAX_REDIRECT
The error exceeds the number of HTTP redirects

ESP_ERR_HTTP_CONNECT
Error open the HTTP connection

ESP_ERR_HTTP_WRITE_DATA
Error write HTTP data

ESP_ERR_HTTP_FETCH_HEADER
Error read HTTP header from server

ESP_ERR_HTTP_INVALID_TRANSPORT
There are no transport support for the input scheme

Type Definitions

typedef struct esp_http_client ***esp_http_client_handle_t**

typedef struct *esp_http_client_event* ***esp_http_client_event_handle_t**

typedef struct *esp_http_client_event* **esp_http_client_event_t**
HTTP Client events data.

typedef *esp_err_t* (***http_event_handle_cb**) (*esp_http_client_event_t* *evt)

Enumerations

enum esp_http_client_event_id_t
HTTP Client events id.

Values:

HTTP_EVENT_ERROR = 0
This event occurs when there are any errors during execution

HTTP_EVENT_ON_CONNECTED
Once the HTTP has been connected to the server, no data exchange has been performed

HTTP_EVENT_HEADER_SENT
After sending all the headers to the server

HTTP_EVENT_ON_HEADER
Occurs when receiving each header sent from the server

HTTP_EVENT_ON_DATA
Occurs when receiving data from the server, possibly multiple portions of the packet

HTTP_EVENT_ON_FINISH
Occurs when finish a HTTP session

HTTP_EVENT_DISCONNECTED
The connection has been disconnected

enum esp_http_client_transport_t
HTTP Client transport.

Values:

HTTP_TRANSPORT_UNKNOWN = 0x0
Unknown

HTTP_TRANSPORT_OVER_TCP
Transport over tcp

HTTP_TRANSPORT_OVER_SSL
Transport over ssl

enum esp_http_client_method_t
HTTP method.

Values:

HTTP_METHOD_GET = 0
HTTP GET Method

HTTP_METHOD_POST
HTTP POST Method

HTTP_METHOD_PUT
HTTP PUT Method

HTTP_METHOD_PATCH
HTTP PATCH Method

HTTP_METHOD_DELETE
HTTP DELETE Method

HTTP_METHOD_MAX

enum esp_http_client_auth_type_t
HTTP Authentication type.

Values:

HTTP_AUTH_TYPE_NONE = 0
No authention

HTTP_AUTH_TYPE_BASIC
HTTP Basic authentication

HTTP_AUTH_TYPE_DIGEST
HTTP Disgest authentication

Example code for this API section is provided in protocols directory of ESP-IDF examples.

3.7 Storage API

3.7.1 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 setting *`CONFIG_ESPTOOLPY_FLASHSIZE`* in `make menuconfig`.

If it is necessary to override the configured flash size at runtime, is 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.

If you use the APIs documented here, then this happens automatically and transparently. However note that it will have some performance impact on other tasks in the system.

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

Note: Most application code should use these `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 transparenlty 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 any 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 For fastest write performance, write a 4 byte aligned size at a 4 byte aligned offset in flash from a source buffer in DRAM. Varying any of these parameters will still work, but will be slower due to buffering.

Note Writing more than 8KB at a time will be split into multiple write operations to avoid disrupting other tasks in the system.

Return esp_err_t

Parameters

- `dest_addr`: Destination address in Flash.
- `src`: Pointer to the source buffer.
- `size`: Length of data, in 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.

Note For fastest read performance, all parameters should be 4 byte aligned. If source address and read size are not 4 byte aligned, read may be split into multiple flash operations. If destination buffer is not 4 byte aligned, a temporary buffer will be allocated on the stack.

Note Reading more than 16KB of data at a time will be split into multiple reads to avoid disruption to other tasks in the system. Consider using spi_flash_mmap() to read large amounts of data.

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 esp_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 64kB MMU pages and configures them to map the requested region of flash memory into the address space. It may reuse MMU pages which already provide the required mapping.

As with any allocator, if mmap/munmap are heavily used then the address space may become fragmented. To troubleshoot issues with page allocation, use spi_flash_mmap_dump() function.

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 to be mapped. This size will be rounded up to a 64kB boundary
- `memory`: Address space where the region should be mapped (data or instruction)
- `out_ptr`: 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**(**const** int **pages*, size_t *page_count*, *spi_flash_mmap_memory_t* *memory*, **const** void ***out_ptr*, *spi_flash_mmap_handle_t* **out_handle*)

Map sequences of pages of flash memory into data or instruction address space.

This function allocates sufficient number of 64kB MMU pages and configures them to map the indicated pages of flash memory contiguously into address space. In this respect, it works in a similar way as spi_flash_mmap() but it allows mapping a (maybe non-contiguous) set of pages into a contiguous region of memory.

Return

- ESP_OK on success
- ESP_ERR_NO_MEM if pages can not be allocated
- ESP_ERR_INVALID_ARG if pagecount is zero or pages array is not in internal memory

Parameters

- `pages`: An array of numbers indicating the 64kB pages in flash to be mapped contiguously into memory. These indicate the indexes of the 64kB pages, not the byte-size addresses as used in other functions. Array must be located in internal memory.
- `page_count`: Number of entries in the pages array
- `memory`: Address space where the region should be mapped (instruction or data)
- `out_ptr`: 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 number of free pages available in mmu table. This could be useful before calling actual spi_flash_mmap (maps flash range to DCache or ICache memory) to check if there is sufficient space available for mapping.

Return number of free pages which can be mmaped

Parameters

- `memory`: memory 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 memory mapped flash 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`: Address space type to look up a flash cache address mapping for (instruction or data)

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. 3) When CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_ALLOWED is disabled, flash writing/erasing API checks for addresses provided by user to avoid corruption of critical flash regions (bootloader, partition table, running application etc.).

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.

spi_flash_is_safe_write_address_t **is_safe_write_address**
checks flash write addresses.

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.

typedef bool (*__spi_flash_is_safe_write_address_t__)(size_t addr, size_t size)
Function to protect SPI flash critical regions corruption.

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_NVS_KEYS = 0x04
Partition for NVS keys.

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

3.7.2 SD/SDIO/MMC Driver

Overview

SD/SDIO/MMC driver currently supports SD memory and IO cards. Support for MCC/eMMC cards will be added in the future. This protocol level driver builds on top of SDMMC and SD SPI host drivers.

SDMMC and SD SPI host drivers (`driver/sdmmc_host.h`) provide APIs to send commands to the slave device(s), send and receive data, and handle error conditions on the bus.

- See *SDMMC Host API* for functions used to initialize and configure SDMMC host.
- See *SD SPI Host API* for functions used to initialize and configure SD SPI host.

SDMMC protocol layer (`sdmmc_cmd.h`), described in this document, handles specifics of SD protocol such as card initialization and data transfer commands.

Protocol layer works with the host via `sdmmc_host_t` structure. This structure contains pointers to various functions of the host.

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 capabilites, 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.

Usage with SD memory cards

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 the host peripheral and free resources allocated by the driver (e.g. `sdmmc_host_deinit()`).

Usage with SDIO cards

Initialization an probing process is the same as with SD memory cards. Only data transfer commands differ in SDIO mode.

During probing and card initialization (done by `sdmmc_card_init()`), the driver only configures the following registers of the IO card:

1. The IO portion of the card is reset by setting RES bit in "I/O Abort" (0x06) register.
2. If 4-line mode is enalbed in host and slot configuration, driver attempts to set "Bus width" field in "Bus Interface Control" (0x07) register. If that succeeds (which means that slave supports 4-line mode), host is also switched to 4-line mode.
3. If high-speed mode is enabled in host configuration, SHS bit is set in "High Speed" (0x13) register.

In particular, the driver does not set any of the bits in I/O Enable, Int Enable registers, IO block sizes, etc. Applications can set these by calling `sdmmc_io_write_byte()`.

For card configuration and data transfer, use one of the following functions:

- `sdmmc_io_read_byte()`, `sdmmc_io_write_byte()` — read and write single byte using IO_RW_DIRECT (CMD52).
- `sdmmc_io_read_bytes()`, `sdmmc_io_write_bytes()` — read and write multiple bytes using IO_RW_EXTENDED (CMD53), in byte mode.
- `sdmmc_io_read_blocks()`, `sdmmc_io_write_blocks()` — read and write blocks of data using IO_RW_EXTENDED (CMD53), in block mode.

SDIO interrupts can be enabled by the application using `sdmmc_io_enable_int()` function. When using SDIO in 1-line mode, D1 line also needs to be connected to use SDIO interrupts.

The application can wait for SDIO interrupt to occur using `sdmmc_io_wait_int()`.

Combo (memory + IO) cards

The driver does not support SD combo cards. Combo cards will be treated as IO cards.

Thread safety

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 `sdmmc_card_t` structure, or when accessing SDMMC or SD SPI host drivers. Such locking is usually implemented in the higher layer (e.g. in the filesystem driver).

API Reference

Header File

- sdmmc/include/sdmmc_cmd.h

Functions

esp_err_t **sdmmc_card_init**(**const** *sdmmc_host_t *host*, *sdmmc_card_t *out_card*)

Probe and initialize SD/MMC card using given host

Note Only SD cards (SDSC and SDHC/SDXC) are supported now. Support for MMC/eMMC cards will be added later.

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- `host`: pointer to structure defining host controller
- `out_card`: pointer to structure which will receive information about the card when the function completes

void **sdmmc_card_print_info** (FILE **stream*, **const** *sdmmc_card_t* **card*)

Print information about the card to a stream.

Parameters

- `stream`: stream obtained using fopen or fdopen
- `card`: card information structure initialized using sdmmc_card_init

esp_err_t **sdmmc_write_sectors** (*sdmmc_card_t* **card*, **const** void **src*, size_t *start_sector*, size_t *sector_count*)

Write given number of sectors to SD/MMC card

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `src`: pointer to data buffer to read data from; data size must be equal to sector_count * card->csd.sector_size
- `start_sector`: sector where to start writing
- `sector_count`: number of sectors to write

esp_err_t **sdmmc_read_sectors** (*sdmmc_card_t* **card*, void **dst*, size_t *start_sector*, size_t *sector_count*)

Write given number of sectors to SD/MMC card

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `dst`: pointer to data buffer to write into; buffer size must be at least sector_count * card->csd.sector_size
- `start_sector`: sector where to start reading
- `sector_count`: number of sectors to read

esp_err_t **sdmmc_io_read_byte** (*sdmmc_card_t* **card*, uint32_t *function*, uint32_t *reg*, uint8_t **out_byte*)

Read one byte from an SDIO card using IO_RW_DIRECT (CMD52)

Return

- ESP_OK on success

- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `function`: IO function number
- `reg`: byte address within IO function
- `out_byte`: output, receives the value read from the card

esp_err_t **sdmmc_io_write_byte**(*sdmmc_card_t* **card*, uint32_t *function*, uint32_t *reg*, uint8_t *in_byte*, uint8_t **out_byte*)

Write one byte to an SDIO card using IO_RW_DIRECT (CMD52)

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `function`: IO function number
- `reg`: byte address within IO function
- `in_byte`: value to be written
- `out_byte`: if not NULL, receives new byte value read from the card (read-after-write).

esp_err_t **sdmmc_io_read_bytes**(*sdmmc_card_t* **card*, uint32_t *function*, uint32_t *addr*, void **dst*, size_t *size*)

Read multiple bytes from an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs read operation using CMD53 in byte mode. For block mode, see sdmmc_io_read_blocks.

Return

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size exceeds 512 bytes
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `function`: IO function number
- `addr`: byte address within IO function where reading starts
- `dst`: buffer which receives the data read from card
- `size`: number of bytes to read

esp_err_t **sdmmc_io_write_bytes**(*sdmmc_card_t* **card*, uint32_t *function*, uint32_t *addr*, **const** void **src*, size_t *size*)

Write multiple bytes to an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs write operation using CMD53 in byte mode. For block mode, see sdmmc_io_write_blocks.

Return

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size exceeds 512 bytes
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `function`: IO function number
- `addr`: byte address within IO function where writing starts
- `src`: data to be written
- `size`: number of bytes to write

esp_err_t **sdmmc_io_read_blocks**(*sdmmc_card_t *card*, uint32_t *function*, uint32_t *addr*, void **dst*, size_t *size*)

Read blocks of data from an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs read operation using CMD53 in block mode. For byte mode, see sdmmc_io_read_bytes.

Return

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size is not divisible by 512 bytes
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `function`: IO function number
- `addr`: byte address within IO function where writing starts
- `dst`: buffer which receives the data read from card
- `size`: number of bytes to read, must be divisible by the card block size.

esp_err_t **sdmmc_io_write_blocks**(*sdmmc_card_t *card*, uint32_t *function*, uint32_t *addr*, **const** void **src*, size_t *size*)

Write blocks of data to an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs write operation using CMD53 in block mode. For byte mode, see sdmmc_io_write_bytes.

Return

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size is not divisible by 512 bytes
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `function`: IO function number
- `addr`: byte address within IO function where writing starts

- `src`: data to be written
- `size`: number of bytes to read, must be divisible by the card block size.

esp_err_t **sdmmc_io_enable_int** (*sdmmc_card_t* **card*)

Enable SDIO interrupt in the SDMMC host

Return

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if the host controller does not support IO interrupts

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init

esp_err_t **sdmmc_io_wait_int** (*sdmmc_card_t* **card*, TickType_t *timeout_ticks*)

Block until an SDIO interrupt is received

Slave uses D1 line to signal interrupt condition to the host. This function can be used to wait for the interrupt.

Return

- ESP_OK if the interrupt is received
- ESP_ERR_NOT_SUPPORTED if the host controller does not support IO interrupts
- ESP_ERR_TIMEOUT if the interrupt does not happen in timeout_ticks

Parameters

- `card`: pointer to card information structure previously initialized using sdmmc_card_init
- `timeout_ticks`: time to wait for the interrupt, in RTOS ticks

Header File

- driver/include/driver/sdmmc_types.h

Structures

struct sdmmc_csd_t

Decoded values from SD card Card Specific Data register

Public Members

int **csd_ver**

CSD structure format

int **mmc_ver**

MMC version (for CID format)

int **capacity**

total number of sectors

int **sector_size**

sector size in bytes

int **read_block_len**
block length for reads

int **card_command_class**
Card Command Class for SD

int **tr_speed**
Max transfer speed

struct sdmmc_cid_t
Decoded values from SD card Card IDentification register

Public Members

int **mfg_id**
manufacturer identification number

int **oem_id**
OEM/product identification number

char **name**[8]
product name (MMC v1 has the longest)

int **revision**
product revision

int **serial**
product serial number

int **date**
manufacturing date

struct sdmmc_scr_t
Decoded values from SD Configuration Register

Public Members

int **sd_spec**
SD Physical layer specification version, reported by card

int **bus_width**
bus widths supported by card: BIT(0) — 1-bit bus, BIT(2) — 4-bit bus

struct sdmmc_switch_func_rsp_t
SD SWITCH_FUNC response buffer

Public Members

uint32_t **data**[512 / 8 / sizeof(uint32_t)]
response data

struct sdmmc_command_t
SD/MMC command information

Public Members

uint32_t **opcode**
SD or MMC command index

uint32_t **arg**
SD/MMC command argument

sdmmc_response_t **response**
response buffer

void ***data**
buffer to send or read into

size_t **datalen**
length of data buffer

size_t **blklen**
block length

int **flags**
see below

esp_err_t **error**
error returned from transfer

int **timeout_ms**
response timeout, in milliseconds

struct sdmmc_host_t
SD/MMC Host description

This structure defines properties of SD/MMC host and functions of SD/MMC host which can be used by upper layers.

Public Members

uint32_t **flags**
flags defining host properties

int **slot**
slot number, to be passed to host functions

int **max_freq_khz**
max frequency supported by the host

float **io_voltage**
I/O voltage used by the controller (voltage switching is not supported)

esp_err_t (***init**) (void)
Host function to initialize the driver

esp_err_t (***set_bus_width**) (int slot, size_t width)
host function to set bus width

size_t (***get_bus_width**) (int slot)
host function to get bus width

esp_err_t (***set_card_clk**) (int slot, uint32_t freq_khz)
host function to set card clock frequency

esp_err_t (*`do_transaction`) (int slot, *sdmmc_command_t* *cmdinfo)
host function to do a transaction

esp_err_t (*`deinit`) (void)
host function to deinitialize the driver

esp_err_t (*`io_int_enable`) (int slot)
Host function to enable SDIO interrupt line

esp_err_t (*`io_int_wait`) (int slot, TickType_t timeout_ticks)
Host function to wait for SDIO interrupt line to be active

int **`command_timeout_ms`**
timeout, in milliseconds, of a single command. Set to 0 to use the default value.

`struct sdmmc_card_t`
SD/MMC card information structure

Public Members

sdmmc_host_t **`host`**
Host with which the card is associated

uint32_t **`ocr`**
OCR (Operation Conditions Register) value

sdmmc_cid_t **`cid`**
decoded CID (Card IDentification) register value

sdmmc_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)

uint32_t **`is_mem`**
Bit indicates if the card is a memory card

uint32_t **`is_sdio`**
Bit indicates if the card is an IO card

uint32_t **`num_io_functions`**
If is_sdio is 1, contains the number of IO functions on the card

uint32_t **`reserved`**
Reserved for future expansion

Macros

`SDMMC_HOST_FLAG_1BIT`
host supports 1-line SD and MMC protocol

`SDMMC_HOST_FLAG_4BIT`
host supports 4-line SD and MMC protocol

`SDMMC_HOST_FLAG_8BIT`
host supports 8-line MMC protocol

SDMMC_HOST_FLAG_SPI
host supports SPI protocol

SDMMC_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

Type Definitions

typedef uint32_t **sdmmc_response_t**[4]
SD/MMC command response buffer

3.7.3 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. If multi-page blob support is enabled, these limits change to the following: 1) 4000 bytes for strings 2) For blobs, 508000 bytes or (97.6% of the partition size - 4000) bytes whichever is lower.

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 is not directly compatible with the ESP32 flash encryption system. However, data can still be stored in encrypted form if NVS encryption is used together with ESP32 flash encryption. Please refer to *NVS Encryption* for more details.

If NVS encryption is not used, it is possible for anyone with physical access to the flash chip to alter, erase, or add key-value pairs. With NVS encryption enabled, it is not possible to alter or add a key-value pair and get recognized as a valid pair without knowing corresponding NVS encryption keys. However, there is no tamper-resistance against erase operation.

The library does try to recover from conditions when flash memory is in an inconsistent state. In particular, one should be able to power off the device at any point and time and then power it back on. This should not result in loss of data, 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.

Corrupted Page header contains invalid data, and further parsing of page data was canceled. Any items previously written into this page will not be accessible. Corresponding flash sector will not be erased immediately, and will be kept along with sectors in *uninitialized* state for later use. This may be useful for debugging.

Mapping from flash sectors to logical pages doesn't have any particular order. Library will inspect sequence numbers of pages found in each flash sector and organize pages in a list based on these numbers.

```
+--------+     +--------+     +--------+     +--------+
| Page 1 |     | Page 2 |     | Page 3 |     | Page 4 |
| Full   +---> | Full   +---> | Active |     | Empty  |   <- states
| #11    |     | #12    |     | #14    |     |        |   <- sequence numbers
+---+----+     +----+---+     +----+---+     +---+----+
    |               |              |             |
    |               |              |             |
    |               |              |             |
+---v------+  +-----v----+  +------v---+  +------v---+
| Sector 3 |  | Sector 0 |  | Sector 2 |  | Sector 1 |    <- physical sectors
+----------+  +----------+  +----------+  +----------+
```

Structure of a page

For now we assume that flash sector size is 4096 bytes and that ESP32 flash encryption hardware operates on 32-byte blocks. It is possible to introduce some settings configurable at compile-time (e.g. via menuconfig) to accommodate flash chips with different sector sizes (although it is not clear if other components in the system, e.g. SPI flash driver and SPI flash cache can support these other sizes).

Page consists of three parts: header, entry state bitmap, and entries themselves. To be compatible with ESP32 flash encryption, entry size is 32 bytes. For integer types, entry holds one key-value pair. For strings and blobs, an entry holds part of key-value pair (more on that in the entry structure description).

The following diagram illustrates page structure. Numbers in parentheses indicate size of each part in bytes.

```
+-----------+--------------+-------------+-------------------------+
| State (4) | Seq. no. (4) | version (1) | Unused (19) | CRC32 (4) |   Header (32)
+-----------+--------------+-------------+-------------------------+
|                Entry state bitmap (32)                            |
+-------------------------------------------------------------------+
|                       Entry 0 (32)                                |
+-------------------------------------------------------------------+
|                       Entry 1 (32)                                |
+-------------------------------------------------------------------+
/                                                                   /
```

(continues on next page)

(continued from previous page)

```
/                                                                 /
+-----------------------------------------------------------------+
|                        Entry 125 (32)                           |
+-----------------------------------------------------------------+
```

Page header and entry state bitmap are always written to flash unencrypted. Entries are encrypted if flash encryption feature of the ESP32 is used.

Page state values are defined in such a way that changing state is possible by writing 0 into some of the bits. Therefore it not necessary to erase the page to change page state, unless that is a change to *erased* state.

The version field in the header reflects NVS format version used. For backward compatibility reasons, it is decremented for every version upgrade starting at 0xff (i.e. 0xff for version-1, 0xfe for version-2 and so on).

CRC32 value in header is calculated over the part which doesn't include state value (bytes 4 to 28). Unused part is currently filled with `0xff` bytes.

The following sections describe structure of entry state bitmap and entry itself.

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. If multi-page blob support is enabled, blobs are allowed to span over multiple pages by dividing them into smaller chunks. For the purpose tracking these chunks, an additional fixed length metadata entry is stored called "blob index" entry. Earlier format of blobs are still supported (can be read and modified). However, once the blobs are modified, they are stored using the new format.

```
+--------+----------+----------+----------------+-----------+---------------+---------
↪-+
| NS (1) | Type (1) | Span (1) | ChunkIndex (1) | CRC32 (4) |    Key (16)   | Data␣
↪(8) |
+--------+----------+----------+----------------+-----------+---------------+---------
↪-+

                                         Primitive  +--------------------------------+
                                        +-------->  |     Data (8)                   |
                                        | Types     +--------------------------------+
                   +-> Fixed length --
                   |                    |           +---------+--------------+--------
↪-------+-------+
                   |                    +-------->  | Size(4) | ChunkCount(1)|␣
↪ChunkStart(1) | Rsv(2)|
```

(continues on next page)

(continued from previous page)

```
    Data format ---+                    Blob Index  +---------+--------------+--------
↪-------+-------+
                   |
                   |                              +----------+---------+-----------+
                   +->   Variable length   -->    | Size (2) | Rsv (2) | CRC32 (4) |
                        (Strings, Blob Data)      +----------+---------+-----------+
```

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.

ChunkIndex Used to store index of the blob-data chunk for blob types. For other types, this should be `0xff`.

CRC32 Checksum calculated over all the bytes in this entry, except for the CRC32 field itself.

Key Zero-terminated ASCII string containing 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 "blob index" entry, these 8 bytes hold the following information about data-chunks:

- **Size** (Only for blob index.) Size, in bytes, of complete blob data.
- **ChunkCount** (Only for blob index.) Total number of blob-data chunks into which the blob was divided during storage.
- **ChunkStart** (Only for blob index.) ChunkIndex of the first blob-data chunk of this blob. Subsequent chunks have chunkIndex incrementely allocated (step of 1).

For string and blob data chunks, 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, key name and ChunkIndex. 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 useage per page is therefore 128 bytes, maximum is 640 bytes.

NVS Encryption

Data stored in NVS partitions can be encrypted using AES-XTS in the manner similar to one mentioned in disc encryption standard IEEE P1619. For the purpose of encryption, each entry is considered as one *sector* and relative address of the entry (w.r.t. partition-start) is fed to the encryption algorithm as *sector-number*. The keys required for nvs encryption are stored in yet another partition, which is protected using *Flash Encryption*. Therefore, enabling *Flash Encryption* is a prerequisite for NVS encryption.

NVS key partition

An application requiring NVS encryption support needs to be compiled with a key-partition of type *data* and subtype *key*. This partition should be marked as *encrypted*. Refer to *Partition Tables* for more details. The size of the partition should be 4096 bytes (minimum partition size). The structure of this partition is depicted below.

```
+-----------+--------------+-------------+----+
|              XTS encryption key(32)         |
+---------------------------------------------+
|              XTS tweak key (32)             |
+---------------------------------------------+
|                  CRC32(4)                   |
+---------------------------------------------+
```

This partition can be generated using *nvs partition generator* utility and flashed onto the device. Since the partition is marked *encrypted* and *Flash Encryption* is enabled, bootloader will encrypt this partition using flash encryption key on first boot. Alternatively, the keys can be generated after startup using `nvs_flash_generate_keys` API provided by `nvs_flash.h`, which will then write those keys onto the key-partition in encrypted form.

It is possible for an application to use different keys for different NVS partitions and thereby have multiple key-partitions. However, it is a responsibilty of the application to provide correct key-partition/keys for the purpose of encryption/decryption.

Encrypted Read/Write

The same NVS APIs `nvs_read_*` or `nvs_write_*` can be used for reading and writing of encrypted nvs partition as well. However, the APIs for initialising NVS partitions are different. `nvs_flash_secure_init` and `nvs_flash_secure_init_partition` are used for initialising instead of `nvs_flash_init` and `nvs_flash_init_partition` respectively. `nvs_sec_cfg_t` structure required for these APIs can be populated using `nvs_flash_generate_keys` or `nvs_flash_read_security_cfg`.

Applications are expected to follow the following steps in order to perform NVS read/write operations with encryption enabled.

1. Find key partition and NVS data partition using `esp_partition_find*` APIs.
2. Populate `nvs_sec_cfg_t` struct using `nvs_flash_read_security_cfg` or `nvs_flash_generate_keys` APIs.
3. Initialise NVS flash partition using `nvs_flash_secure_init` or `nvs_flash_secure_init_partition` APIs.
4. Open a namespace using `nvs_open` or `nvs_open_from_part` APIs
5. Perform NVS read/write operations using `nvs_read_*` or `nvs_write_*`
6. Deinitialise NVS partition using `nvs_flash_deinit`.

NVS Partition Generator Utility

This utility helps in generating NVS-esque partition binary file which can be flashed separately on a dedicated partition via a flashing utility. Key-value pairs to be flashed onto the partition can be provided via a CSV file. Refer to *NVS Partition Generator Utility* for more details.

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

esp_err_t **nvs_flash_secure_init**(*nvs_sec_cfg_t* **cfg*)

Initialize the default NVS partition.

This API initialises the default NVS partition. The default NVS partition is the one that is labeled "nvs" in the partition table.

Return

- ESP_OK if storage was successfully initialized.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if no partition with label "nvs" is found in the partition table
- one of the error codes from the underlying flash storage driver

Parameters

- `cfg`: Security configuration (keys) to be used for NVS encryption/decryption. If cfg is NULL, no encryption is used.

esp_err_t **nvs_flash_secure_init_partition**(**const** char **partition_label*, *nvs_sec_cfg_t* **cfg*)

Initialize NVS flash storage for the specified partition.

Return

- ESP_OK if storage was successfully initialized.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if specified partition is not found in the partition table
- one of the error codes from the underlying flash storage driver

Parameters

- `partition_label`: Label of the partition. Note that internally a reference to passed value is kept and it should be accessible for future operations
- `cfg`: Security configuration (keys) to be used for NVS encryption/decryption. If cfg is null, no encryption/decryption is used.

esp_err_t **nvs_flash_generate_keys**(**const** *esp_partition_t *partition*, *nvs_sec_cfg_t *cfg*)

Generate and store NVS keys in the provided esp partition.

Return -ESP_OK, if cfg was read successfully; -or error codes from esp_partition_write/erase APIs.

Parameters

- `partition`: Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- `cfg`: Pointer to nvs security configuration structure. Pointer must be non-NULL. Generated keys will be populated in this structure.

esp_err_t **nvs_flash_read_security_cfg**(**const** *esp_partition_t *partition*, *nvs_sec_cfg_t *cfg*)

Read NVS security configuration from a partition.

Note Provided parition is assumed to be marked 'encrypted'.

Return -ESP_OK, if cfg was read successfully; -ESP_ERR_NVS_KEYS_NOT_INITIALIZED, if the partition is not yet written with keys. -ESP_ERR_NVS_CORRUPT_KEY_PART, if the partition containing keys is found to be corrupt -or error codes from esp_partition_read API.

Parameters

- `partition`: Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- `cfg`: Pointer to nvs security configuration structure. Pointer must be non-NULL.

Structures

struct nvs_sec_cfg_t

Key for encryption and decryption.

Public Members

uint8_t **eky**[**NVS_KEY_SIZE**]

XTS encryption and decryption key

uint8_t **tky**[**NVS_KEY_SIZE**]

XTS tweak key

Macros

NVS_KEY_SIZE

Header File

- nvs_flash/include/nvs.h

Functions

esp_err_t **nvs_set_i8** (*nvs_handle 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 (4000 bytes if multi-page blob support is enabled).

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␣
↪dynamic array:
size_t required_size;
```

(continues on next page)

(continued from previous page)

```
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 (508000 bytes or (97.6% of the partition size - 4000) bytes whichever is lower, in case multi-page blob support is enabled).

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

esp_err_t **nvs_get_stats**(**const** char **part_name*, *nvs_stats_t* **nvs_stats*)

Fill structure *nvs_stats_t*. It provides info about used memory the partition.

This function calculates to runtime the number of used entries, free entries, total entries, and amount namespace in partition.

```
// Example of nvs_get_stats() to get the number of used entries and free entries:
nvs_stats_t nvs_stats;
nvs_get_stats(NULL, &nvs_stats);
printf("Count: UsedEntries = (%d), FreeEntries = (%d), AllEntries = (%d)\n",
       nvs_stats.used_entries, nvs_stats.free_entries, nvs_stats.total_entries);
```

Return

- ESP_OK if the changes have been written successfully. Return param nvs_stats will be filled.
- ESP_ERR_NVS_PART_NOT_FOUND if the partition with label "name" is not found. Return param nvs_stats will be filled 0.
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized. Return param nvs_stats will be filled 0.
- ESP_ERR_INVALID_ARG if nvs_stats equal to NULL.
- ESP_ERR_INVALID_STATE if there is page with the status of INVALID. Return param nvs_stats will be filled not with correct values because not all pages will be counted. Counting will be interrupted at the first INVALID page.

Parameters

- `part_name`: Partition name NVS in the partition table. If pass a NULL than will use NVS_DEFAULT_PART_NAME ("nvs").
- `nvs_stats`: Returns filled structure nvs_states_t. It provides info about used memory the partition.

esp_err_t **nvs_get_used_entry_count**(*nvs_handle handle*, size_t **used_entries*)

Calculate all entries in a namespace.

Note that to find out the total number of records occupied by the namespace, add one to the returned value used_entries (if err is equal to ESP_OK). Because the name space entry takes one entry.

```
// Example of nvs_get_used_entry_count() to get amount of all key-value pairs in␣
↪one namespace:
nvs_handle handle;
nvs_open("namespace1", NVS_READWRITE, &handle);
...
size_t used_entries;
size_t total_entries_namespace;
if(nvs_get_used_entry_count(handle, &used_entries) == ESP_OK){
    // the total number of records occupied by the namespace
    total_entries_namespace = used_entries + 1;
}
```

Return

- ESP_OK if the changes have been written successfully. Return param used_entries will be filled valid value.
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized. Return param used_entries will be filled 0.
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL. Return param used_entries will be filled 0.
- ESP_ERR_INVALID_ARG if nvs_stats equal to NULL.
- Other error codes from the underlying storage driver. Return param used_entries will be filled 0.

Parameters

- `handle`: Handle obtained from nvs_open function.
- `used_entries`: Returns amount of used entries from a namespace.

Structures

struct nvs_stats_t

Note Info about storage space NVS.

Public Members

size_t **used_entries**
Amount of used entries.

size_t **free_entries**
Amount of free entries.

size_t **total_entries**
Amount all available entries.

size_t **namespace_count**
Amount name space.

Macros

ESP_ERR_NVS_BASE
Starting number of error codes

ESP_ERR_NVS_NOT_INITIALIZED
The storage driver is not initialized

ESP_ERR_NVS_NOT_FOUND
Id namespace doesn't exist yet and mode is NVS_READONLY

ESP_ERR_NVS_TYPE_MISMATCH
The type of set or get operation doesn't match the type of value stored in NVS

ESP_ERR_NVS_READ_ONLY
Storage handle was opened as read only

ESP_ERR_NVS_NOT_ENOUGH_SPACE
There is not enough space in the underlying storage to save the value

ESP_ERR_NVS_INVALID_NAME
Namespace name doesn't satisfy constraints

ESP_ERR_NVS_INVALID_HANDLE
Handle has been closed or is NULL

ESP_ERR_NVS_REMOVE_FAILED
The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

ESP_ERR_NVS_KEY_TOO_LONG
Key name is too long

ESP_ERR_NVS_PAGE_FULL
Internal error; never returned by nvs API functions

ESP_ERR_NVS_INVALID_STATE
NVS is in an inconsistent state due to a previous error. Call nvs_flash_init and nvs_open again, then retry.

ESP_ERR_NVS_INVALID_LENGTH
String or blob length is not sufficient to store data

ESP_ERR_NVS_NO_FREE_PAGES
NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call nvs_flash_init again.

ESP_ERR_NVS_VALUE_TOO_LONG
String or blob length is longer than supported by the implementation

ESP_ERR_NVS_PART_NOT_FOUND
Partition with specified name is not found in the partition table

ESP_ERR_NVS_NEW_VERSION_FOUND
NVS partition contains data in new format and cannot be recognized by this version of code

ESP_ERR_NVS_XTS_ENCR_FAILED
XTS encryption failed while writing NVS entry

ESP_ERR_NVS_XTS_DECR_FAILED
XTS decryption failed while reading NVS entry

ESP_ERR_NVS_XTS_CFG_FAILED
XTS configuration setting failed

ESP_ERR_NVS_XTS_CFG_NOT_FOUND
XTS configuration not found

ESP_ERR_NVS_ENCR_NOT_SUPPORTED
NVS encryption is not supported in this version

ESP_ERR_NVS_KEYS_NOT_INITIALIZED
NVS key partition is uninitialized

ESP_ERR_NVS_CORRUPT_KEY_PART
NVS key partition is corrupt

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

3.7.4 NVS Partition Generator Utility

Introduction

nvs_flash/nvs_partition_generator/nvs_partition_gen.py utility is designed to help create a binary file, compatible with NVS architecture defined in *Non-Volatile Storage*, based on user provided key-value pairs in a CSV file. Utility is ideally suited for generating a binary blob, containing data specific to ODM/OEM, which can be flashed externally at the time of device manufacturing. This helps manufacturers set unique value for various parameters for each device, e.g. serial number, while using same application firmaware for all devices.

Prerequisites

To use this utility in encryption mode, the following packages need to be installed:

- cryptography package

This dependency is already captured by including these packages in *requirement.txt* in top level IDF directory.

CSV file format

Each row of the .csv file should have 4 parameters, separated by comma. Below is the description of each of these parameters:

Key Key of the data. Data can later be accessed from an application via this key.

Type Supported values are `file`, `data` and `namespace`.

Encoding Supported values are: `u8`, `i8`, `u16`, `u32`, `i32`, `string`, `hex2bin`, `base64` and `binary`. This specifies how actual data values are encoded in the resultant binary file. Difference between `string` and `binary` encoding is that `string` data is terminated with a NULL character, whereas `binary` data is not.

> **Note:** For `file` type, only `hex2bin`, `base64`, `string` and `binary` is supported as of now.

Value Data value.

Note: Encoding and Value cells for `namespace` field type should be empty. Encoding and Value of `namespace` is fixed and isn't configurable. Any value in these cells are ignored.

Note: First row of the CSV file should always be column header and isn't configurable.

Below is an example dump of such CSV file:

```
key,type,encoding,value     <-- column header
namespace_name,namespace,,  <-- First entry should be of type "namespace"
key1,data,u8,1
key2,file,string,/path/to/file
```

Note: Make sure there are no spaces before and after ',' in CSV file.

NVS Entry and Namespace association

When a new namespace entry is encountered in the CSV file, each follow-up entries will be part of that namespace, until next namespace entry is found, in which case all the follow-up entries will be part of the new namespace.

Note: First entry in a CSV file should always be `namespace` entry.

Multipage Blob Support

By default, binary blobs are not allowed to span over multiple pages as written in the format mentioned in section *Structure of entry*. If this format is intended to be used, the utility provides an option to enable this feature.

Encryption Support

This utility allows you to create an enrypted binary file also. Encryption used is AES-XTS encryption. Refer to *NVS Encryption* for more details.

Running the utility

Usage:

```
python nvs_partition_gen.py [--version {v1,v2}] input output
```

You can run this utility in two modes:

- Normal mode - Binary generated in this mode is an unencrypted binary file.
- Encryption mode - Binary generated in this mode is an encrypted binary file.

In normal mode:

A sample CSV file is provided with the utility. You can run the utility using below command:

```
python nvs_partition_generator.py sample.csv sample.bin
```

In encryption mode:

You can run the utility using below commands:

- By taking encryption keys as an input file. A sample encryption keys file is provided with the utility:

```
python nvs_partition_gen.py sample.csv sample_encrypted.bin --encrypt␣
↪True --keyfile testdata/keys.txt
```

- By enabling generation of encryption keys:

```
python nvs_partition_gen.py sample.csv sample_encrypted.bin --encrypt␣
↪True --keygen True
```

Note: In encryption mode, this utility creates a binary file named *encryption_keys.bin* containing the encryption keys used. This binary file is compatible with NVS key-partition structure. Refer to *NVS key partition* for more details.

You can also provide the format version number while running this utility:

- Multipage Blob Support Enabled (v2)
- Multipage Blob Support Disabled (v1)

Multipage Blob Support Enabled (v2):

You can run the utility in this format by setting the version parameter to v2, as shown below. A sample CSV file is provided with the utility:

```
python nvs_partition_gen.py sample_multipage_blob.csv partition_multipage_blob.bin --
↪version v2
```

Multipage Blob Support Disabled (v1):

You can run the utility in this format by setting the version parameter to v1, as shown below. A sample CSV file is provided with the utility:

```
python nvs_partition_gen.py sample_singlepage_blob.csv partition_single_page.bin --
↪version v1
```

Arguments	Description
input	Path to CSV file to parse. Will use stdin if omitted
output	Path to output converted binary file. Will use stdout if omitted
–version {v1,v2} | Set version. Default: v2	
–keygen {True,False} | Generate keys for encryption. Default: False (Applicable only if encryption mode is true)	
–encrypt {True,False}	Set encryption mode. Default: False
–keyfile KEYFILE	File having key for encryption (Applicable only if encryption mode is true)

Caveats

- Utility doesn't check for duplicate keys and will write data pertaining to both keys. User needs to make sure keys are distinct.
- Once a new page is created, no data will be written in the space left in previous page. Fields in the CSV file need to be ordered in such a way so as to optimize memory.
- 64-bit datatype is not yet supported.

3.7.5 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 in 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,
// ... other members initialized
```

(continues on next page)

(continued from previous page)

```
// 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));
```

Synchronous input/output multiplexing

If you want to use synchronous input/output multiplexing by `select()` then you need to register the VFS with `start_select()` and `end_select()` functions similarly to the following example:

```
// In definition of esp_vfs_t:
    .start_select = &uart_start_select,
    .end_select = &uart_end_select,
// ... other members initialized
```

`start_select()` is called for setting up the environment for detection of read/write/error conditions on file descriptors belonging to the given VFS driver.

`end_select()` is called to stop/deinitialize/free the environment which was setup by `start_select()`.

Note: `end_select()` might be called without a previous `start_select()` call in some rare circumstances. `end_select()` should fail gracefully if this is the case.

Please refer to the reference implementation for the UART peripheral in vfs/vfs_uart.c and most particularly to functions *`esp_vfs_dev_uart_register()`*, `uart_start_select()` and `uart_end_select()`.

Examples demonstrating the use of `select()` with VFS file descriptors are the peripherals/uart_select and the system/select examples.

If `select()` is used for socket file descriptors only then one can enable the *`CONFIG_USE_ONLY_LWIP_SELECT`* option which can reduce the code size and improve performance.

Note: Don't change the socket driver during an active `select()` call or you might experience some undefined behavior.

Paths

Each registered FS has a path prefix associated with it. This prefix 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

File descriptors are small positive integers from `0` to `FD_SETSIZE - 1` where `FD_SETSIZE` is defined in newlib's `sys/types.h`. The largest file descriptors (configured by `CONFIG_LWIP_MAX_SOCKETS`) are reserved for sockets. The VFS component contains a lookup-table called `s_fd_table` for mapping global file descriptors to VFS driver indexes registered in the `s_vfs` array.

Standard IO streams (stdin, stdout, stderr)

If "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 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_fd_range**(**const** *esp_vfs_t* **vfs*, void **ctx*, int *min_fd*, int *max_fd*)

Special case function for registering a VFS that uses a method other than open() to open new file descriptors from the interval <min_fd; max_fd).

This is a special-purpose function intended for registering LWIP sockets to VFS.

Return ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSes are registered, ESP_ERR_INVALID_ARG if the file descriptor boundaries are incorrect.

Parameters

- `vfs`: Pointer to *esp_vfs_t*. Meaning is the same as for esp_vfs_register().
- `ctx`: Pointer to context structure. Meaning is the same as for esp_vfs_register().
- `min_fd`: The smallest file descriptor this VFS will use.
- `max_fd`: Upper boundary for file descriptors this VFS will use (the biggest file descriptor plus one).

esp_err_t **esp_vfs_register_with_id**(**const** *esp_vfs_t* **vfs*, void **ctx*, *esp_vfs_id_t* **vfs_id*)

Special case function for registering a VFS that uses a method other than open() to open new file descriptors. In comparison with esp_vfs_register_fd_range, this function doesn't pre-registers an interval of file descriptors. File descriptors can be registered later, by using esp_vfs_register_fd.

Return ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSes are registered, ESP_ERR_INVALID_ARG if the file descriptor boundaries are incorrect.

Parameters

- `vfs`: Pointer to *esp_vfs_t*. Meaning is the same as for esp_vfs_register().

- `ctx`: Pointer to context structure. Meaning is the same as for esp_vfs_register().
- `vfs_id`: Here will be written the VFS ID which can be passed to esp_vfs_register_fd for registering file descriptors.

esp_err_t **esp_vfs_unregister**(**const** char **base_path*)

Unregister a virtual filesystem for given path prefix

Return ESP_OK if successful, ESP_ERR_INVALID_STATE if VFS for given prefix hasn't been registered

Parameters

- `base_path`: file prefix previously used in esp_vfs_register call

esp_err_t **esp_vfs_register_fd**(*esp_vfs_id_t vfs_id*, int **fd*)

Special function for registering another file descriptor for a VFS registered by esp_vfs_register_with_id.

Return ESP_OK if the registration is successful, ESP_ERR_NO_MEM if too many file descriptors are registered, ESP_ERR_INVALID_ARG if the arguments are incorrect.

Parameters

- `vfs_id`: VFS identificator returned by esp_vfs_register_with_id.
- `fd`: The registered file descriptor will be written to this address.

esp_err_t **esp_vfs_unregister_fd**(*esp_vfs_id_t vfs_id*, int *fd*)

Special function for unregistering a file descriptor belonging to a VFS registered by esp_vfs_register_with_id.

Return ESP_OK if the registration is successful, ESP_ERR_INVALID_ARG if the arguments are incorrect.

Parameters

- `vfs_id`: VFS identificator returned by esp_vfs_register_with_id.
- `fd`: File descriptor which should be unregistered.

int **esp_vfs_select**(int *nfds*, fd_set **readfds*, fd_set **writefds*, fd_set **errorfds*, **struct** timeval **timeout*)

Synchronous I/O multiplexing which implements the functionality of POSIX select() for VFS.

Return The number of descriptors set in the descriptor sets, or -1 when an error (specified by errno) have occurred.

Parameters

- `nfds`: Specifies the range of descriptors which should be checked. The first nfds descriptors will be checked in each set.
- `readfds`: If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for being ready to read, and on output indicates which descriptors are ready to read.
- `writefds`: If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for being ready to write, and on output indicates which descriptors are ready to write.
- `errorfds`: If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for error conditions, and on output indicates which descriptors have error conditions.

- `timeout`: If not NULL, then points to timeval structure which specifies the time period after which the functions should time-out and return. If it is NULL, then the function will not time-out.

void **esp_vfs_select_triggered**(*SemaphoreHandle_t* **signal_sem*)

Notification from a VFS driver about a read/write/error condition.

This function is called when the VFS driver detects a read/write/error condition as it was requested by the previous call to start_select.

Parameters

- `signal_sem`: semaphore handle which was passed to the driver by the start_select call

void **esp_vfs_select_triggered_isr**(*SemaphoreHandle_t* **signal_sem*, BaseType_t **woken*)

Notification from a VFS driver about a read/write/error condition (ISR version)

This function is called when the VFS driver detects a read/write/error condition as it was requested by the previous call to start_select.

Parameters

- `signal_sem`: semaphore handle which was passed to the driver by the start_select call
- `woken`: is set to pdTRUE if the function wakes up a task with higher priority

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

esp_err_t (***start_select**)(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, *SemaphoreHandle_t* *signal_sem)

start_select is called for setting up synchronous I/O multiplexing of the desired file descriptors in the given VFS

int (***socket_select**)(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, **struct** timeval *timeout)

socket select function for socket FDs with the functionality of POSIX select(); this should be set only for the socket VFS

void (*`stop_socket_select`)()
called by VFS to interrupt the socket_select call when select is activated from a non-socket VFS driver; set only for the socket driver

void (*`stop_socket_select_isr`)(BaseType_t *woken)
stop_socket_select which can be called from ISR; set only for the socket driver

void *(*`get_socket_select_semaphore`)()
end_select is called to stop the I/O multiplexing and deinitialize the environment created by start_select for the given VFS

void (*`end_select`)()
get_socket_select_semaphore returns semaphore allocated in the socket driver; set only for the socket driver

Macros

`MAX_FDS`
Maximum number of (global) file descriptors.

`ESP_VFS_PATH_MAX`
Maximum length of path prefix (not including zero terminator)

`ESP_VFS_FLAG_DEFAULT`
Default value of flags member in *esp_vfs_t* structure.

`ESP_VFS_FLAG_CONTEXT_PTR`
Flag which indicates that FS needs extra context pointer in syscalls.

Type Definitions

`typedef` int **`esp_vfs_id_t`**

Header File

- vfs/include/esp_vfs_dev.h

Functions

void **`esp_vfs_dev_uart_register`**()
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.

3.7.6 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

fatfs/src/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 FatFs `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 <http://www.elm-chan.org/fsw/ff/doc/mount.html>*.
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_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 FatFs `f_mount` function for the same drive number, with NULL `FATFS*` argument, to unmount the filesystem.
8. Call FatFs `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

fatfs/src/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.

size_t **allocation_unit_size**

If format_if_mount_failed is set, and mount fails, format the card with given allocation unit size. Must be a power of 2, between sector size and 128 * sector size. For SD cards, sector size is always 512 bytes. For wear_levelling, sector size is determined by CONFIG_WL_SECTOR_SIZE option.

Using larger allocation unit size will result in higher read/write performance and higher overhead when storing small files.

Setting this field to 0 will result in allocation unit set to the sector size.

esp_err_t **esp_vfs_fat_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

Using FatFs with VFS in read-only mode

Convenience functions, `esp_vfs_fat_rawflash_mount()` and `esp_vfs_fat_rawflash_unmount()`, are provided by fatfs/src/esp_vfs_fat.h header file in order to perform steps 1-3 and 7-9 for read-only FAT partitions.

These are particularly helpful for data partitions written only once during factory provisioning and need not be changed by production application throughout the lifetime.

esp_err_t **esp_vfs_fat_rawflash_mount**(**const** char **base_path*, **const** char **partition_label*, **const** *esp_vfs_fat_mount_config_t* **mount_config*)

Convenience function to initialize read-only FAT filesystem and register it in VFS.

This is an all-in-one function which does the following:

- finds the partition with defined partition_label. Partition label should be configured in the partition table.
- mounts FAT partition using FATFS library
- registers FATFS library with VFS, with prefix given by base_prefix variable

Note Wear levelling is not used when FAT is mounted in read-only mode using this function.

Return

- ESP_OK on success
- ESP_ERR_NOT_FOUND if the partition table does not contain FATFS partition with given label
- ESP_ERR_INVALID_STATE if esp_vfs_fat_rawflash_mount was already called for the same partition
- ESP_ERR_NO_MEM if memory can not be allocated
- ESP_FAIL if partition can not be mounted
- other error codes from SPI flash driver, or FATFS drivers

Parameters

- `base_path`: path where FATFS partition should be mounted (e.g. "/spiflash")
- `partition_label`: label of the partition which should be used
- `mount_config`: pointer to structure with extra parameters for mounting FATFS

esp_err_t **esp_vfs_fat_rawflash_unmount**(**const** char **base_path*, **const** char **partition_label*)

Unmount FAT filesystem and release resources acquired using esp_vfs_fat_rawflash_mount.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount hasn't been called

Parameters

- `base_path`: path where partition should be registered (e.g. "/spiflash")
- `partition_label`: label of partition to be unmounted

FatFS disk IO layer

FatFs has been extended with 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.

3.7.7 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 memoty 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 avalible 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.

size_t **allocation_unit_size**

If format_if_mount_failed is set, and mount fails, format the card with given allocation unit size. Must be a power of 2, between sector size and 128 * sector size. For SD cards, sector size is always 512 bytes. For wear_levelling, sector size is determined by CONFIG_WL_SECTOR_SIZE option.

Using larger allocation unit size will result in higher read/write performance and higher overhead when storing small files.

Setting this field to 0 will result in allocation unit set to the sector size.

esp_err_t **esp_vfs_fat_spiflash_unmount**(**const** char **base_path*, *wl_handle_t wl_handle*)

Unmount FAT filesystem and release resources acquired using esp_vfs_fat_spiflash_mount.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount hasn't been called

Parameters

- `base_path`: path where partition should be registered (e.g. "/spiflash")
- `wl_handle`: wear levelling driver handle returned by esp_vfs_fat_spiflash_mount

Mid level API Reference

Header File

- wear_levelling/include/wear_levelling.h

Functions

esp_err_t **wl_mount**(**const** *esp_partition_t **partition*, *wl_handle_t **out_handle*)

Mount WL for defined partition.

Return

- ESP_OK, if the allocation was successfully;
- ESP_ERR_INVALID_ARG, if WL allocation was unsuccessful;
- ESP_ERR_NO_MEM, if there was no memory to allocate WL components;

Parameters

- `partition`: that will be used for access
- `out_handle`: handle of the WL instance

esp_err_t **wl_unmount**(*wl_handle_t handle*)

Unmount WL for defined partition.

Return

- ESP_OK, if the operation completed successfully;
- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL partition handle

esp_err_t **wl_erase_range** (*wl_handle_t handle*, size_t *start_addr*, size_t *size*)
Erase part of the WL storage.

Return

- ESP_OK, if the range was erased successfully;
- ESP_ERR_INVALID_ARG, if iterator or dst are NULL;
- ESP_ERR_INVALID_SIZE, if erase would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL handle that are related to the partition
- `start_addr`: Address where erase operation should start. Must be aligned to the result of function wl_sector_size(...).
- `size`: Size of the range which should be erased, in bytes. Must be divisible by result of function wl_sector_size(...)..

esp_err_t **wl_write** (*wl_handle_t handle*, size_t *dest_addr*, **const** void **src*, size_t *size*)
Write data to the WL storage.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using wl_erase_range function.

Note Prior to writing to WL storage, make sure it has been erased with wl_erase_range call.

Return

- ESP_OK, if data was written successfully;
- ESP_ERR_INVALID_ARG, if dst_offset exceeds partition size;
- ESP_ERR_INVALID_SIZE, if write would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL handle that are related to the partition
- `dest_addr`: Address where the data should be written, relative to the beginning of the partition.
- `src`: Pointer to the source buffer. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- `size`: Size of data to be written, in bytes.

esp_err_t **wl_read** (*wl_handle_t handle*, size_t *src_addr*, void **dest*, size_t *size*)
Read data from the WL storage.

Return

- ESP_OK, if data was read successfully;
- ESP_ERR_INVALID_ARG, if src_offset exceeds partition size;
- ESP_ERR_INVALID_SIZE, if read would go out of bounds of the partition;

- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL module instance that was initialized before
- `dest`: Pointer to the buffer where data should be stored. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- `src_addr`: Address of the data to be read, relative to the beginning of the partition.
- `size`: Size of data to be read, in bytes.

size_t **wl_size**(*wl_handle_t handle*)
Get size of the WL storage.

Return usable size, in bytes

Parameters

- `handle`: WL module handle that was initialized before

size_t **wl_sector_size**(*wl_handle_t handle*)
Get sector size of the WL instance.

Return sector size, in bytes

Parameters

- `handle`: WL module handle that was initialized before

Macros

WL_INVALID_HANDLE

Type Definitions

typedef int32_t **wl_handle_t**
wear levelling handle

3.7.8 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:

```
mkspiffs -c [src_folder] -b 4096 -p 256 -s 0x100000 spiffs.bin
```

To flash the image to ESP32 at offset 0x110000:

```
python esptool.py --chip esp32 --port [port] --baud [baud] write_flash -z 0x110000␣
↪spiffs.bin
```

See also

- *Partition Table documentation*

Application Example

An example for using SPIFFS is provided in storage/spiffs 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

- spiffs/include/esp_spiffs.h

Header File

- spiffs/include/esp_spiffs.h

Functions

esp_err_t **esp_vfs_spiffs_register**(**const** *esp_vfs_spiffs_conf_t *conf*)

Register and mount SPIFFS to VFS with given path prefix.

Return

- ESP_OK if success
- ESP_ERR_NO_MEM if objects could not be allocated
- ESP_ERR_INVALID_STATE if already mounted or partition is encrypted

- ESP_ERR_NOT_FOUND if partition for SPIFFS was not found
- ESP_FAIL if mount or format fails

Parameters

- `conf`: Pointer to *esp_vfs_spiffs_conf_t* configuration structure

esp_err_t **esp_vfs_spiffs_unregister**(**const** char **partition_label*)
Unregister and unmount SPIFFS from VFS

Return

- ESP_OK if successful
- ESP_ERR_INVALID_STATE already unregistered

Parameters

- `partition_label`: 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.

3.8 System API

3.8.1 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* and *ESP-IDF FreeRTOS Additions*.

Task API

Header File

- freertos/include/freertos/task.h

Functions

BaseType_t **xTaskCreatePinnedToCore**(TaskFunction_t *pvTaskCode*, **const** char ***const** *pcName*, **const** uint32_t *usStackDepth*, void ***const** *pvParameters*, UBaseType_t *uxPriority*, *TaskHandle_t* ***const** *pvCreatedTask*, **const** BaseType_t *xCoreID*)

Create a new task with a specified affinity.

This function is similar to xTaskCreate, but allows setting task affinity in SMP system.

Return pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

Parameters

- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN - default is 16.
- `usStackDepth`: The size of the task stack specified as the number of 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 )
```

(continues on next page)

(continued from previous page)

```
{
static uint8_t ucParameterToPass;
TaskHandle_t xHandle = NULL;

 // Create the task, storing the handle.  Note that the passed parameter␣
↪ucParameterToPass
 // must exist for the lifetime of the task, so in this case is declared static. ␣
↪If it was just an
 // an automatic stack variable it might no longer exist, or at least have been␣
↪corrupted, by the time
 // the new task attempts to access it.
 xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY,
↪ &xHandle );
    configASSERT( xHandle );

 // Use the handle to delete the task.
    if( xHandle != NULL )
    {
     vTaskDelete( xHandle );
    }
}
```

Return pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

Note If program uses thread local variables (ones specified with "__thread" keyword) then storage for them will be allocated on the task's stack.

Parameters

- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN - default is 16.
- `usStackDepth`: The size of the task stack specified as the number of 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 (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 ];
```

(continues on next page)

(continued from previous page)

```
// Function that implements the task being created.
void vTaskCode( void * pvParameters )
{
    // The parameter value is expected to be 1 as 1 is passed in the
    // pvParameters value in the call to xTaskCreateStatic().
    configASSERT( ( uint32_t ) pvParameters == 1UL );

    for( ;; )
    {
        // Task code goes here.
    }
}

// Function that creates a task.
void vOtherFunction( void )
{
    TaskHandle_t xHandle = NULL;

    // Create the task without using any dynamic memory allocation.
    xHandle = xTaskCreateStatic(
                  vTaskCode,       // Function that implements the task.
                  "NAME",          // Text name for the task.
                  STACK_SIZE,      // Stack size in 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.

Note If program uses thread local variables (ones specified with "__thread" keyword) then storage for them will be allocated on the task's stack.

Parameters

- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. The maximum length of the string is defined by configMAX_TASK_NAME_LEN in FreeRTOSConfig.h.
- `ulStackDepth`: The size of the task stack specified as the number of 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.
```

(continues on next page)

(continued from previous page)

```
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 );

 // ...

 // 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
```

(continues on next page)

(continued from previous page)

```
 // another task calls vTaskResume( xHandle ).

 //...


 // Resume the suspended task ourselves.
 vTaskResume( xHandle );

 // The created task will once again get microcontroller processing
 // time in accordance with its priority within the system.
}
```

Parameters

- `xTaskToResume`: Handle to the task being readied.

BaseType_t **`xTaskResumeFromISR`** (*TaskHandle_t xTaskToResume*)

An implementation of vTaskResume() that can be called from within an ISR.

INCLUDE_xTaskResumeFromISR must be defined as 1 for this function to be available. See the configuration section for more information.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to xTaskResumeFromISR ().

xTaskResumeFromISR() should not be used to synchronise a task with an interrupt if there is a chance that the interrupt could arrive prior to the task being suspended - as this can lead to interrupts being missed. Use of a semaphore as a synchronisation mechanism would avoid this eventuality.

Return pdTRUE if resuming the task should result in a context switch, otherwise pdFALSE. This is used by the ISR to determine if a context switch may be required following the ISR.

Parameters

- `xTaskToResume`: Handle to the task being readied.

void **`vTaskSuspendAll`** (void)

Suspends the scheduler without disabling interrupts.

Context switches will not occur while the scheduler is suspended.

After calling vTaskSuspendAll () the calling task will continue to execute without risk of being swapped out until a call to xTaskResumeAll () has been made.

API functions that have the potential to cause a context switch (for example, vTaskDelayUntil(), xQueueSend(), etc.) must not be called while the scheduler is suspended.

Example usage:

```
void vTask1( void * pvParameters )
{
 for( ;; )
 {
     // Task code goes here.

     // ...

     // At some point the task wants to perform a long operation during
```

(continues on next page)

(continued from previous page)

```
    // which it does not want to get swapped out.  It cannot use
    // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
    // operation may cause interrupts to be missed - including the
    // ticks.

    // Prevent the real time kernel swapping out the task.
    vTaskSuspendAll ();

    // Perform the operation here.  There is no need to use critical
    // sections as we have all the microcontroller processing time.
    // During this time interrupts will still operate and the kernel
    // tick count will be maintained.

    // ...

    // The operation is complete.  Restart the kernel.
    xTaskResumeAll ();
 }
}
```

BaseType_t **xTaskResumeAll** (void)

Resumes scheduler activity after it was suspended by a call to vTaskSuspendAll().

xTaskResumeAll() only resumes the scheduler. It does not unsuspend tasks that were previously suspended by a call to vTaskSuspend().

Example usage:

```
void vTask1( void * pvParameters )
{
 for( ;; )
 {
     // Task code goes here.

     // ...

     // At some point the task wants to perform a long operation during
     // which it does not want to get swapped out.  It cannot use
     // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
     // operation may cause interrupts to be missed - including the
     // ticks.

     // Prevent the real time kernel swapping out the task.
     vTaskSuspendAll ();

     // Perform the operation here.  There is no need to use critical
     // sections as we have all the microcontroller processing time.
     // During this time interrupts will still operate and the real
     // time kernel tick count will be maintained.

     // ...

     // 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 () )
     {
```

(continues on next page)

(continued from previous page)

```
            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 )
             {
```

(continues on next page)

(continued from previous page)

```
                sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n",␣
↪pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter,␣
↪ulStatsAsPercentage );
            }
            else
            {
                // If the percentage is zero here then the task has
                // consumed less than 1% of the total run time.
                sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n",␣
↪pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter );
            }

            pcWriteBuffer += strlen( ( char * ) pcWriteBuffer );
        }
    }

    // The array is no longer needed, free the memory it consumes.
    vPortFree( pxTaskStatusArray );
 }
}
```

Note This function is intended for debugging use only as its use results in the scheduler remaining suspended for an extended period.

Return The number of TaskStatus_t structures that were populated by uxTaskGetSystemState(). This should equal the number returned by the uxTaskGetNumberOfTasks() API function, but will be zero if the value passed in the uxArraySize parameter was too small.

Parameters

- `pxTaskStatusArray`: A pointer to an array of TaskStatus_t structures. The array must contain at least one TaskStatus_t structure for each task that is under the control of the RTOS. The number of tasks under the control of the RTOS can be determined using the uxTaskGetNumberOfTasks() API function.
- `uxArraySize`: The size of the array pointed to by the pxTaskStatusArray parameter. The size is specified as the number of indexes in the array, or the number of TaskStatus_t structures contained in the array, not by the number of bytes in the array.
- `pulTotalRunTime`: If configGENERATE_RUN_TIME_STATS is set to 1 in FreeRTOSConfig.h then *pulTotalRunTime is set by uxTaskGetSystemState() to the total run time (as defined by the run time stats clock, see http://www.freertos.org/rtos-run-time-stats.html) since the target booted. pulTotalRunTime can be set to NULL to omit the total run time information.

void **vTaskList** (char **pcWriteBuffer*)

List all the current tasks.

configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. See the configuration section of the FreeRTOS.org website for more information.

Lists all the current tasks, along with their current state and stack usage high water mark.

Note This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Tasks are reported as blocked ('B'), ready ('R'), deleted ('D') or suspended ('S').

vTaskList() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays task names, states and stack usage.

Note This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskList() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskList().

Parameters

- `pcWriteBuffer`: A buffer into which the above mentioned details will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

void **`vTaskGetRunTimeStats`** (char **pcWriteBuffer*)

Get the state of running tasks as a string

configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. The application must also then provide definitions for portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE() to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.

Setting configGENERATE_RUN_TIME_STATS to 1 will result in a total accumulated execution time being stored for each task. The resolution of the accumulated time value depends on the frequency of the timer configured by the portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() macro. Calling vTaskGetRunTimeStats() writes the total execution time of each task into a buffer, both as an absolute count value and as a percentage of the total system execution time.

Note This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

vTaskGetRunTimeStats() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays the amount of time each task has spent in the Running state in both absolute and percentage terms.

Note This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskGetRunTimeStats() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskGetRunTimeStats().

Parameters

- `pcWriteBuffer`: A buffer into which the execution times will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

BaseType_t **xTaskNotify** (*TaskHandle_t xTaskToNotify*, uint32_t *ulValue*, *eNotifyAction eAction*)

Send task notification.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

When configUSE_TASK_NOTIFICATIONS is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (uint32_t).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWait() or ulTaskNotifyTake(). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use xTaskNotifyWait() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTake() to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See http://www.FreeRTOS.org/RTOS-task-notifications.html for details.

Return Dependent on the value of eAction. See the description of the eAction parameter.

Parameters

- `xTaskToNotify`: The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().
- `ulValue`: Data that can be sent with the notification. How the data is used depends on the value of the eAction parameter.
- `eAction`: Specifies how the notification updates the task's notification value, if at all. Valid values for eAction are as follows:
  - eSetBits: The task's notification value is bitwise ORed with ulValue. xTaskNofify() 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. xTaskNofify() 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 );
```

(continues on next page)

(continued from previous page)

```
    } 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 witin 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;
```

(continues on next page)

(continued from previous page)

```
void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

// ...

if( xQueue1 != 0 )
{
    // Send an uint32_t.  Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueGenericSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10,␣
↪queueSEND_TO_BACK ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
}

if( xQueue2 != 0 )
{
    // Send a pointer to a struct AMessage object.  Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueGenericSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0,␣
↪queueSEND_TO_BACK );
}

// ... Rest of task code.
}
```

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 )
```

(continues on next page)

(continued from previous page)

```
{
struct AMessage *pxRxedMessage;

   if( xQueue != 0 )
   {
       // Receive a message on the created queue.  Block for 10 ticks if a
       // message is not immediately available.
       if( xQueueGenericReceive( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
       {
           // pcRxedMessage now points to the struct AMessage variable posted
           // by vATask.
       }
   }

   // ... Rest of task code.
}
```

Return pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Parameters

- `xQueue`: The handle to the queue from which the item is to be received.
- `pvBuffer`: Pointer to the buffer into which the received item will be copied.
- `xTicksToWait`: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. xQueueGenericReceive() will return immediately if the queue is empty and xTicksToWait is 0.
- `xJustPeek`: When set to true, the item received from the queue is not actually removed from the queue - meaning a subsequent call to xQueueReceive() will return the same item. When set to false, the item being received from the queue is also removed from the queue.

UBaseType_t **uxQueueMessagesWaiting**(**const** *QueueHandle_t xQueue*)

Return the number of messages stored in a queue.

Return The number of messages available in the queue.

Parameters

- `xQueue`: A handle to the queue being queried.

UBaseType_t **uxQueueSpacesAvailable**(**const** *QueueHandle_t xQueue*)

Return the number of free spaces available in a queue. This is equal to the number of items that can be sent to the queue before the queue becomes full if no items are removed.

Return The number of spaces available in the queue.

Parameters

- `xQueue`: A handle to the queue being queried.

void **vQueueDelete**(*QueueHandle_t xQueue*)

Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

Parameters

- `xQueue`: A handle to the queue to be deleted.

BaseType_t **xQueueReceiveFromISR**(*QueueHandle_t xQueue*, void ***const** *pvBuffer*, BaseType_t ***const** *pxHigherPriorityTaskWoken*)

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Example usage:

```
QueueHandle_t xQueue;

// Function to create a queue and post some values.
void vAFunction( void *pvParameters )
{
char cValueToPost;
const TickType_t xTicksToWait = ( TickType_t )0xff;

   // 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 cRxedChar;

   while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxedChar, &
↪xTaskWokenByReceive) )
   {
       // A character was received.  Output the character now.
       vOutputCharacter( cRxedChar );

       // If removing the character from the queue woke the task that was
       // posting onto the queue 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;
```

(continues on next page)

(continued from previous page)

```
    {
        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 creaet 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 creaet 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://wwwFreeRTOS.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://wwwFreeRTOS.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 ];
};

#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 * uxItemsSize) bytes. If uxItemSize is zero then pucQueueStorageBuffer can be NULL.
- `pxQueueBuffer`: Must point to a variable of type StaticQueue_t, which will be used to hold the queue's data structure.

xQueueSendToFront (xQueue, pvItemToQueue, xTicksToWait)

This is a macro that calls xQueueGenericSend().

Post an item to the front of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Example usage:

```
struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
} xMessage;

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 );
}
```

(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.

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.
```

(continues on next page)

(continued from previous page)

```
    }
}

if( xQueue2 != 0 )
{
    // Send a pointer to a struct AMessage object.  Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
}

// ... Rest of task code.
}
```

Return pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Parameters

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- `xTicksToWait`: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

`xQueueOverwrite`(xQueue, pvItemToQueue)

Only for use with queues that have a length of one - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR () for an alternative which may be used in an ISR.

Example usage:

```
void vFunction( void *pvParameters )
{
QueueHandle_t xQueue;
uint32_t ulVarToSend, ulValReceived;

// Create a queue to hold one uint32_t value.  It is strongly
// recommended *not* to use xQueueOverwrite() on queues that can
// contain more than one value, and doing so will trigger an assertion
// if configASSERT() is defined.
xQueue = xQueueCreate( 1, sizeof( uint32_t ) );

// Write the value 10 to the queue using xQueueOverwrite().
ulVarToSend = 10;
xQueueOverwrite( xQueue, &ulVarToSend );

// Peeking the queue should now return 10, but leave the value 10 in
// the queue.  A block time of zero is used as it is known that the
// queue holds a value.
```

(continues on next page)

(continued from previous page)

```
 ulValReceived = 0;
 xQueuePeek( xQueue, &ulValReceived, 0 );

 if( ulValReceived != 10 )
 {
     // Error unless the item was removed by a different task.
 }

 // The queue is still full.  Use xQueueOverwrite() to overwrite the
 // value held in the queue with 100.
 ulVarToSend = 100;
 xQueueOverwrite( xQueue, &ulVarToSend );

 // This time read from the queue, leaving the queue empty once more.
 // A block time of 0 is used again.
 xQueueReceive( xQueue, &ulValReceived, 0 );

 // The value read should be the last value written, even though the
 // queue was already full when the value was written.
 if( ulValReceived != 100 )
 {
     // Error!
 }

 // ...
}
```

Return xQueueOverwrite() is a macro that calls xQueueGenericSend(), and therefore has the same return values as xQueueSendToFront(). However, pdPASS is the only value that can be returned because xQueueOverwrite() will write to the queue even when the queue is already full.

Parameters

- `xQueue`: The handle of the queue to which the data is being sent.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.

xQueuePeek (xQueue, pvBuffer, xTicksToWait)

This is a macro that calls the xQueueGenericReceive() function.

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueueReceive().

This macro must not be used in an interrupt service routine. See xQueuePeekFromISR() for an alternative that can be called from an interrupt service routine.

Example usage:

```
struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
```

(continues on next page)

(continued from previous page)

```
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
if( xQueue == 0 )
{
    // Failed to create the queue.
}

// ...

// Send a pointer to a struct AMessage object.  Don't block if the
// queue is already full.
pxMessage = & xMessage;
xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

// ... Rest of task code.
}

// Task to peek the data from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;

if( xQueue != 0 )
{
    // Peek a message on the created queue.  Block for 10 ticks if a
    // message is not immediately available.
    if( xQueuePeek( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
    {
        // pcRxedMessage now points to the struct AMessage variable posted
        // by vATask, but the item still remains on the queue.
    }
}

// ... Rest of task code.
}
```

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 *pxRxedMessage;

 if( xQueue != 0 )
 {
     // Receive a message on the created queue.  Block for 10 ticks if a
     // message is not immediately available.
     if( xQueueReceive( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
     {
         // pcRxedMessage now points to the struct AMessage variable posted
         // by vATask.
     }
```

(continues on next page)

(continued from previous page)

```
 }

 // ... Rest of task code.
}
```

Return pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Parameters

- `xQueue`: The handle to the queue from which the item is to be received.
- `pvBuffer`: Pointer to the buffer into which the received item will be copied.
- `xTicksToWait`: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. xQueueReceive() will return immediately if xTicksToWait is zero and the queue is empty. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

xQueueSendToFrontFromISR(xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the front of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPrioritTaskWoken;

   // 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;
```

(continues on next page)

(continued from previous page)

```
void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to xSemaphoreCreateBinary().
   // The semaphore's data structures will be placed in the xSemaphoreBuffer
   // variable, the address of which is passed into the function.  The
   // function's parameter is not NULL, so the function will not attempt any
   // dynamic memory allocation, and therefore the function will not return
   // return NULL.
   xSemaphore = xSemaphoreCreateBinary( &xSemaphoreBuffer );

   // Rest of task code goes here.
}
```

Return If the semaphore is created then a handle to the created semaphore is returned. If pxSemaphoreBuffer is NULL then NULL is returned.

Parameters

- `pxStaticSemaphore`: Must point to a variable of type StaticSemaphore_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically.

`xSemaphoreTake`(xSemaphore, xBlockTime)

Macro to obtain a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

// A task that creates a semaphore.
void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.
   vSemaphoreCreateBinary( xSemaphore );
}

// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
   // ... Do other things.

   if( xSemaphore != NULL )
   {
       // 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 );
```

(continues on next page)

(continued from previous page)

```
        }
        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.
```

(continues on next page)

(continued from previous page)

```
            // ...
            // For some reason due to the nature of the code further calls to
            // xSemaphoreTakeRecursive() are made on the same mutex.  In real
            // code these would not be just sequential calls as this would make
            // no sense.  Instead the calls are likely to be buried inside
            // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

            // The mutex has now been 'taken' three times, so will not be
            // available to another task until it has also been given back
            // three times.  Again it is unlikely that real code would have
            // these calls sequentially, but instead buried in a more complex
            // call structure.  This is just for illustrative purposes.
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );

            // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}
```

Return pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

Parameters

- `xMutex`: A handle to the mutex being obtained. This is the handle returned by xSemaphoreCreateRecursiveMutex();
- `xBlockTime`: The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If the task already owns the semaphore then xSemaphoreTakeRecursive() will return immediately no matter what the value of xBlockTime.

xSemaphoreGive(xSemaphore)

Macro to release a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(). and obtained using sSemaphoreTake().

This macro must not be used from an ISR. See xSemaphoreGiveFromISR () for an alternative which can be used from an ISR.

This macro must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

void vATask( void * pvParameters )
```

(continues on next page)

(continued from previous page)

```
{
   // Create the semaphore to guard a shared resource.
   vSemaphoreCreateBinary( xSemaphore );

   if( xSemaphore != NULL )
   {
       if( xSemaphoreGive( xSemaphore ) != pdTRUE )
       {
           // We would expect this call to fail because we cannot give
           // a semaphore without first "taking" it!
       }

       // Obtain the semaphore - don't block if the semaphore is not
       // immediately available.
       if( xSemaphoreTake( xSemaphore, ( TickType_t ) 0 ) )
       {
           // We now have the semaphore and can access the shared resource.

           // ...

           // We have finished accessing the shared resource so can free the
           // semaphore.
           if( xSemaphoreGive( xSemaphore ) != pdTRUE )
           {
               // We would not expect this call to fail because we must have
               // obtained the semaphore to get here.
           }
       }
   }
}
```

Return pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

Parameters

- `xSemaphore`: A handle to the semaphore being released. This is the handle returned when the semaphore was created.

`xSemaphoreGiveRecursive`(xMutex)

Macro to recursively release, or 'give', a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;
```

(continues on next page)

(continued from previous page)

```
// 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( 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;
   }
```

(continues on next page)

(continued from previous page)

```
    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 provided the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See 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 provided the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
   // A mutex cannot be used before it has been created.  xMutexBuffer is
```

(continues on next page)

(continued from previous page)

```
    // 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 mutexs 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.
```

(continues on next page)

(continued from previous page)

```
        // 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 mutexs 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 xSemaphoreCreateCounting() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateCounting() function. (see http://www.freertos.org/a00111.html). If a counting semaphore is created using xSemaphoreCreateCountingStatic() then the application writer must provide the memory. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore = NULL;

   // Counting semaphore cannot be used before they have been created.  Create
   // a counting semaphore using xSemaphoreCreateCountingStatic().  The max
   // value to which the semaphore can count is 10, and the initial value
   // assigned to the count will be 0.  The address of xSemaphoreBuffer is
   // passed in and will be used to hold the semaphore structure, so no dynamic
   // memory allocation will be used.
   xSemaphore = xSemaphoreCreateCounting( 10, 0, &xSemaphoreBuffer );

   // No memory allocation was attempted so xSemaphore cannot be NULL, so there
   // is no need to check its value.
}
```

Return If the counting semaphore was successfully created then a handle to the created counting semaphore is returned. If pxSemaphoreBuffer was NULL then NULL is returned.

Parameters

- `uxMaxCount`: The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.
- `uxInitialCount`: The count value assigned to the semaphore when it is created.
- `pxSemaphoreBuffer`: Must point to a variable of type StaticSemaphore_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically.

vSemaphoreDelete(xSemaphore)

Delete a semaphore. This function must be used with care. For example, do not delete a mutex type semaphore if the mutex is held by a task.

Parameters

- `xSemaphore`: A handle to the semaphore to be deleted.

xSemaphoreGetMutexHolder(xSemaphore)

If xMutex is indeed a mutex type semaphore, return the current mutex holder. If xMutex is not a mutex type semaphore, or the mutex is available (not held by a task), return NULL.

Note: This is a good way of determining if the calling task is the mutex holder, but not a good way of determining the identity of the mutex holder as the holder may change between the function exiting and the returned value being tested.

uxSemaphoreGetCount(xSemaphore)

If the semaphore is a counting semaphore then uxSemaphoreGetCount() returns its current count value. If the semaphore is a binary semaphore then uxSemaphoreGetCount() returns 1 if the semaphore is available, and 0 if the semaphore is not available.

Type Definitions

typedef *QueueHandle_t* **SemaphoreHandle_t**

Timer API

Header File

- freertos/include/freertos/timers.h

Functions

TimerHandle_t **xTimerCreate**(**const** char ***const** *pcTimerName*, **const** TickType_t *xTimerPeriodInTicks*, **const** UBaseType_t *uxAutoReload*, void ***const** *pvTimerID*, *TimerCallbackFunction_t pxCallbackFunction*)

Creates a new software timer instance, and returns a handle by which the created software timer can be referenced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using xTimerCreate() then the required memory is automatically dynamically allocated inside the xTimerCreate() function. (see http://www.freertos.org/a00111.html). If a software timer is created using xTimerCreateStatic() then the application writer must provide the memory

that will get used by the software timer. xTimerCreateStatic() therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Example usage:

```
#define NUM_TIMERS 5

// An array to hold handles to the created timers.
TimerHandle_t xTimers[ NUM_TIMERS ];

// An array to hold a count of the number of times each timer expires.
int32_t lExpireCounters[ NUM_TIMERS ] = { 0 };

// Define a callback function that will be used by multiple timer instances.
// The callback function does nothing but count the number of times the
// associated timer expires, and stop the timer once the timer has expired
// 10 times.
void vTimerCallback( TimerHandle_t pxTimer )
{
int32_t lArrayIndex;
const int32_t xMaxExpiryCountBeforeStopping = 10;

       // Optionally do something if the pxTimer parameter is NULL.
       configASSERT( pxTimer );

    // Which timer expired?
    lArrayIndex = ( int32_t ) pvTimerGetTimerID( pxTimer );

    // Increment the number of times that pxTimer has expired.
    lExpireCounters[ lArrayIndex ] += 1;

    // If the timer has expired 10 times then stop it from running.
    if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
    {
        // Do not use a block time if calling a timer API function from a
        // timer callback function, as doing so could cause a deadlock!
        xTimerStop( pxTimer, 0 );
    }
}

void main( void )
{
int32_t x;

    // Create then start some timers.  Starting the timers before the scheduler
    // has been started means the timers will start running immediately that
    // the scheduler starts.
    for( x = 0; x < NUM_TIMERS; x++ )
    {
        xTimers[ x ] = xTimerCreate(    "Timer",       // Just a text name, not␣
↪used by the kernel.
                                        ( 100 * x ),   // The timer period in␣
↪ticks.
                                        pdTRUE,        // The timers will auto-
↪reload themselves when they expire.
```

(continues on next page)

(continued from previous page)

```
                                        ( void * ) x,  // Assign each timer a␣
↪unique id equal to its array index.
                                        vTimerCallback // Each timer calls the␣
↪same callback when it expires.
                                    );

        if( xTimers[ x ] == NULL )
        {
            // The timer was not created.
        }
        else
        {
            // Start the timer.  No block time is specified, and even if one was
            // it would be ignored because the scheduler has not yet been
            // started.
            if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
            {
                // The timer could not be set into the Active state.
            }
        }
    }

    // ...
    // Create tasks here.
    // ...

    // Starting the scheduler will start the timers running as they have already
    // been set into the active state.
    vTaskStartScheduler();

    // Should not reach here.
    for( ;; );
}
```

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 / 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
```

(continues on next page)

(continued from previous page)

```
    // to the StaticTimer_t structure that will hold the software timer
    // structure.  If the parameter is passed as NULL then the structure will be
    // allocated dynamically, just as if xTimerCreate() had been called.
    xTimer = xTimerCreateStatic( "T1",             // Text name for the task. ␣
↪Helps debugging only.  Not used by FreeRTOS.
                                 xTimerPeriod,     // The period of the timer in␣
↪ticks.
                                 pdTRUE,           // This is an auto-reload␣
↪timer.
                                 ( void * ) &uxVariableToIncrement,    // A␣
↪variable incremented by the software timer's callback function
                                 prvTimerCallback, // The function to execute␣
↪when the timer expires.
                                 &xTimerBuffer );  // The buffer that will hold␣
↪the software timer structure.

    // The scheduler has not started yet so a block time is not used.
    xReturned = xTimerStart( xTimer, 0 );

    // ...
    // Create tasks here.
    // ...

    // Starting the scheduler will start the timers running as they have already
    // been set into the active state.
    vTaskStartScheduler();

    // Should not reach here.
    for( ;; );
}
```

Return If the timer is created then a handle to the created timer is returned. If pxTimerBuffer was NULL then NULL is returned.

Parameters

- `pcTimerName`: A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
- `xTimerPeriodInTicks`: The timer period. The time is defined in tick periods so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xTimerPeriodInTicks should be set to 100. Alternatively, if the timer must expire after 500ms, then xPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
- `uxAutoReload`: If uxAutoReload is set to pdTRUE then the timer will expire repeatedly with a frequency set by the xTimerPeriodInTicks parameter. If uxAutoReload is set to pdFALSE then the timer will be a one-shot timer and enter the dormant state after it expires.
- `pvTimerID`: An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
- `pxCallbackFunction`: The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is "void vCallbackFunction(TimerHandle_t xTimer);".
- `pxTimerBuffer`: Must point to a variable of type StaticTimer_t, which will be then be used to hold

the software timer's data structures, removing the need for the memory to be allocated dynamically.

void ***pvTimerGetTimerID** (*TimerHandle_t xTimer*)

Returns the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used within the callback function to identify which timer actually expired.

Example usage:

Return The ID assigned to the timer being queried.

Parameters

- `xTimer`: The timer being queried.

See the xTimerCreate() API function example usage scenario.

void **vTimerSetTimerID** (*TimerHandle_t xTimer*, void **pvNewID*)

Sets the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used as time specific (timer local) storage.

Example usage:

Parameters

- `xTimer`: The timer being updated.
- `pvNewID`: The ID to assign to the timer.

See the xTimerCreate() API function example usage scenario.

BaseType_t **xTimerIsTimerActive** (*TimerHandle_t xTimer*)

Queries a timer to see if it is active or dormant.

A timer will be dormant if:

```
1) It has been created but not started, or

2) It is an expired one-shot timer that has not been restarted.
```

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Example usage:

```
// This function assumes xTimer has already been created.
void vAFunction( TimerHandle_t xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and␣
↪equivalently "if( xTimerIsTimerActive( xTimer ) )"
    {
        // xTimer is active, do something.
    }
```

(continues on next page)

(continued from previous page)

```
    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 it 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 )
{
   BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;

   // Query the hardware to determine which interface needs processing.
   xInterfaceToService = prvCheckInterfaces();

    // The actual processing is to be deferred to a task.  Request the
    // vProcessInterface() callback function is executed, passing in the
   // number of the interface that needs processing.  The interface to
   // service is passed in the second parameter.  The first parameter is
   // not used in this case.
   xHigherPriorityTaskWoken = pdFALSE;
   xTimerPendFunctionCallFromISR( vProcessInterface, NULL, ( uint32_t )␣
↪xInterfaceToService, &xHigherPriorityTaskWoken );

   // If xHigherPriorityTaskWoken is now set to pdTRUE then a context
   // switch should be requested.  The macro used is port specific and will
   // be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
   // the documentation page for the port being used.
   portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

Return pdPASS is returned if the message was successfully sent to the timer daemon task, otherwise pdFALSE is returned.

Parameters

- `xFunctionToPend`: The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype.
- `pvParameter1`: The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void *, or the void * can be used to point to a structure.
- `ulParameter2`: The value of the callback function's second parameter.
- `pxHigherPriorityTaskWoken`: As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task (which is set using configTIMER_TASK_PRIORITY in FreeRTOSConfig.h) is higher than the priority of the currently running task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE within xTimerPendFunctionCallFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialised

to pdFALSE. See the example code below.

BaseType_t **xTimerPendFunctionCall** (*PendedFunction_t xFunctionToPend*, void **pvParameter1*, uint32_t *ulParameter2*, TickType_t *xTicksToWait*)

Used to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

Return pdPASS is returned if the message was successfully sent to the timer daemon task, otherwise pdFALSE is returned.

Parameters

- `xFunctionToPend`: The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype.
- `pvParameter1`: The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void *, or the void * can be used to point to a structure.
- `ulParameter2`: The value of the callback function's second parameter.
- `xTicksToWait`: Calling this function will result in a message being sent to the timer daemon task on a queue. xTicksToWait is the amount of time the calling task should remain in the Blocked state (so not using any processing time) for space to become available on the timer queue if the queue is found to be full.

const char ***pcTimerGetTimerName** (*TimerHandle_t xTimer*)

Returns the name that was assigned to a timer when the timer was created.

Return The name assigned to the timer specified by the xTimer parameter.

Parameters

- `xTimer`: The handle of the timer being queried.

Macros

tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR

tmrCOMMAND_EXECUTE_CALLBACK

tmrCOMMAND_START_DONT_TRACE

tmrCOMMAND_START

tmrCOMMAND_RESET

tmrCOMMAND_STOP

tmrCOMMAND_CHANGE_PERIOD

tmrCOMMAND_DELETE

tmrFIRST_FROM_ISR_COMMAND

tmrCOMMAND_START_FROM_ISR

tmrCOMMAND_RESET_FROM_ISR

tmrCOMMAND_STOP_FROM_ISR

tmrCOMMAND_CHANGE_PERIOD_FROM_ISR

xTimerStart (xTimer, xTicksToWait)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerStart() starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerStart() has equivalent functionality to the xTimerReset() API function.

Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after xTimerStart() was called, where 'n' is the timers defined period.

It is valid to call xTimerStart() before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when xTimerStart() was called.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerStart() to be available.

Example usage:

Return pdFAIL will be returned if the start command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer being started/restarted.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when xTimerStart() was called. xTicksToWait is ignored if xTimerStart() is called before the scheduler is started.

See the xTimerCreate() API function example usage scenario.

xTimerStop (xTimer, xTicksToWait)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerStop() stops a timer that was previously started using either of the The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() or xTimerChangePeriodFromISR() API functions.

Stopping a timer ensures the timer is not in the active state.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerStop() to be available.

Example usage:

Return pdFAIL will be returned if the stop command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer being stopped.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue, should the queue already be full when xTimerStop() was called. xTicksToWait is ignored if xTimerStop() is called before the scheduler is started.

See the xTimerCreate() API function example usage scenario.

xTimerChangePeriod(xTimer, xNewPeriod, xTicksToWait)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerChangePeriod() changes the period of a timer that was previously created using the xTimerCreate() API function.

xTimerChangePeriod() can be called to change the period of an active or dormant state timer.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerChangePeriod() to be available.

Example usage:

```
// This function assumes xTimer has already been created.  If the timer
// referenced by xTimer is already active when it is called, then the timer
// is deleted.  If the timer referenced by xTimer is not active when it is
// called, then the period of the timer is set to 500ms and the timer is
// started.
void vAFunction( TimerHandle_t xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and␣
↪equivalently "if( xTimerIsTimerActive( xTimer ) )"
    {
        // xTimer is already active - delete it.
        xTimerDelete( xTimer );
    }
    else
    {
        // xTimer is not active, change its period to 500ms.  This will also
        // cause the timer to start.  Block for a maximum of 100 ticks if the
        // change period command cannot immediately be sent to the timer
        // command queue.
        if( xTimerChangePeriod( xTimer, 500 / portTICK_PERIOD_MS, 100 ) == pdPASS␣
↪)
        {
            // The command was successfully sent.
        }
        else
        {
            // The command could not be sent, even after waiting for 100 ticks
            // to pass.  Take appropriate action here.
        }
    }
}
```

Return pdFAIL will be returned if the change period command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent

to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer that is having its period changed.
- `xNewPeriod`: The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the change period command to be successfully sent to the timer command queue, should the queue already be full when xTimerChangePeriod() was called. xTicksToWait is ignored if xTimerChangePeriod() is called before the scheduler is started.

xTimerDelete(xTimer, xTicksToWait)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerDelete() deletes a timer that was previously created using the xTimerCreate() API function.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerDelete() to be available.

Example usage:

Return pdFAIL will be returned if the delete command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer being deleted.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the delete command to be successfully sent to the timer command queue, should the queue already be full when xTimerDelete() was called. xTicksToWait is ignored if xTimerDelete() is called before the scheduler is started.

See the xTimerChangePeriod() API function example usage scenario.

xTimerReset(xTimer, xTicksToWait)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerReset() re-starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerReset() will cause the timer to re-evaluate its expiry time so that it is relative to when xTimerReset() was called. If the timer was in the dormant state then xTimerReset() has equivalent functionality to the xTimerStart() API function.

Resetting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after xTimerReset() was called,

where 'n' is the timers defined period.

It is valid to call xTimerReset() before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when xTimerReset() was called.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerReset() to be available.

Example usage:

```
// When a key is pressed, an LCD back-light is switched on.  If 5 seconds pass
// without a key being pressed, then the LCD back-light is switched off.  In
// this case, the timer is a one-shot timer.

TimerHandle_t xBacklightTimer = NULL;

// The callback function assigned to the one-shot timer.  In this case the
// parameter is not used.
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    // The timer expired, therefore 5 seconds must have passed since a key
    // was pressed.  Switch off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

// The key press event handler.
void vKeyPressEventHandler( char cKey )
{
    // Ensure the LCD back-light is on, then reset the timer that is
    // responsible for turning the back-light off after 5 seconds of
    // key inactivity.  Wait 10 ticks for the command to be successfully sent
    // if it cannot be sent immediately.
    vSetBacklightState( BACKLIGHT_ON );
    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.
                                  );
```

(continues on next page)

(continued from previous page)

```
    if( xBacklightTimer == NULL )
    {
        // The timer was not created.
    }
    else
    {
        // Start the timer.  No block time is specified, and even if one was
        // it would be ignored because the scheduler has not yet been
        // started.
        if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
        {
            // The timer could not be set into the Active state.
        }
    }

    // ...
    // Create tasks here.
    // ...

    // Starting the scheduler will start the timer running as it has already
    // been set into the active state.
    xTaskStartScheduler();

    // Should not reach here.
    for( ;; );
}
```

Return pdFAIL will be returned if the reset command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer being reset/started/restarted.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the reset command to be successfully sent to the timer command queue, should the queue already be full when xTimerReset() was called. xTicksToWait is ignored if xTimerReset() is called before the scheduler is started.

xTimerStartFromISR(xTimer, pxHigherPriorityTaskWoken)

A version of xTimerStart() that can be called from an interrupt service routine.

Example usage:

```
// This scenario assumes xBacklightTimer has already been created.  When a
// key is pressed, an LCD back-light is switched on.  If 5 seconds pass
// without a key being pressed, then the LCD back-light is switched off.  In
// this case, the timer is a one-shot timer, and unlike the example given for
// the xTimerReset() function, the key press event handler is an interrupt
// service routine.
```

(continues on next page)

(continued from previous page)

```
// The callback function assigned to the one-shot timer.  In this case the
// parameter is not used.
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    // The timer expired, therefore 5 seconds must have passed since a key
    // was pressed.  Switch off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

// The key press interrupt service routine.
void vKeyPressEventInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // Ensure the LCD back-light is on, then restart the timer that is
    // responsible for turning the back-light off after 5 seconds of
    // key inactivity.  This is an interrupt service routine so can only
    // call FreeRTOS API functions that end in "FromISR".
    vSetBacklightState( BACKLIGHT_ON );

    // xTimerStartFromISR() or xTimerResetFromISR() could be called here
    // as both cause the timer to re-calculate its expiry time.
    // xHigherPriorityTaskWoken was initialised to pdFALSE when it was
    // declared (in this function).
    if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) !=␣
↪pdPASS )
    {
        // The start command was not executed successfully.  Take appropriate
        // action here.
    }

    // Perform the rest of the key processing here.

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed.  The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler.  Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

Return pdFAIL will be returned if the start command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStartFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer being started/restarted.
- `pxHigherPriorityTaskWoken`: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStart-

FromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStartFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStartFromISR() function. If xTimerStartFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

xTimerStopFromISR(xTimer, pxHigherPriorityTaskWoken)

A version of xTimerStop() that can be called from an interrupt service routine.

Example usage:

```
// This scenario assumes xTimer has already been created and started.  When
// an interrupt occurs, the timer should be simply stopped.

// The interrupt service routine that stops the timer.
void vAnExampleInterruptServiceRoutine( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // The interrupt has occurred - simply stop the timer.
    // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
    // (within this function).  As this is an interrupt service routine, only
    // FreeRTOS API functions that end in "FromISR" can be used.
    if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        // The stop command was not executed successfully.  Take appropriate
        // action here.
    }

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed.  The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler.  Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

Return pdFAIL will be returned if the stop command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer being stopped.
- `pxHigherPriorityTaskWoken`: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStopFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStopFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority

equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStopFromISR() function. If xTimerStopFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

xTimerChangePeriodFromISR(xTimer, xNewPeriod, pxHigherPriorityTaskWoken)

A version of xTimerChangePeriod() that can be called from an interrupt service routine.

Example usage:

```
// This scenario assumes xTimer has already been created and started.  When
// an interrupt occurs, the period of xTimer should be changed to 500ms.

// The interrupt service routine that changes the period of xTimer.
void vAnExampleInterruptServiceRoutine( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // The interrupt has occurred - change the period of xTimer to 500ms.
    // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
    // (within this function).  As this is an interrupt service routine, only
    // FreeRTOS API functions that end in "FromISR" can be used.
    if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        // The command to change the timers period was not executed
        // successfully.  Take appropriate action here.
    }

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed.  The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler.  Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

Return pdFAIL will be returned if the command to change the timers period could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer that is having its period changed.
- `xNewPeriod`: The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / 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 )
```

(continues on next page)

(continued from previous page)

```
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

Return pdFAIL will be returned if the reset command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerResetFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer that is to be started, reset, or restarted.
- `pxHigherPriorityTaskWoken`: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerResetFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerResetFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerResetFromISR() function. If xTimerResetFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Type Definitions

typedef void ***TimerHandle_t**

Type by which software timers are referenced. For example, a call to xTimerCreate() returns an TimerHandle_t variable that can then be used to reference the subject timer in calls to other software timer API functions (for example, xTimerStart(), xTimerReset(), etc.).

typedef void (***TimerCallbackFunction_t**) (*TimerHandle_t* xTimer)

Defines the prototype to which timer callback functions must conform.

typedef void (***PendedFunction_t**) (void *, uint32_t)

Defines the prototype to which functions used with the xTimerPendFunctionCallFromISR() function must conform.

Event Group API

Header File

- freertos/include/freertos/event_groups.h

Functions

EventGroupHandle_t **xEventGroupCreate** (void)

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using xEventGroupCreate() then the required memory

is automatically dynamically allocated inside the xEventGroupCreate() function. (see http://www.freertos.org/a00111.html). If an event group is created using xEventGropuCreateStatic() then the application writer must instead provide the memory that will get used by the event group. xEventGroupCreateStatic() therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the configUSE_16_BIT_TICKS setting in FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If configUSE_16_BIT_TICKS is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The EventBits_t type is used to store event bits within an event group.

Example usage:

```
// Declare a variable to hold the created event group.
EventGroupHandle_t xCreatedEventGroup;

// Attempt to create the event group.
xCreatedEventGroup = xEventGroupCreate();

// Was the event group created successfully?
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
```

(continues on next page)

(continued from previous page)

```
// 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.
     }
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
        // called.  Both will now be clear (not set).
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // Bit 0 was set before xEventGroupClearBits() was called.  It will
        // now be clear.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // Bit 4 was set before xEventGroupClearBits() was called.  It will
        // now be clear.
    }
    else
    {
        // Neither bit 0 nor bit 4 were set in the first place.
    }
}
```

Return The value of the event group before the specified bits were cleared.

Parameters

- `xEventGroup`: The event group in which the bits are to be cleared.
- `uxBitsToClear`: A bitwise value that indicates the bit or bits to clear in the event group. For example, to clear bit 3 only, set uxBitsToClear to 0x08. To clear bit 3 and bit 0 set uxBitsToClear to 0x09.

EventBits_t **xEventGroupSetBits** (*EventGroupHandle_t xEventGroup*, **const** *EventBits_t uxBitsToSet*)

Set bits within an event group. This function cannot be called from an interrupt. xEventGroupSetBitsFromISR() is a version that can be called from an interrupt.

Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.

Example usage:

```
#define BIT_0     ( 1 << 0 )
#define BIT_4     ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

    // 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
```

(continues on next page)

(continued from previous page)

```
        // 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;
```

(continues on next page)

(continued from previous page)

```
void vTask0( void *pvParameters )
{
EventBits_t uxReturn;
TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

    for( ;; )
    {
       // Perform task functionality here.

       // Set bit 0 in the event flag to note this task has reached the
       // sync point.  The other two tasks will set the other two bits defined
       // by ALL_SYNC_BITS.  All three tasks have reached the synchronisation
       // point when all the ALL_SYNC_BITS are set.  Wait a maximum of 100ms
       // for this to happen.
       uxReturn = xEventGroupSync( xEventBits, TASK_0_BIT, ALL_SYNC_BITS,␣
↪xTicksToWait );

       if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
       {
           // All three tasks reached the synchronisation point before the call
           // to xEventGroupSync() timed out.
       }
   }
}

void vTask1( void *pvParameters )
{
    for( ;; )
    {
       // Perform task functionality here.

       // Set bit 1 in the event flag to note this task has reached the
       // synchronisation point.  The other two tasks will set the other two
       // bits defined by ALL_SYNC_BITS.  All three tasks have reached the
       // synchronisation point when all the ALL_SYNC_BITS are set.  Wait
       // indefinitely for this to happen.
       xEventGroupSync( xEventBits, TASK_1_BIT, ALL_SYNC_BITS, portMAX_DELAY );

       // xEventGroupSync() was called with an indefinite block time, so
       // this task will only reach here if the syncrhonisation 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 );
```

(continues on next page)

(continued from previous page)

```
        // xEventGroupSync() was called with an indefinite block time, so
        // this task will only reach here if the syncrhonisation 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.
     }
}
```

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.
```

(continues on next page)

(continued from previous page)

```
      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**

3.8.2 FreeRTOS Additions

Overview

ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0 with significant modifications for SMP compatibility (see *ESP-IDF FreeRTOS SMP Changes*). However various features specific to ESP-IDF FreeRTOS have been added. The features are as follows:

Ring Buffers: Ring buffers were added to provide a form of buffer that could accept entries of arbitrary lengths.

Hooks: ESP-IDF FreeRTOS hooks provides support for registering extra Idle and Tick hooks at run time. Moreover, the hooks can be asymmetric amongst both CPUs.

Ring Buffers

The ESP-IDF FreeRTOS ring buffer is a strictly FIFO buffer that supports arbitrarily sized items. Ring buffers are a more memory efficient alternative to FreeRTOS queues in situations where the size of items is variable. The capacity of a ring buffer is not measured by the number of items it can store, but rather by the amount of memory used for storing items. Items are sent to ring buffers by copy, however for efficiency reasons **items are retrieved by reference**. As a result, all retrieved items **must also be returned** in order for them to be removed from the ring buffer completely. The ring buffers are split into the three following types:

No-Split buffers will guarantee that an item is stored in contiguous memory and will not attempt to split an item under any circumstances. Use no-split buffers when items must occupy contiguous memory.

Allow-Split buffers will allow an item to be split when wrapping around if doing so will allow the item to be stored. Allow-split buffers are more memory efficient than no-split buffers but can return an item in two parts when retrieving.

Byte buffers do not store data as separate items. All data is stored as a sequence of bytes, and any number of bytes and be sent or retrieved each time. Use byte buffers when separate items do not need to be maintained (e.g. a byte stream).

Note: No-split/allow-split buffers will always store items at 32-bit aligned addresses. Therefore when retrieving an item, the item pointer is guaranteed to be 32-bit aligned.

Note: Each item stored in no-split/allow-split buffers will **require an additional 8 bytes for a header**. Item sizes will also be rounded up to a 32-bit aligned size (multiple of 4 bytes), however the true item size is recorded within the header. The sizes of no-split/allow-split buffers will also be rounded up when created.

Usage

The following example demonstrates the usage of `xRingbufferCreate()` and `xRingbufferSend()` to create a ring buffer then send an item to it.

```
#include "freertos/ringbuf.h"
static char tx_item[] = "test_item";

...

    //Create ring buffer
    RingbufHandle_t buf_handle;
    buf_handle = xRingbufferCreate(1028, RINGBUF_TYPE_NOSPLIT);
    if (buf_handle == NULL) {
        printf("Failed to create ring buffer\n");
    }

    //Send an item
    UBaseType_t res =  xRingbufferSend(buf_handle, tx_item, sizeof(tx_item), pdMS_TO_
↪TICKS(1000));
    if (res != pdTRUE) {
        printf("Failed to send item\n");
    }
```

The following example demonstrates retrieving and returning an item from a **no-split ring buffer** using `xRingbufferReceive()` and `vRingbufferReturnItem()`

```
...

    //Receive an item from no-split ring buffer
    size_t item_size;
    char *item = (char *)xRingbufferReceive(buf_handle, &item_size, pdMS_TO_
↪TICKS(1000));

    //Check received item
    if (item != NULL) {
        //Print item
        for (int i = 0; i < item_size; i++) {
            printf("%c", item[i]);
        }
        printf("\n");
        //Return Item
        vRingbufferReturnItem(buf_handle, (void *)item);
    } else {
        //Failed to receive item
        printf("Failed to receive item\n");
    }
```

The following example demonstrates retrieving and returning an item from an **allow-split ring buffer** using `xRingbufferReceiveSplit()` and `vRingbufferReturnItem()`

```
...

    //Receive an item from allow-split ring buffer
    size_t item_size1, item_size2;
    char *item1, *item2;
    BaseType_t ret = xRingbufferReceiveSplit(buf_handle, (void **)&item1, (void **)&
↪item2, &item_size1, &item_size2, pdMS_TO_TICKS(1000));
```

(continues on next page)

(continued from previous page)

```
    //Check received item
    if (ret == pdTRUE && item1 != NULL) {
        for (int i = 0; i < item_size1; i++) {
            printf("%c", item1[i]);
        }
        vRingbufferReturnItem(buf_handle, (void *)item1);
        //Check if item was split
        if (item2 != NULL) {
            for (int i = 0; i < item_size2; i++) {
                printf("%c", item2[i]);
            }
            vRingbufferReturnItem(buf_handle, (void *)item2);
        }
        printf("\n");
    } else {
        //Failed to receive item
        printf("Failed to receive item\n");
    }
```

The following example demonstrates retrieving and returning an item from a **byte buffer** using *xRingbufferReceiveUpTo()* and *vRingbufferReturnItem()*

```
...

    //Receive data from byte buffer
    size_t item_size;
    char *item = (char *)xRingbufferReceiveUpTo(buf_handle, &item_size, pdMS_TO_
↪TICKS(1000), sizeof(tx_item));

    //Check received data
    if (item != NULL) {
        //Print item
        for (int i = 0; i < item_size; i++) {
            printf("%c", item[i]);
        }
        printf("\n");
        //Return Item
        vRingbufferReturnItem(buf_handle, (void *)item);
    } else {
        //Failed to receive item
        printf("Failed to receive item\n");
    }
```

For ISR safe versions of the functions used above, call *xRingbufferSendFromISR()*, *xRingbufferReceiveFromISR()*, *xRingbufferReceiveSplitFromISR()*, *xRingbufferReceiveUpToFromISR()*, and *vRingbufferReturnItemFromISR()*

Sending to Ring Buffer

The following diagrams illustrate the differences between no-split/allow-split buffers and byte buffers with regards to sending items/data. The diagrams assume that three items of sizes **18, 3, and 27 bytes** are sent respectively to a **buffer of 128 bytes**.

For no-split/allow-split buffers, a header of 8 bytes precedes every data item. Furthermore, the space occupied by each item is **rounded up to the nearest 32-bit aligned size** in order to maintain overall 32-bit alignment. However the true

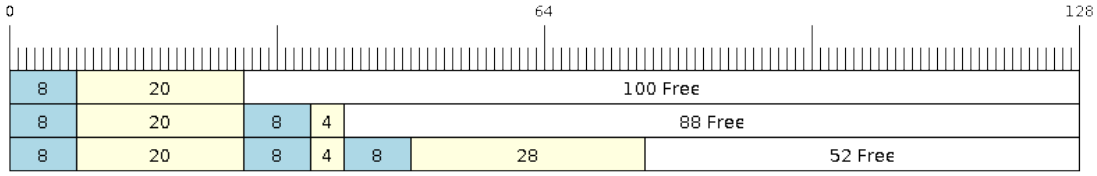


Fig. 12: Sending items to no-split/allow-split ring buffers

size of the item is recorded inside the header which will be returned when the item is retrieved.

Referring to the diagram above, the 18, 3, and 27 byte items are **rounded up to 20, 4, and 28 bytes** respectively. An 8 byte header is then added in front of each item.

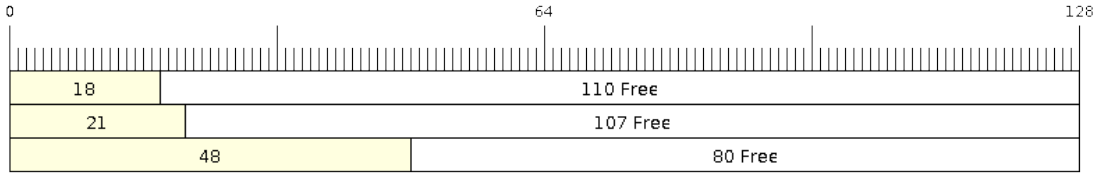


Fig. 13: Sending items to byte buffers

Byte buffers treat data as a sequence of bytes and does not incur any overhead (no headers). As a result, all data sent to a byte buffer is merged into a single item.

Referring to the diagram above, the 18, 3, and 27 byte items are sequentially written to the byte buffer and **merged into a single item of 48 bytes**.

Wrap around

The following diagrams illustrate the differences between no-split, allow-split, and byte buffers when a sent item requires a wrap around. The diagrams assumes a buffer of **128 bytes** with **56 bytes of free space that wraps around** and a sent item of **28 bytes**.

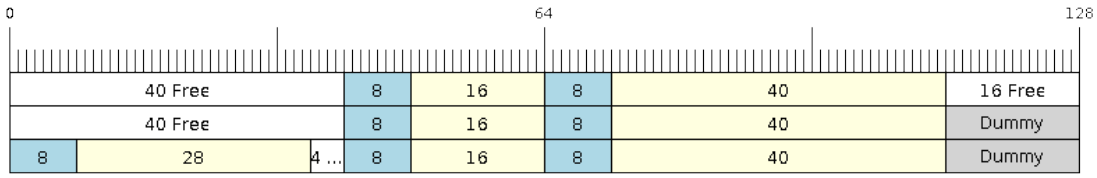


Fig. 14: Wrap around in no-split buffers

No-split buffers will **only store an item in continuous free space and will not split an item under any circumstances**. When the free space at the tail of the buffer is insufficient to completely store the item and its header, the

free space at the tail will be **marked as dummy data**. The buffer will then wrap around and store the item in the free space at the head of the buffer.

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to store the 28 byte item. Therefore the 16 bytes is marked as dummy data and the item is written to the free space at the head of the buffer instead.

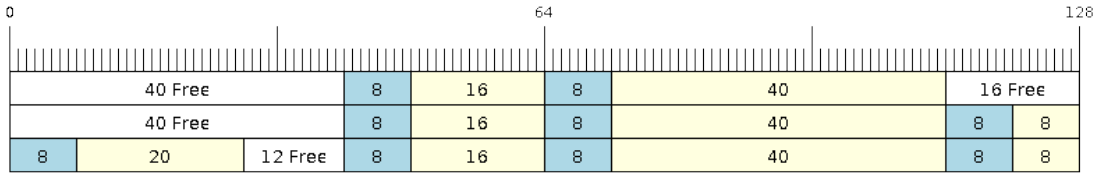


Fig. 15: Wrap around in allow-split buffers

Allow-split buffers will attempt to **split the item into two parts** when the free space at the tail of the buffer is insufficient to store the item data and its header. Both parts of the split item will have their own headers (therefore incurring an extra 8 bytes of overhead).

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to store the 28 byte item. Therefore the item is split into two parts (8 and 20 bytes) and written as two parts to the buffer.

Note: Allow-split buffers treats the both parts of the split item as two separate items, therefore call `xRingbufferReceiveSplit()` instead of `xRingbufferReceive()` to receive both parts of a split item in a thread safe manner.

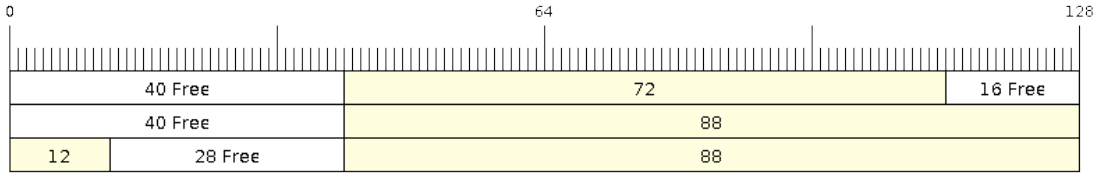


Fig. 16: Wrap around in byte buffers

Byte buffers will **store as much data as possible into the free space at the tail of buffer**. The remaining data will then be stored in the free space at the head of the buffer. No overhead is incurred when wrapping around in byte buffers.

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to completely store the 28 bytes of data. Therefore the 16 bytes of free space is filled with data, and the remaining 12 bytes are written to the free space at the head of the buffer. The buffer now contains data in two separate continuous parts, and each part continuous will be treated as a separate item by the byte buffer.

Retrieving/Returning

The following diagrams illustrates the differences between no-split/allow-split and byte buffers in retrieving and returning data.

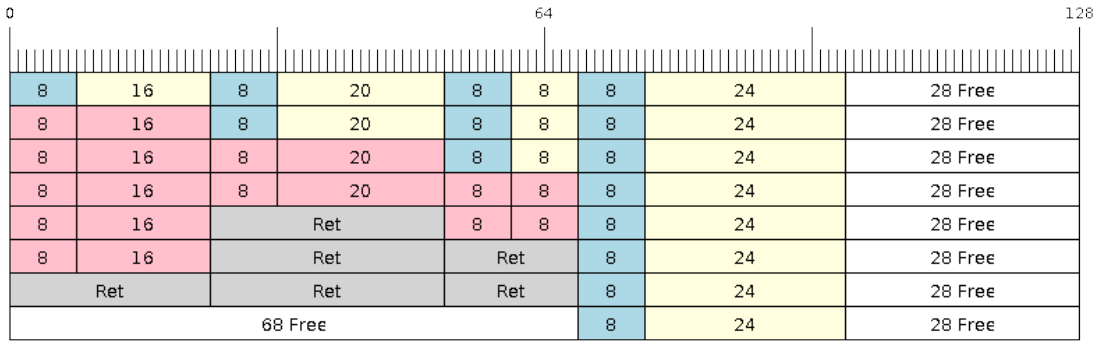


Fig. 17: Retrieving/Returning items in no-split/allow-split ring buffers

Items in no-split/allow-split buffers are **retrieved in strict FIFO order** and **must be returned** for the occupied space to be freed. Multiple items can be retrieved before returning, and the items do not necessarily need to be returned in the order they were retrieved. However the freeing of space must occur in FIFO order, therefore not returning the earliest retrieved item will prevent the space of subsequent items from being freed.

Referring to the diagram above, the **16, 20, and 8 byte items are retrieved in FIFO order**. However the items are not returned in they were retrieved (20, 8, 16). As such, the space is not freed until the first item (16 byte) is returned.

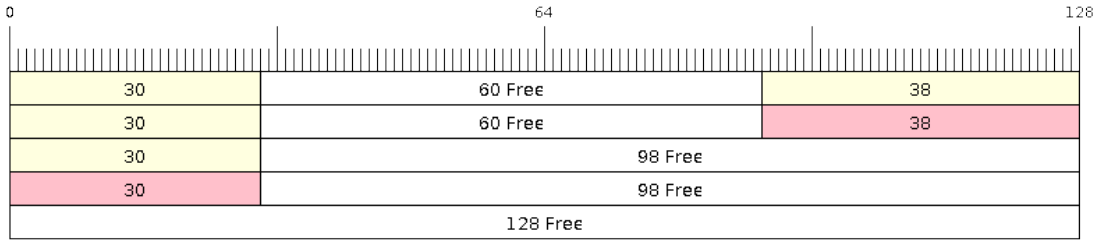


Fig. 18: Retrieving/Returning data in byte buffers

Byte buffers **do not allow multiple retrievals before returning** (every retrieval must be followed by a return before another retrieval is permitted). When using `xRingbufferReceive()` or `xRingbufferReceiveFromISR()`, all continuous stored data will be retrieved. `xRingbufferReceiveUpTo()` or `xRingbufferReceiveUpToFromISR()` can be used to restrict the maximum number of bytes retrieved. Since every retrieval must be followed by a return, the space will be freed as soon as the data is returned.

Referring to the diagram above, the 38 bytes of continuous stored data at the tail of the buffer is retrieved, returned, and freed. The next call to `xRingbufferReceive()` or `xRingbufferReceiveFromISR()` then wraps around and does the same to the 30 bytes of continuous stored data at the head of the buffer.

Ring Buffers with Queue Sets

Ring buffers can be added to FreeRTOS queue sets using `xRingbufferAddToQueueSetRead()` such that every time a ring buffer receives an item or data, the queue set is notified. Once added to a queue set, every attempt to

retrieve an item from a ring buffer should be preceded by a call to `xQueueSelectFromSet()`. To check whether the selected queue set member is the ring buffer, call `xRingbufferCanRead()`.

The following example demonstrates queue set usage with ring buffers.

```
#include "freertos/queue.h"
#include "freertos/ringbuf.h"

...

    //Create ring buffer and queue set
    RingbufHandle_t buf_handle = xRingbufferCreate(1028, RINGBUF_TYPE_NOSPLIT);
    QueueSetHandle_t queue_set = xQueueCreateSet(3);

    //Add ring buffer to queue set
    if (xRingbufferAddToQueueSetRead(buf_handle, queue_set) != pdTRUE) {
        printf("Failed to add to queue set\n");
    }

...

    //Block on queue set
    xQueueSetMemberHandle member = xQueueSelectFromSet(queue_set, pdMS_TO_
↪TICKS(1000));

    //Check if member is ring buffer
    if (member != NULL && xRingbufferCanRead(buf_handle, member) == pdTRUE) {
        //Member is ring buffer, receive item from ring buffer
        size_t item_size;
        char *item = (char *)xRingbufferReceive(buf_handle, &item_size, 0);

        //Handle item
        ...

    } else {
        ...
    }
```

Ring Buffer API Reference

Note: Ideally, ring buffers can be used with multiple tasks in an SMP fashion where the **highest priority task will always be serviced first.** However due to the usage of binary semaphores in the ring buffer's underlying implementation, priority inversion may occur under very specific circumstances.

The ring buffer governs sending by a binary semaphore which is given whenever space is freed on the ring buffer. The highest priority task waiting to send will repeatedly take the semaphore until sufficient free space becomes available or until it times out. Ideally this should prevent any lower priority tasks from being serviced as the semaphore should always be given to the highest priority task.

However in between iterations of acquiring the semaphore, there is a **gap in the critical section** which may permit another task (on the other core or with an even higher priority) to free some space on the ring buffer and as a result give the semaphore. Therefore the semaphore will be given before the highest priority task can re-acquire the semaphore. This will result in the **semaphore being acquired by the second highest priority task** waiting to send, hence causing priority inversion.

This side effect will not affect ring buffer performance drastically given if the number of tasks using the ring buffer

simultaneously is low, and the ring buffer is not operating near maximum capacity.

Header File

- freertos/include/freertos/ringbuf.h

Functions

RingbufHandle_t **xRingbufferCreate**(size_t *xBufferSize*, *ringbuf_type_t xBufferType*)

Create a ring buffer.

Note xBufferSize of no-split/allow-split buffers will be rounded up to the nearest 32-bit aligned size.

Return A handle to the created ring buffer, or NULL in case of error.

Parameters

- `xBufferSize`: Size of the buffer in bytes. Note that items require space for overhead in no-split/allow-split buffers
- `xBufferType`: Type of ring buffer, see documentation.

RingbufHandle_t **xRingbufferCreateNoSplit**(size_t *xItemSize*, size_t *xItemNum*)

Create a ring buffer of type RINGBUF_TYPE_NOSPLIT for a fixed item_size.

This API is similar to xRingbufferCreate(), but it will internally allocate additional space for the headers.

Return A RingbufHandle_t handle to the created ring buffer, or NULL in case of error.

Parameters

- `xItemSize`: Size of each item to be put into the ring buffer
- `xItemNum`: Maximum number of items the buffer needs to hold simultaneously

BaseType_t **xRingbufferSend**(*RingbufHandle_t xRingbuffer*, **const** void **pvItem*, size_t *xItemSize*, TickType_t *xTicksToWait*)

Insert an item into the ring buffer.

Attempt to insert an item into the ring buffer. This function will block until enough free space is available or until it timesout.

Note For no-split/allow-split ring buffers, the actual size of memory that the item will occupy will be rounded up to the nearest 32-bit aligned size. This is done to ensure all items are always stored in 32-bit aligned fashion.

Return

- pdTRUE if succeeded
- pdFALSE on time-out or when the data is larger than the maximum permissible size of the buffer

Parameters

- `xRingbuffer`: Ring buffer to insert the item into
- `pvItem`: Pointer to data to insert. NULL is allowed if xItemSize is 0.
- `xItemSize`: Size of data to insert.

- `xTicksToWait`: Ticks to wait for room in the ring buffer.

BaseType_t **xRingbufferSendFromISR**(*RingbufHandle_t xRingbuffer*, **const** void **pvItem*, size_t *xItemSize*, BaseType_t **pxHigherPriorityTaskWoken*)

Insert an item into the ring buffer in an ISR.

Attempt to insert an item into the ring buffer from an ISR. This function will return immediately if there is insufficient free space in the buffer.

Note For no-split/allow-split ring buffers, the actual size of memory that the item will occupy will be rounded up to the nearest 32-bit aligned size. This is done to ensure all items are always stored in 32-bit aligned fashion.

Return

- pdTRUE if succeeded
- pdFALSE when the ring buffer does not have space.

Parameters

- `xRingbuffer`: Ring buffer to insert the item into
- `pvItem`: Pointer to data to insert. NULL is allowed if xItemSize is 0.
- `xItemSize`: Size of data to insert.
- `pxHigherPriorityTaskWoken`: Value pointed to will be set to pdTRUE if the function woke up a higher priority task.

void ****xRingbufferReceive**(*RingbufHandle_t xRingbuffer*, size_t **pxItemSize*, TickType_t *xTicksToWait*)

Retrieve an item from the ring buffer.

Attempt to retrieve an item from the ring buffer. This function will block until an item is available or until it timesout.

Note A call to vRingbufferReturnItem() is required after this to free the item retrieved.

Return

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL on timeout, *pxItemSize is untouched in that case.

Parameters

- `xRingbuffer`: Ring buffer to retrieve the item from
- `pxItemSize`: Pointer to a variable to which the size of the retrieved item will be written.
- `xTicksToWait`: Ticks to wait for items in the ring buffer.

void ****xRingbufferReceiveFromISR**(*RingbufHandle_t xRingbuffer*, size_t **pxItemSize*)

Retrieve an item from the ring buffer in an ISR.

Attempt to retrieve an item from the ring buffer. This function returns immediately if there are no items available for retrieval

Note A call to vRingbufferReturnItemFromISR() is required after this to free the item retrieved.

Note Byte buffers do not allow multiple retrievals before returning an item

Return

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL when the ring buffer is empty, *pxItemSize is untouched in that case.

Parameters

- `xRingbuffer`: Ring buffer to retrieve the item from
- `pxItemSize`: Pointer to a variable to which the size of the retrieved item will be written.

BaseType_t **`xRingbufferReceiveSplit`**(*RingbufHandle_t xRingbuffer*, void **ppvHeadItem, void **ppvTailItem, size_t *pxHeadItemSize, size_t *pxTailItemSize, TickType_t xTicksToWait)

Retrieve a split item from an allow-split ring buffer.

Attempt to retrieve a split item from an allow-split ring buffer. If the item is not split, only a single item is retried. If the item is split, both parts will be retrieved. This function will block until an item is available or until it timesout.

Note Call(s) to vRingbufferReturnItem() is required after this to free up the item(s) retrieved.

Note This function should only be called on allow-split buffers

Return

- pdTRUE if an item (split or unsplit) was retrieved
- pdFALSE when no item was retrieved

Parameters

- `xRingbuffer`: Ring buffer to retrieve the item from
- `ppvHeadItem`: Double pointer to first part (set to NULL if no items were retrieved)
- `ppvTailItem`: Double pointer to second part (set to NULL if item is not split)
- `pxHeadItemSize`: Pointer to size of first part (unmodified if no items were retrieved)
- `pxTailItemSize`: Pointer to size of second part (unmodified if item is not split)
- `xTicksToWait`: Ticks to wait for items in the ring buffer.

BaseType_t **`xRingbufferReceiveSplitFromISR`**(*RingbufHandle_t xRingbuffer*, void **ppvHeadItem, void **ppvTailItem, size_t *pxHeadItemSize, size_t *pxTailItemSize)

Retrieve a split item from an allow-split ring buffer in an ISR.

Attempt to retrieve a split item from an allow-split ring buffer. If the item is not split, only a single item is retried. If the item is split, both parts will be retrieved. This function returns immediately if there are no items available for retrieval

Note Calls to vRingbufferReturnItemFromISR() is required after this to free up the item(s) retrieved.

Note This function should only be called on allow-split buffers

Return

- pdTRUE if an item (split or unsplit) was retrieved
- pdFALSE when no item was retrieved

Parameters

- `xRingbuffer`: Ring buffer to retrieve the item from

- `ppvHeadItem`: Double pointer to first part (set to NULL if no items were retrieved)
- `ppvTailItem`: Double pointer to second part (set to NULL if item is not split)
- `pxHeadItemSize`: Pointer to size of first part (unmodified if no items were retrieved)
- `pxTailItemSize`: Pointer to size of second part (unmodified if item is not split)

void ***xRingbufferReceiveUpTo**(*RingbufHandle_t xRingbuffer*, size_t **pxItemSize*, TickType_t *xTicksToWait*, size_t *xMaxSize*)

Retrieve bytes from a byte buffer, specifying the maximum amount of bytes to retrieve.

Attempt to retrieve data from a byte buffer whilst specifying a maximum number of bytes to retrieve. This function will block until there is data available for retrieval or until it timesout.

Note A call to vRingbufferReturnItem() is required after this to free up the data retrieved.

Note This function should only be called on byte buffers

Note Byte buffers do not allow multiple retrievals before returning an item

Return

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL on timeout, *pxItemSize is untouched in that case.

Parameters

- `xRingbuffer`: Ring buffer to retrieve the item from
- `pxItemSize`: Pointer to a variable to which the size of the retrieved item will be written.
- `xTicksToWait`: Ticks to wait for items in the ring buffer.
- `xMaxSize`: Maximum number of bytes to return.

void ***xRingbufferReceiveUpToFromISR**(*RingbufHandle_t xRingbuffer*, size_t **pxItemSize*, size_t *xMaxSize*)

Retrieve bytes from a byte buffer, specifying the maximum amount of bytes to retrieve. Call this from an ISR.

Attempt to retrieve bytes from a byte buffer whilst specifying a maximum number of bytes to retrieve. This function will return immediately if there is no data available for retrieval.

Note A call to vRingbufferReturnItemFromISR() is required after this to free up the data received.

Note This function should only be called on byte buffers

Note Byte buffers do not allow multiple retrievals before returning an item

Return

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL when the ring buffer is empty, *pxItemSize is untouched in that case.

Parameters

- `xRingbuffer`: Ring buffer to retrieve the item from
- `pxItemSize`: Pointer to a variable to which the size of the retrieved item will be written.
- `xMaxSize`: Maximum number of bytes to return.

void **vRingbufferReturnItem**(*RingbufHandle_t xRingbuffer*, void **pvItem*)

Return a previously-retrieved item to the ring buffer.

Note If a split item is retrieved, both parts should be returned by calling this function twice

Parameters

- `xRingbuffer`: Ring buffer the item was retrieved from
- `pvItem`: Item that was received earlier

void **vRingbufferReturnItemFromISR** (*RingbufHandle_t xRingbuffer*, void **pvItem*, BaseType_t **pxHigherPriorityTaskWoken*)

Return a previously-retrieved item to the ring buffer from an ISR.

Note If a split item is retrieved, both parts should be returned by calling this function twice

Parameters

- `xRingbuffer`: Ring buffer the item was retrieved from
- `pvItem`: Item that was received earlier
- `pxHigherPriorityTaskWoken`: Value pointed to will be set to pdTRUE if the function woke up a higher priority task.

void **vRingbufferDelete** (*RingbufHandle_t xRingbuffer*)

Delete a ring buffer.

Parameters

- `xRingbuffer`: Ring buffer to delete

size_t **xRingbufferGetMaxItemSize** (*RingbufHandle_t xRingbuffer*)

Get maximum size of an item that can be placed in the ring buffer.

This function returns the maximum size an item can have if it was placed in an empty ring buffer.

Return Maximum size, in bytes, of an item that can be placed in a ring buffer.

Parameters

- `xRingbuffer`: Ring buffer to query

size_t **xRingbufferGetCurFreeSize** (*RingbufHandle_t xRingbuffer*)

Get current free size available for an item/data in the buffer.

This gives the real time free space available for an item/data in the ring buffer. This represents the maximum size an item/data can have if it was currently sent to the ring buffer.

Warning This API is not thread safe. So, if multiple threads are accessing the same ring buffer, it is the application's responsibility to ensure atomic access to this API and the subsequent Send

Return Current free size, in bytes, available for an entry

Parameters

- `xRingbuffer`: Ring buffer to query

BaseType_t **xRingbufferAddToQueueSetRead** (*RingbufHandle_t xRingbuffer*, *QueueSetHandle_t xQueueSet*)

Add the ring buffer's read semaphore to a queue set.

The ring buffer's read semaphore indicates that data has been written to the ring buffer. This function adds the ring buffer's read semaphore to a queue set.

Return

- pdTRUE on success, pdFALSE otherwise

Parameters

- `xRingbuffer`: Ring buffer to add to the queue set
- `xQueueSet`: Queue set to add the ring buffer's read semaphore to

BaseType_t **`xRingbufferCanRead`**(*RingbufHandle_t xRingbuffer*, *QueueSetMemberHandle_t xMember*)

Check if the selected queue set member is the ring buffer's read semaphore.

This API checks if queue set member returned from xQueueSelectFromSet() is the read semaphore of this ring buffer. If so, this indicates the ring buffer has items waiting to be retrieved.

Return

- pdTRUE when semaphore belongs to ring buffer
- pdFALSE otherwise.

Parameters

- `xRingbuffer`: Ring buffer which should be checked
- `xMember`: Member returned from xQueueSelectFromSet

BaseType_t **`xRingbufferRemoveFromQueueSetRead`**(*RingbufHandle_t xRingbuffer*, *QueueSetHandle_t xQueueSet*)

Remove the ring buffer's read semaphore from a queue set.

This specifically removes a ring buffer's read semaphore from a queue set. The read semaphore is used to indicate when data has been written to the ring buffer

Return

- pdTRUE on success
- pdFALSE otherwise

Parameters

- `xRingbuffer`: Ring buffer to remove from the queue set
- `xQueueSet`: Queue set to remove the ring buffer's read semaphore from

void **`vRingbufferGetInfo`**(*RingbufHandle_t xRingbuffer*, UBaseType_t **uxFree*, UBaseType_t **uxRead*, UBaseType_t **uxWrite*, UBaseType_t **uxItemsWaiting*)

Get information about ring buffer status.

Get information of the a ring buffer's current status such as free/read/write pointer positions, and number of items waiting to be retrieved. Arguments can be set to NULL if they are not required.

Parameters

- `xRingbuffer`: Ring buffer to remove from the queue set
- `uxFree`: Pointer use to store free pointer position
- `uxRead`: Pointer use to store read pointer position
- `uxWrite`: Pointer use to store write pointer position

- `uxItemsWaiting`: Pointer use to store number of items (bytes for byte buffer) waiting to be retrieved

void **xRingbufferPrintInfo** (*RingbufHandle_t xRingbuffer*)

Debugging function to print the internal pointers in the ring buffer.

Parameters

- `xRingbuffer`: Ring buffer to show

Type Definitions

typedef void ***RingbufHandle_t**

Type by which ring buffers are referenced. For example, a call to xRingbufferCreate() returns a RingbufHandle_t variable that can then be used as a parameter to xRingbufferSend(), xRingbufferReceive(), etc.

Enumerations

enum ringbuf_type_t

Values:

RINGBUF_TYPE_NOSPLIT = 0

No-split buffers will only store an item in contiguous memory and will never split an item. Each item requires an 8 byte overhead for a header and will always internally occupy a 32-bit aligned size of space.

RINGBUF_TYPE_ALLOWSPLIT

Allow-split buffers will split an item into two parts if necessary in order to store it. Each item requires an 8 byte overhead for a header, splitting incurs an extra header. Each item will always internally occupy a 32-bit aligned size of space.

RINGBUF_TYPE_BYTEBUF

Byte buffers store data as a sequence of bytes and do not maintain separate items, therefore byte buffers have no overhead. All data is stored as a sequence of byte and any number of bytes can be sent or retrieved each time.

Hooks

FreeRTOS consists of Idle Hooks and Tick Hooks which allow for application specific functionality to be added to the Idle Task and Tick Interrupt. ESP-IDF provides its own Idle and Tick Hook API in addition to the hooks provided by Vanilla FreeRTOS. ESP-IDF hooks have the added benefit of being run time configurable and asymmetrical.

Vanilla FreeRTOS Hooks

Idle and Tick Hooks in vanilla FreeRTOS are implemented by the user defining the functions `vApplicationIdleHook()` and `vApplicationTickHook()` respectively somewhere in the application. Vanilla FreeRTOS will run the user defined Idle Hook and Tick Hook on every iteration of the Idle Task and Tick Interrupt respectively.

Vanilla FreeRTOS hooks are referred to as **Legacy Hooks** in ESP-IDF FreeRTOS. To enable legacy hooks, *CONFIG_FREERTOS_LEGACY_HOOKS*, *CONFIG_FREERTOS_LEGACY_IDLE_HOOK*, and *CONFIG_FREERTOS_LEGACY_TICK_HOOK* should all be enabled in `make menuconfig`.

Due to vanilla FreeRTOS being designed for single core, `vApplicationIdleHook()` and `vApplicationTickHook()` can only be defined once. However, the ESP32 is dual core in nature, therefore same Idle Hook and Tick Hook are used for both cores (in other words, the hooks are symmetrical for both cores).

ESP-IDF Idle and Tick Hooks

Due to the the dual core nature of the ESP32, it may be necessary for some applications to have separate hooks for each core. Furthermore, it may be necessary for the Idle Tasks or Tick Interrupts to execute multiple hooks that are configurable at run time. Therefore the ESP-IDF provides it's own hooks API in addition to the legacy hooks provided by Vanilla FreeRTOS.

The ESP-IDF tick/idle hooks are registered at run time, and each tick/idle hook must be registered to a specific CPU. When the idle task runs/tick Interrupt occurs on a particular CPU, the CPU will run each of its registered idle/tick hooks in turn.

Hooks API Reference

Header File

- 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__)()

3.8.3 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_calloc**(size_t *n*, size_t *size*, uint32_t *caps*)

Allocate a chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

Equivalent semantics to libc calloc(), for capability-aware memory.

In IDF, `calloc(p)` is equivalent to `heap_caps_calloc(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 paramters

void ***`heap_caps_realloc_prefer`**(void **ptr*, size_t *size*, size_t *num*, ...)

Allocate a chunk of memory as preference in decreasing order.

Return Pointer to a new buffer of size 'size', or NULL if allocation failed.

Parameters

- `ptr`: Pointer to previously allocated memory, or NULL for a new allocation.
- `size`: Size of the new buffer requested, or 0 to free the buffer.
- `num`: Number of variable paramters

void ***`heap_caps_calloc_prefer`**(size_t *n*, size_t *size*, size_t *num*, ...)

Allocate a chunk of memory as preference in decreasing order.

Return A pointer to the memory allocated on success, NULL on failure

Parameters

- `n`: Number of continuing chunks of memory to allocate

- `size`: Size, in bytes, of a chunk of memory to allocate
- `num`: Number of variable paramters

void **heap_caps_dump**(uint32_t *caps*)

Dump the full structure of all heaps with matching capabilities.

Prints a large amount of output to serial (because of locking limitations, the output bypasses stdout/stderr). For each (variable sized) block in each matching heap, the following output is printed on a single line:

- Block address (the data buffer returned by malloc is 4 bytes after this if heap debugging is set to Basic, or 8 bytes otherwise).
- Data size (the data size may be larger than the size requested by malloc, either due to heap fragmentation or because of heap debugging level).
- Address of next block in the heap.
- If the block is free, the address of the next free block is also printed.

Parameters

- `caps`: Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

void **heap_caps_dump_all**()

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 overriden 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.

3.8.4 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 *`CONFIG_HEAP_CORRUPTION_DETECTION`* can be set to one of three levels:

Basic (no poisoning)

This is the default level. No special heap corruption features are enabled, but provided assertions are enabled (the default configuration) then a heap corruption error will be printed if any of the heap's internal data structures appear overwritten or corrupted. This usually indicates a buffer overrun or out of bounds write.

If assertions are enabled, an assertion will also trigger if a double-free occurs (the same memory is freed twice).

Calling *`heap_caps_check_integrity()`* in Basic mode will check the integrity of all heap structures, and print errors if any appear to be corrupted.

Light Impact

At this level, heap memory is additionally "poisoned" with head and tail "canary bytes" before and after each block which is allocated. If an application writes outside the bounds of allocated buffers, the canary bytes will be corrupted and the integrity check will fail.

The head canary word is 0xABBA1234 (3412BAAB in byte order), and the tail canary word is 0xBAAD5678 (7856ADBA in byte order).

"Basic" heap corruption checks can also detect most out of bounds writes, but this setting is more precise as even a single byte overrun can be detected. With Basic heap checks, the number of overrun bytes before a failure is detected will depend on the properties of the heap.

Enabling "Light Impact" checking increases memory usage, each individual allocation will use 9 to 12 additional bytes of memory (depending on alignment).

Each time `free()` is called in Light Impact mode, the head and tail canary bytes of the buffer being freed are checked against the expected values.

When *`heap_caps_check_integrity()`* is called, all allocated blocks of heap memory have their canary bytes checked against the expected values.

In both cases, the check is that the first 4 bytes of an allocated block (before the buffer returned to the user) should be the word 0xABBA1234. Then the last 4 bytes of the allocated block (after the buffer returned to the user) should be the word 0xBAAD5678.

Different values usually indicate buffer underrun or overrun, respectively.

Comprehensive

This level incorporates the "light impact" detection features plus additional checks for uninitialised-access and use-after-free bugs. In this mode, all freshly allocated memory is filled with the pattern 0xCE, and all freed memory is filled with the pattern 0xFE.

Enabling "Comprehensive" detection has a substantial runtime performance impact (as all memory needs to be set to the allocation patterns each time a malloc/free completes, and the memory also needs to be checked each time.) However it allows easier detection of memory corruption bugs which are much more subtle to find otherwise. It is recommended to only enable this mode when debugging, not in production.

Crashes in Comprehensive Mode

If an application crashes reading/writing an address related to 0xCECECECE in Comprehensive mode, this indicates it has read uninitialized memory. The application should be changed to either use calloc() (which zeroes memory), or initialize the memory before using it. The value 0xCECECECE may also be seen in stack-allocated automatic variables, because in IDF most task stacks are originally allocated from the heap and in C stack memory is uninitialized by default.

If an application crashes and the exception register dump indicates that some addresses or values were 0xFEFEFEFE, this indicates it is reading heap memory after it has been freed (a "use after free bug".) The application should be changed to not access heap memory after it has been freed.

If a call to malloc() or realloc() causes a crash because it expected to find the pattern 0xFEFEFEFE in free memory and a different pattern was found, then this indicates the app has a use-after-free bug where it is writing to memory which has already been freed.

Manual Heap Checks in Comprehensive Mode

Calls to *heap_caps_check_integrity()* may print errors relating to 0xFEFEFEFE, 0xABBA1234 or 0xBAAD5678. In each case the checker is expecting to find a given pattern, and will error out if this is not found:

- For free heap blocks, the checker expects to find all bytes set to 0xFE. Any other values indicate a use-after-free bug where free memory has been incorrectly overwritten.
- For allocated heap blocks, the behaviour is the same as for *Light Impact* mode. The canary bytes 0xABBA1234 and 0xBAAD5678 are checked at the head and tail of each allocated buffer, and any variation indicates a buffer overrun/underrun.

Heap Tracing

Heap Tracing allows tracing of code which allocates/frees memory.

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 *CONFIG_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) );

    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 mode. 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 occured, 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 ***alloced_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

3.8.5 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 sourse* 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. If the current core is not the core that registered this interrupt, this routine will be assigned to the core that allocated this interrupt, blocking and waiting until the resource is successfully released.

Note When the handler shares its source with other handlers, the interrupt status bits it's responsible for should be managed properly before freeing it. see `esp_intr_disable` for more details. Please do not call this function in `esp_ipc_call_blocking`.

Return ESP_ERR_INVALID_ARG the handle is NULL ESP_FAIL failed to release this handle ESP_OK otherwise

Parameters

- `handle`: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

int **esp_intr_get_cpu**(*intr_handle_t* *handle*)

Get CPU number an interrupt is tied to.

Return The core number where the interrupt is allocated

Parameters

- `handle`: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

int **esp_intr_get_intno**(*intr_handle_t handle*)
Get the allocated interrupt for a certain handle.

Return The interrupt number

Parameters

- `handle`: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

esp_err_t **esp_intr_disable**(*intr_handle_t handle*)
Disable the interrupt associated with the handle.

Note

1. For local interrupts (ESP_INTERNAL_* sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.
2. When several handlers sharing a same interrupt source, interrupt status bits, which are handled in the handler to be disabled, should be masked before the disabling, or handled in other enabled interrupts properly. Miss of interrupt status handling will cause infinite interrupt calls and finally system crash.

Return ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

Parameters

- `handle`: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

esp_err_t **esp_intr_enable**(*intr_handle_t handle*)
Enable the interrupt associated with the handle.

Note For local interrupts (ESP_INTERNAL_* sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.

Return ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

Parameters

- `handle`: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

esp_err_t **esp_intr_set_in_iram**(*intr_handle_t handle*, bool *is_in_iram*)
Set the "in IRAM" status of the handler.

Note Does not work on shared interrupts.

Return ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

Parameters

- `handle`: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus
- `is_in_iram`: Whether the handler associated with this handle resides in IRAM. Handlers residing in IRAM can be called when cache is disabled.

void **esp_intr_noniram_disable**()
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**

3.8.6 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 form the TWDT, the TWDT can be deinitialized by calling `esp_task_wdt_deinit()`.

By default `CONFIG_TASK_WDT` in `make menuconfig` will be enabled causing the TWDT to be initialized automatically during startup. Likewise `CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU0` and `CONFIG_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 reenable them upon leaving the breakpoint. This means that interrupt watchdog and task watchdog functionality will essentially be disabled. No warnings or panics from either watchdogs will be generated when the ESP32 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 non-CPU-specific parts of interrupt watchdog. This is called in the init code if the interrupt watchdog is enabled in menuconfig.

Task Watchdog API Reference

A full example using the Task Watchdog is available in esp-idf: system/task_watchdog

Header File

- 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

3.8.7 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 `CONFIG_IPC_TASK_STACK_SIZE` in *menuconfig*. The IPC API is protected by a mutex hence simultaneous IPC calls are not possible.

Care should 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

3.8.8 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

Application Example

The following example illustrates usage of esp_timer APIs: system/esp_timer.

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).

int64_t **esp_timer_get_next_alarm**()

Get the timestamp when the next timeout is expected to occur.

Return Timestamp of the nearest timer event, in microseconds. The timebase is the same as for the values returned by esp_timer_get_time.

esp_err_t **esp_timer_dump**(FILE **stream*)

Dump the list of timers to a stream.

If CONFIG_ESP_TIMER_PROFILING option is enabled, this prints the list of all the existing timers. Otherwise, only the list active timers is printed.

The format is:

name period alarm times_armed times_triggered total_callback_run_time

where:

name — timer name (if CONFIG_ESP_TIMER_PROFILING is defined), or timer pointer period — period of timer, in microseconds, or 0 for one-shot timer alarm - time of the next alarm, in microseconds since boot, or 0 if the timer is not started

The following fields are printed if CONFIG_ESP_TIMER_PROFILING is defined:

times_armed — number of times the timer was armed via esp_timer_start_X times_triggered - number of times the callback was called total_callback_run_time - total time taken by callback to execute, across all calls

Return

- ESP_OK on success
- ESP_ERR_NO_MEM if can not allocate temporary buffer for the output

Parameters

- `stream`: stream (such as stdout) to dump the information to

Structures

struct esp_timer_create_args_t

Timer configuration passed to esp_timer_create.

Public Members

esp_timer_cb_t **callback**

Function to call when timer expires.

void ***arg**

Argument to pass to the callback.

esp_timer_dispatch_t **dispatch_method**

Call the callback from task or from ISR.

const char ***name**

Timer name, used in esp_timer_dump function.

Type Definitions

typedef struct esp_timer ***esp_timer_handle_t**

Opaque type representing a single esp_timer.

typedef void (***esp_timer_cb_t**) (void *arg)

Timer callback function type.

Parameters

- `arg`: pointer to opaque user-specific data

Enumerations

enum esp_timer_dispatch_t

Method for dispatching timer callback.

Values:

ESP_TIMER_TASK

Callback is called from timer task.

3.8.9 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.

The log levels are Error, Warning, Info, Debug, and Verbose (from lowest to highest level of verbosity).

At compile time, filtering is done using *CONFIG_LOG_DEFAULT_LEVEL* option, 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 reduce logging level per module. Modules are identified by their tags, which are human-readable ASCII zero-terminated strings.

Note that *esp_log_level_set()* can not increase logging level beyound that set by *CONFIG_LOG_DEFAULT_LEVEL*. To increase log level for a specific file at compile time, *LOG_LOCAL_LEVEL* macro can be used (see below for details).

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 (lowest)
- `ESP_LOGW` - warning
- `ESP_LOGI` - info
- `ESP_LOGD` - debug
- `ESP_LOGV` - verbose (highest)

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.

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. By calling a simple API, all log output may be routed to JTAG instead, making logging several times faster. For details please refer to section *Logging to Host*.

Application Example

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.

Note that this function can not raise log level above the level set using CONFIG_LOG_DEFAULT_LEVEL setting in menuconfig.

To raise log level above the default one for a given file, define LOG_LOCAL_LEVEL to one of the ESP_LOG_* values, before including esp_log.h in this file.

Parameters

- `tag`: Tag of the log entries to enable. Must be a non-NULL zero terminated string. Value "*" resets log level for all tags to the given value.
- `level`: Selects log level to enable. Only logs at this and lower verbosity levels will be shown.

vprintf_like_t **esp_log_set_vprintf** (*vprintf_like_t func*)

Set function used to output log entries.

By default, log output goes to UART0. This function can be used to redirect log output to some other destination, such as file or network. Returns the original log handler, which may be necessary to return output to the previous destination.

Return func old Function used for output.

Parameters

- `func`: new Function used for output. Must have same signature as vprintf.

uint32_t **esp_log_timestamp** (void)

Function which returns timestamp to be used in log output.

This function is used in expansion of ESP_LOGx macros. In the 2nd stage bootloader, and at early application startup stage this function uses CPU cycle counter as time source. Later when FreeRTOS scheduler start running, it switches to FreeRTOS tick count.

For now, we ignore millisecond counter overflow.

Return timestamp, in milliseconds

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_EARLY_LOGE (tag, format, ...)
macro to output logs in startup code, before heap allocator and syscalls have been initialized. log at `ESP_LOG_ERROR` level.

See `printf`,`ESP_LOGE`

ESP_EARLY_LOGW (tag, format, ...)
macro to output logs in startup code at `ESP_LOG_WARN` level.

See `ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

ESP_EARLY_LOGI (tag, format, ...)
macro to output logs in startup code at `ESP_LOG_INFO` level.

See `ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

ESP_EARLY_LOGD (tag, format, ...)
macro to output logs in startup code at `ESP_LOG_DEBUG` level.

See `ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

ESP_EARLY_LOGV (tag, format, ...)
macro to output logs in startup code at `ESP_LOG_VERBOSE` level.

See `ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

ESP_LOG_EARLY_IMPL (tag, format, log_level, log_tag_letter, ...)

ESP_LOGE (tag, format, ...)

ESP_LOGW (tag, format, ...)

ESP_LOGI (tag, format, ...)

ESP_LOGD (tag, format, ...)

ESP_LOGV (tag, format, ...)

ESP_LOG_LEVEL (level, tag, format, ...)
runtime macro to output logs at a specified 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 specified 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.

3.8.10 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_apptrace_init**()

Initializes application tracing module.

Note Should be called before any esp_apptrace_xxx call.

Return ESP_OK on success, otherwise see esp_err_t

void **esp_apptrace_down_buffer_config**(uint8_t **buf*, uint32_t *size*)

Configures down buffer.

Note Needs to be called before initiating any data transfer using esp_apptrace_buffer_get and esp_apptrace_write. This function does not protect internal data by lock.

Parameters

- `buf`: Address of buffer to use for down channel (host to target) data.
- `size`: Size of the buffer.

uint8_t ***esp_apptrace_buffer_get**(*esp_apptrace_dest_t* *dest*, uint32_t *size*, uint32_t *tmo*)

Allocates buffer for trace data. After data in buffer are ready to be sent off esp_apptrace_buffer_put must be called to indicate it.

Return non-NULL on success, otherwise NULL.

Parameters

- `dest`: Indicates HW interface to send data.
- `size`: Size of data to write to trace buffer.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

esp_err_t **esp_apptrace_buffer_put**(*esp_apptrace_dest_t* *dest*, uint8_t **ptr*, uint32_t *tmo*)

Indicates that the data in buffer are ready to be sent off. This function is a counterpart of and must be preceeded by esp_apptrace_buffer_get.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- `dest`: Indicates HW interface to send data. Should be identical to the same parameter in call to esp_apptrace_buffer_get.
- `ptr`: Address of trace buffer to release. Should be the value returned by call to esp_apptrace_buffer_get.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

esp_err_t **esp_apptrace_write**(*esp_apptrace_dest_t* *dest*, **const** void **data*, uint32_t *size*, uint32_t *tmo*)

Writes data to trace buffer.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- `dest`: Indicates HW interface to send data.

- `data`: Address of data to write to trace buffer.
- `size`: Size of data to write to trace buffer.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

int **esp_apptrace_vprintf_to** (*esp_apptrace_dest_t dest*, uint32_t *tmo*, **const** char **fmt*, va_list *ap*)
vprintf-like function to sent log messages to host via specified HW interface.

Return Number of bytes written.

Parameters

- `dest`: Indicates HW interface to send data.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.
- `fmt`: Address of format string.
- `ap`: List of arguments.

int **esp_apptrace_vprintf** (**const** char **fmt*, va_list *ap*)
vprintf-like function to sent log messages to host.

Return Number of bytes written.

Parameters

- `fmt`: Address of format string.
- `ap`: List of arguments.

esp_err_t **esp_apptrace_flush** (*esp_apptrace_dest_t dest*, uint32_t *tmo*)
Flushes remaining data in trace buffer to host.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- `dest`: Indicates HW interface to flush data on.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

esp_err_t **esp_apptrace_flush_nolock** (*esp_apptrace_dest_t dest*, uint32_t *min_sz*, uint32_t *tmo*)
Flushes remaining data in trace buffer to host without locking internal data. This is special version of esp_apptrace_flush which should be called from panic handler.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- `dest`: Indicates HW interface to flush data on.
- `min_sz`: Threshold for flushing data. If current filling level is above this value, data will be flushed. TRAX destinations only.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

esp_err_t **esp_apptrace_read** (*esp_apptrace_dest_t dest*, void **data*, uint32_t **size*, uint32_t *tmo*)
Reads host data from trace buffer.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- `dest`: Indicates HW interface to read the data on.
- `data`: Address of buffer to put data from trace buffer.
- `size`: Pointer to store size of read data. Before call to this function pointed memory must hold requested size of data
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

uint8_t ***esp_apptrace_down_buffer_get*** (*esp_apptrace_dest_t dest*, uint32_t **size*, uint32_t *tmo*)

Rertrieves incoming data buffer if any. After data in buffer are processed esp_apptrace_down_buffer_put must be called to indicate it.

Return non-NULL on success, otherwise NULL.

Parameters

- `dest`: Indicates HW interface to receive data.
- `size`: Address to store size of available data in down buffer. Must be initializaed with requested value.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

esp_err_t **esp_apptrace_down_buffer_put** (*esp_apptrace_dest_t dest*, uint8_t **ptr*, uint32_t *tmo*)

Indicates that the data in down buffer are processesd. This function is a counterpart of and must be preceeded by esp_apptrace_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_apptrace_down_buffer_get.
- `ptr`: Address of trace buffer to release. Should be the value returned by call to esp_apptrace_down_buffer_get.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinetly.

bool **esp_apptrace_host_is_connected** (*esp_apptrace_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_apptrace_fopen*** (*esp_apptrace_dest_t dest*, **const** char **path*, **const** char **mode*)

Opens file on host. This function has the same semantic as 'fopen' except for the first argument.

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_apptrace_fclose**(*esp_apptrace_dest_t dest*, void **stream*)

Closes file on host. This function has the same semantic as 'fclose' except for the first argument.

Return Zero on success, otherwise non-zero. See fclose for details.

Parameters

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by esp_apptrace_fopen.

size_t **esp_apptrace_fwrite**(*esp_apptrace_dest_t dest*, **const** void **ptr*, size_t *size*, size_t *nmemb*, void **stream*)

Writes to file on host. This function has the same semantic as 'fwrite' except for the first argument.

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_apptrace_fopen.

size_t **esp_apptrace_fread**(*esp_apptrace_dest_t dest*, void **ptr*, size_t *size*, size_t *nmemb*, void **stream*)

Read file on host. This function has the same semantic as 'fread' except for the first argument.

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_apptrace_fopen.

int **esp_apptrace_fseek**(*esp_apptrace_dest_t dest*, void **stream*, long *offset*, int *whence*)

Set position indicator in file on host. This function has the same semantic as 'fseek' except for the first argument.

Return Zero on success, otherwise non-zero. See fseek for details.

Parameters

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by esp_apptrace_fopen.
- `offset`: Offset. See fseek for details.
- `whence`: Position in file. See fseek for details.

int **esp_apptrace_ftell** (*esp_apptrace_dest_t dest*, void **stream*)
Get current position indicator for file on host. This function has the same semantic as 'ftell' except for the first argument.

Return Current position in file. See ftell for details.

Parameters

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by esp_apptrace_fopen.

int **esp_apptrace_fstop** (*esp_apptrace_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_apptrace_dest_t
Application trace data destinations bits.

Values:

ESP_APPTRACE_DEST_TRAX = 0x1
JTAG destination.

ESP_APPTRACE_DEST_UART0 = 0x2
UART destination.

3.8.11 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 `CONFIG_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) and automatic light sleep 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, `CONFIG_PM_DFS_INIT_AUTO` option can be enabled in menuconfig. If enabled, maximal CPU frequency is determined by `CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ` setting, and minimal CPU frequency is set to the XTAL frequency.

Note: Automatic light sleep is based on FreeRTOS Tickless Idle functionality. `esp_pm_configure()` will return an *ESP_ERR_NOT_SUPPORTED* error if `CONFIG_FREERTOS_USE_TICKLESS_IDLE` option is not enabled in menuconfig, but automatic light sleep is requested.

Note: In light sleep, peripherals are clock gated, and interrupts (from GPIOs and internal peripherals) will not be generated. Wakeup source described in *Sleep Modes* documentation can be used to wake from light sleep state. For example, EXT0 and EXT1 wakeup source can be used to wake up from a GPIO.

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, the lock must be released the same number of times as it was acquired, in order for power managment 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.

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 `CONFIG_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).
- When none of the locks are aquired, and light sleep is enabled in a call to `esp_pm_configure()`, the system will go into light sleep mode. The duration of light sleep will be determined by:
  - FreeRTOS tasks blocked with finite timeouts
  - Timers registered with *High resolution timer* APIs

  Light sleep will duration will be chosen to wake up before the nearest event (task being unblocked, or timer elapses).

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 thte 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

3.8.12 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 and can be disabled using `esp_sleep_disable_wakeup_source()` API. Next section describes these APIs in detail. Wakeup sources can be configured at any moment before entering light or deep sleep mode.

Additionally, the application can force specific powerdown modes for the RTC peripherals and RTC memories using `esp_sleep_pd_config()` API.

Once wakeup sources are configured, application can enter sleep mode using `esp_light_sleep_start()` or `esp_deep_sleep_start()` APIs. At this point the hardware will be configured according to the requested wakeup sources, and RTC controller will either power down or power off the CPUs and digital peripherals.

WiFi/BT and sleep modes

In deep sleep and light sleep modes, wireless peripherals are powered down. Before entering deep sleep or light sleep modes, applications must disable WiFi and BT using appropriate calls (`esp_bluedroid_disable()`, `esp_bt_controller_disable()`, `esp_wifi_stop()`). WiFi and BT connections will not be maintained in deep sleep or light sleep, even if these functions are not called.

If WiFi connection needs to be maintained, enable WiFi modem sleep, and enable automatic light sleep feature (see *Power Management APIs*). This will allow the system to wake up from sleep automatically when required by WiFi driver, thereby maintaining connection to the AP.

Wakeup sources

Timer

RTC controller has a built in timer which can be used to wake up the chip after a predefined amount of time. Time is specified at microsecond precision, but the actual resolution depends on the clock source selected for RTC SLOW_CLK. See chapter "Reset and Clock" of the ESP32 Technical Reference Manual for details about RTC clock options.

This wakeup mode doesn't require RTC peripherals or RTC memories to be powered on during sleep.

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.

Note The FSM mode of the touch button should be configured as the timer trigger mode.

Return

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if additional current by touch (CONFIG_ESP32_RTC_EXTERNAL_CRYSTAL_ADDITIONAL_CURRENT) is enabled.
- 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);
gpio_pulldown_en(gpio_num);
```

Warning: After wake up from sleep, IO pad(s) used for wakeup will be configured as RTC IO. Before using these pads as digital GPIOs, reconfigure them using `rtc_gpio_deinit(gpio_num)` function.

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 are have RTC functionality can be used in this bit map: 0,2,4,12-15,25-27,32-39.
- `mode`: select logic function used to determine wakeup condition:
  - ESP_EXT1_WAKEUP_ALL_LOW: wake up when all selected GPIOs are low
  - ESP_EXT1_WAKEUP_ANY_HIGH: wake up when any of the selected GPIOs is high

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 memeory. 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_NOT_SUPPORTED if additional current by touch (CONFIG_ESP32_RTC_EXTERNAL_CRYSTAL_ADDITIONAL_CURRENT) is enabled.
- 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 overriden, 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 overriden 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 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_XTAL
XTAL oscillator.

ESP_PD_DOMAIN_MAX
Number of domains.

enum esp_sleep_pd_option_t
Power down options.

Values:

ESP_PD_OPTION_OFF
Power down the power domain in sleep mode.

ESP_PD_OPTION_ON
Keep power domain enabled during sleep mode.

ESP_PD_OPTION_AUTO
Keep power domain enabled in sleep mode, if it is needed by one of the wakeup options. Otherwise power it down.

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 indefinetly, 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 indefinetly, until external reset is applied.

void **esp_deep_sleep_start**()
Enter deep sleep with the configured wakeup options.

This function does not return.

Configuring IOs

Some ESP32 IOs have internal pullups or pulldowns, which are enabled by default. If an external circuit drives this pin in deep sleep mode, current consumption may increase due to current flowing through these pullups and pulldowns.

To isolate a pin, preventing extra current draw, call `rtc_gpio_isolate()` function.

For example, on ESP32-WROVER module, GPIO12 is pulled up externally. GPIO12 also has an internal pull-down in the ESP32 chip. This means that in deep sleep, some current will flow through these external and internal resistors, increasing deep sleep current above the minimal possible value. Add the following code before `esp_deep_sleep_start()` to remove this extra current:

`c++ rtc_gpio_isolate(GPIO_NUM_12); `

UART output handling

Before entering sleep mode, *esp_deep_sleep_start()* will flush the contents of UART FIFOs.

When entering light sleep mode using *esp_light_sleep_start()*, UART FIFOs will not be flushed. Instead, UART output will be suspended, and remaining characters in the FIFO will be sent out after wakeup from light sleep.

Checking sleep wakeup cause

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

typedef *esp_sleep_source_t* **esp_sleep_wakeup_cause_t**

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 GPIOn caused wakeup, BIT(n) will be set

Disable sleep wakeup source

Previously configured wakeup source can be disabled later using *esp_sleep_disable_wakeup_source()* API. This function deactivates trigger for source defined as input parameter if it should not be used to wake up from sleep.

enum esp_sleep_source_t

Sleep wakeup cause.

Values:

ESP_SLEEP_WAKEUP_UNDEFINED

In case of deep sleep, reset was not caused by exit from deep sleep.

ESP_SLEEP_WAKEUP_EXT0

Wakeup caused by external signal using RTC_IO.

ESP_SLEEP_WAKEUP_EXT1

Wakeup caused by external signal using RTC_CNTL.

ESP_SLEEP_WAKEUP_TIMER
Wakeup caused by timer.

ESP_SLEEP_WAKEUP_TOUCHPAD
Wakeup caused by touchpad.

ESP_SLEEP_WAKEUP_ULP
Wakeup caused by ULP program.

esp_err_t **esp_sleep_disable_wakeup_source**(*esp_sleep_source_t source*)
Disable wakeup source.

This function is used to deactivate wake up trigger for source defined as parameter of the function.

See docs/sleep-modes.rst for details.

Note This function does not modify wake up configuration in RTC. It will be performed in esp_sleep_start function.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if trigger was not active

Parameters

- `source`: - number of source to disable of type esp_sleep_source_t

Application Example

Implementation of basic functionality of deep sleep is shown in protocols/sntp example, where ESP module is periodically waken up to retrive time from NTP server.

More extensive example in system/deep_sleep illustrates usage of various deep sleep wakeup triggers and ULP coprocessor programming.

3.8.13 Base MAC address

Overview

Serveral 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.

3.8.14 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.

Secure OTA Updates Without Secure boot

The verification of signed OTA updates can be performed even without enabling hardware secure boot. For doing so, refer *Signed App Verification Without Hardware Secure Boot*

See also

- *Partition Table documentation*
- *Lower-Level SPI Flash/Partition API*
- *ESP HTTPS OTA*

Application Example

End-to-end example of OTA firmware update workflow: system/ota.

API Reference

Header File

- app_update/include/esp_ota_ops.h

Functions

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().

3.8.15 ESP HTTPS OTA

Overview

`esp_https_ota` provides simplified APIs to perform firmware upgrades over HTTPS. It's an abstraction layer over existing OTA APIs.

Application Example

```
esp_err_t do_firmware_upgrade()
{
    esp_http_client_config_t config = {
        .url = CONFIG_FIRMWARE_UPGRADE_URL,
        .cert_pem = (char *)server_cert_pem_start,
    };
    esp_err_t ret = esp_https_ota(&config);
    if (ret == ESP_OK) {
        esp_restart();
    } else {
        return ESP_FAIL;
    }
    return ESP_OK;
}
```

Signature Verification

For additional security, signature of OTA firmware images can be verified. For that, refer *Secure OTA Updates Without Secure boot*

API Reference

Header File

- esp_https_ota/include/esp_https_ota.h

Functions

esp_err_t **esp_https_ota**(**const** *esp_http_client_config_t* **config*)

HTTPS OTA Firmware upgrade.

This function performs HTTPS OTA Firmware upgrade

Note For secure HTTPS updates, the `cert_pem` member of `config` structure must be set to the server certificate.

Return

- ESP_OK: OTA data updated, next reboot will use specified partition.
- ESP_FAIL: For generic failure.
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image
- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- For other return codes, refer OTA documentation in esp-idf's app_update component.

Parameters

- `config`: pointer to *esp_http_client_config_t* structure.

3.8.16 ESP-pthread

Overview

This module offers Espressif specific extensions to the pthread library that can be used to influence the behaviour of pthreads. C

- Stack size of the pthreads
- Priority of the created pthreads
- Inheriting this configuration across threads

Example to tune the stack size of the pthread:

```
main()
{
    pthread_t t1;

    esp_pthread_cfg_t cfg;
    cfg.stack_size = (4 * 1024);
    esp_pthread_set_cfg(&cfg);

    pthread_create(&t1, NULL, thread_func);
}
```

The API can also be used for inheriting the settings across threads. For example:

```
void * my_thread2(void * p)
{
      /* This thread will inherit the stack size of 4K */
      printf("In my_thread2\n");
}

void * my_thread1(void * p)
{
      printf("In my_thread1\n");
      pthread_t t2;
      pthread_create(&t2, NULL, my_thread2);
}

main()
{

     pthread_t t1;

     esp_pthread_cfg_t cfg;
     cfg.stack_size = (4 * 1024);
     cfg.inherit_cfg = true;
     esp_pthread_set_cfg(&cfg);

     pthread_create(&t1, NULL, my_thread1);
}
```

API Reference

Header File

- pthread/include/esp_pthread.h

Functions

esp_err_t **esp_pthread_set_cfg**(**const** *esp_pthread_cfg_t* **cfg*)

Configure parameters for creating pthread.

This API allows you to configure how the subsequent pthread_create() call will behave. This call can be used to setup configuration parameters like stack size, priority, configuration inheritance etc.

If the 'inherit' flag in the configuration structure is enabled, then the same configuration is also inherited in the thread subtree.

Return

- ESP_OK if configuration was successfully set
- ESP_ERR_NO_MEM if out of memory

Parameters

- `cfg`: The pthread config parameters

esp_err_t **esp_pthread_get_cfg**(*esp_pthread_cfg_t* **p*)

Get current pthread creation configuration.

This will retrieve the current configuration that will be used for creating threads.

Return

- ESP_OK if the configuration was available
- ESP_ERR_NOT_FOUND if a configuration wasn't previously set

Parameters

- `p`: Pointer to the pthread config structure that will be updated with the currently configured parameters

Structures

struct esp_pthread_cfg_t

pthread configuration structure that influences pthread creation

Public Members

size_t **stack_size**

the stack size of the pthread

size_t **prio**

the thread's priority

bool **inherit_cfg**

inherit this configuration further

3.8.17 Error Codes and Helper Functions

This section lists definitions of common ESP-IDF error codes and several helper functions related to error handling.

For general information about error codes in ESP-IDF, see *Error Handling*.

For the full list of error codes defined in ESP-IDF, see *Error Code Reference*.

API Reference

Header File

- esp32/include/esp_err.h

Functions

const char ***esp_err_to_name** (*esp_err_t code*)

Returns string for esp_err_t error codes.

This function finds the error code in a pre-generated lookup-table and returns its string representation.

The function is generated by the Python script tools/gen_esp_err_to_name.py which should be run each time an esp_err_t error is modified, created or removed from the IDF project.

Return string error message

Parameters

- `code`: esp_err_t error code

const char ***esp_err_to_name_r** (*esp_err_t code*, char **buf*, size_t *buflen*)

Returns string for esp_err_t and system error codes.

This function finds the error code in a pre-generated lookup-table of esp_err_t errors and returns its string representation. If the error code is not found then it is attempted to be found among system errors.

The function is generated by the Python script tools/gen_esp_err_to_name.py which should be run each time an esp_err_t error is modified, created or removed from the IDF project.

Return buf containing the string error message

Parameters

- `code`: esp_err_t error code
- `buf`: buffer where the error message should be written
- `buflen`: Size of buffer buf. At most buflen bytes are written into the buf buffer (including the terminating null byte).

Macros

ESP_OK

esp_err_t value indicating success (no error)

ESP_FAIL

Generic esp_err_t code indicating failure

ESP_ERR_NO_MEM

Out of memory

ESP_ERR_INVALID_ARG

Invalid argument

ESP_ERR_INVALID_STATE

Invalid state

ESP_ERR_INVALID_SIZE

Invalid size

ESP_ERR_NOT_FOUND

Requested resource not found

ESP_ERR_NOT_SUPPORTED

Operation or feature not supported

ESP_ERR_TIMEOUT
Operation timed out

ESP_ERR_INVALID_RESPONSE
Received response was invalid

ESP_ERR_INVALID_CRC
CRC or checksum was invalid

ESP_ERR_INVALID_VERSION
Version was invalid

ESP_ERR_INVALID_MAC
MAC address was invalid

ESP_ERR_WIFI_BASE
Starting number of WiFi error codes

ESP_ERR_MESH_BASE
Starting number of MESH error codes

ESP_ERROR_CHECK(x)
Macro which can be used to check the error code, and terminate the program in case the code is not ESP_OK. Prints the error code, error location, and the failed statement to serial output.

Disabled if assertions are disabled.

Type Definitions

typedef int32_t **esp_err_t**

Example code for this API section is provided in system directory of ESP-IDF examples.

3.9 Configuration Options

3.9.1 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.

3.9.2 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 supress 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 overriden by the ones in `sdkconfig.defaults`. For more information, see *Custom sdkconfig defaults*.

3.9.3 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.

SDK tool configuration

- *CONFIG_TOOLPREFIX*
- *CONFIG_PYTHON*
- *CONFIG_MAKE_WARN_UNDEFINED_VARIABLES*

CONFIG_TOOLPREFIX

Compiler toolchain path/prefix

The prefix/path that is used to call the toolchain. The default setting assumes a crosstool-ng gcc setup that is in your PATH.

Found in SDK tool configuration

CONFIG_PYTHON

Python 2 interpreter

The executable name/path that is used to run python. On some systems Python 2.x may need to be invoked as python2.

(Note: This option is used with the GNU Make build system only, not idf.py or CMake-based builds.)

Found in SDK tool configuration

CONFIG_MAKE_WARN_UNDEFINED_VARIABLES

'make' warns on undefined variables

Adds –warn-undefined-variables to MAKEFLAGS. This causes make to print a warning any time an undefined variable is referenced.

This option helps find places where a variable reference is misspelled or otherwise missing, but it can be unwanted if you have Makefiles which depend on undefined variables expanding to an empty string.

Found in SDK tool configuration

Serial flasher config

- *CONFIG_ESPTOOLPY_PORT*
- *CONFIG_ESPTOOLPY_BAUD*
- *CONFIG_ESPTOOLPY_BAUD_OTHER_VAL*
- *CONFIG_ESPTOOLPY_BAUD*
- *CONFIG_ESPTOOLPY_COMPRESSED*
- *CONFIG_FLASHMODE*
- CONFIG_ESPTOOLPY_FLASHMODE
- *CONFIG_ESPTOOLPY_FLASHFREQ*
- *CONFIG_ESPTOOLPY_FLASHFREQ*
- *CONFIG_ESPTOOLPY_FLASHSIZE*
- *CONFIG_ESPTOOLPY_FLASHSIZE*
- *CONFIG_ESPTOOLPY_FLASHSIZE_DETECT*
- *CONFIG_ESPTOOLPY_BEFORE*
- *CONFIG_ESPTOOLPY_BEFORE*
- *CONFIG_ESPTOOLPY_AFTER*
- *CONFIG_ESPTOOLPY_AFTER*
- *CONFIG_MONITOR_BAUD*
- *CONFIG_MONITOR_BAUD_OTHER_VAL*
- *CONFIG_MONITOR_BAUD*

CONFIG_ESPTOOLPY_PORT

Default serial port

The serial port that's connected to the ESP chip. This can be overridden by setting the ESPPORT environment variable.

This value is ignored when using the CMake-based build system or idf.py.

Found in Serial flasher config

CONFIG_ESPTOOLPY_BAUD

Default baud rate

Default baud rate to use while communicating with the ESP chip. Can be overridden by setting the ESPBAUD variable.

This value is ignored when using the CMake-based build system or idf.py.

Found in Serial flasher config

Available options:

- 115200 baud (*CONFIG_ESPTOOLPY_BAUD_115200B*)
- 230400 baud (*CONFIG_ESPTOOLPY_BAUD_230400B*)
- 921600 baud (*CONFIG_ESPTOOLPY_BAUD_921600B*)
- 2Mbaud (*CONFIG_ESPTOOLPY_BAUD_2MB*)

- Other baud rate (*CONFIG_ESPTOOLPY_BAUD_OTHER*)

CONFIG_ESPTOOLPY_BAUD_OTHER_VAL
Other baud rate value

Found in Serial flasher config

CONFIG_ESPTOOLPY_COMPRESSED
Use compressed upload

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.

Found in Serial flasher config

CONFIG_FLASHMODE
Flash SPI mode

Mode the flash chip is flashed in, as well as the default mode for the binary to run in.

Found in Serial flasher config

Available options:

- QIO (*CONFIG_FLASHMODE_QIO*)
- QOUT (*CONFIG_FLASHMODE_QOUT*)
- DIO (*CONFIG_FLASHMODE_DIO*)
- DOUT (*CONFIG_FLASHMODE_DOUT*)

CONFIG_ESPTOOLPY_FLASHFREQ
Flash SPI speed

The SPI flash frequency to be used.

Found in Serial flasher config

Available options:

- 80 MHz (*CONFIG_ESPTOOLPY_FLASHFREQ_80M*)
- 40 MHz (*CONFIG_ESPTOOLPY_FLASHFREQ_40M*)
- 26 MHz (*CONFIG_ESPTOOLPY_FLASHFREQ_26M*)
- 20 MHz (*CONFIG_ESPTOOLPY_FLASHFREQ_20M*)

CONFIG_ESPTOOLPY_FLASHSIZE
Flash size

SPI flash size, in megabytes

Found in Serial flasher config

Available options:

- 1 MB (*CONFIG_ESPTOOLPY_FLASHSIZE_1MB*)
- 2 MB (*CONFIG_ESPTOOLPY_FLASHSIZE_2MB*)
- 4 MB (*CONFIG_ESPTOOLPY_FLASHSIZE_4MB*)
- 8 MB (*CONFIG_ESPTOOLPY_FLASHSIZE_8MB*)
- 16 MB (*CONFIG_ESPTOOLPY_FLASHSIZE_16MB*)

CONFIG_ESPTOOLPY_FLASHSIZE_DETECT
Detect flash size when flashing bootloader

If this option is set, 'make flash' targets will automatically detect the flash size and update the bootloader image when flashing.

Found in Serial flasher config

CONFIG_ESPTOOLPY_BEFORE
Before flashing

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.

Found in Serial flasher config

Available options:

- Reset to bootloader (*CONFIG_ESPTOOLPY_BEFORE_RESET*)
- No reset (*CONFIG_ESPTOOLPY_BEFORE_NORESET*)

CONFIG_ESPTOOLPY_AFTER
After flashing

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.

Found in Serial flasher config

Available options:

- Reset after flashing (*CONFIG_ESPTOOLPY_AFTER_RESET*)
- Stay in bootloader (*CONFIG_ESPTOOLPY_AFTER_NORESET*)

CONFIG_MONITOR_BAUD
'make monitor' baud rate

Baud rate to use when running 'make monitor' to view serial output from a running chip.

Can override by setting the MONITORBAUD environment variable.

Found in Serial flasher config

Available options:

- 9600 bps (*CONFIG_MONITOR_BAUD_9600B*)
- 57600 bps (*CONFIG_MONITOR_BAUD_57600B*)
- 115200 bps (*CONFIG_MONITOR_BAUD_115200B*)
- 230400 bps (*CONFIG_MONITOR_BAUD_230400B*)
- 921600 bps (*CONFIG_MONITOR_BAUD_921600B*)
- 2 Mbps (*CONFIG_MONITOR_BAUD_2MB*)
- Custom baud rate (*CONFIG_MONITOR_BAUD_OTHER*)

CONFIG_MONITOR_BAUD_OTHER_VAL
Custom baud rate value

Found in Serial flasher config

Bootloader config

- *CONFIG_LOG_BOOTLOADER_LEVEL*
- *CONFIG_LOG_BOOTLOADER_LEVEL*
- *CONFIG_BOOTLOADER_SPI_WP_PIN*
- *CONFIG_BOOTLOADER_VDDSDIO_BOOST*
- *CONFIG_BOOTLOADER_FACTORY_RESET*
- *CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET*
- *CONFIG_BOOTLOADER_OTA_DATA_ERASE*
- *CONFIG_BOOTLOADER_DATA_FACTORY_RESET*
- *CONFIG_BOOTLOADER_APP_TEST*
- *CONFIG_BOOTLOADER_NUM_PIN_APP_TEST*
- *CONFIG_BOOTLOADER_HOLD_TIME_GPIO*

CONFIG_LOG_BOOTLOADER_LEVEL

Bootloader log verbosity

Specify how much output to see in bootloader logs.

Found in Bootloader config

Available options:

- No output (*CONFIG_LOG_BOOTLOADER_LEVEL_NONE*)
- Error (*CONFIG_LOG_BOOTLOADER_LEVEL_ERROR*)
- Warning (*CONFIG_LOG_BOOTLOADER_LEVEL_WARN*)
- Info (*CONFIG_LOG_BOOTLOADER_LEVEL_INFO*)
- Debug (*CONFIG_LOG_BOOTLOADER_LEVEL_DEBUG*)
- Verbose (*CONFIG_LOG_BOOTLOADER_LEVEL_VERBOSE*)

CONFIG_BOOTLOADER_SPI_WP_PIN

SPI Flash WP Pin when customising pins via efuse (read help)

This value is ignored unless flash mode is set to QIO or QOUT *and* the SPI flash pins have been overriden 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.

Found in Bootloader config

CONFIG_BOOTLOADER_VDDSDIO_BOOST

VDDSDIO LDO voltage

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.

Found in Bootloader config

Available options:

- 1.8V (*CONFIG_BOOTLOADER_VDDSDIO_BOOST_1_8V*)
- 1.9V (*CONFIG_BOOTLOADER_VDDSDIO_BOOST_1_9V*)

CONFIG_BOOTLOADER_FACTORY_RESET
GPIO triggers factory reset

Allows to reset the device to factory settings: - clear one or more data partitions; - boot from "factory" partition. The factory reset will occur if there is a GPIO input pulled low while device starts up. See settings below.

Found in Bootloader config

CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET
Number of the GPIO input for factory reset

The selected GPIO will be configured as an input with internal pull-up enabled. To trigger a factory reset, this GPIO must be pulled low on reset. Note that GPIO34-39 do not have an internal pullup and an external one must be provided.

Found in Bootloader config

CONFIG_BOOTLOADER_OTA_DATA_ERASE
Clear OTA data on factory reset (select factory partition)

The device will boot from "factory" partition (or OTA slot 0 if no factory partition is present) after a factory reset.

Found in Bootloader config

CONFIG_BOOTLOADER_DATA_FACTORY_RESET
Comma-separated names of partitions to clear on factory reset

Allows customers to select which data partitions will be erased while factory reset.

Specify the names of partitions as a comma-delimited with optional spaces for readability. (Like this: "nvs, phy_init, . . . ") Make sure that the name specified in the partition table and here are the same. Partitions of type "app" cannot be specified here.

Found in Bootloader config

CONFIG_BOOTLOADER_APP_TEST
GPIO triggers boot from test app partition

Allows to run the test app from "TEST" partition. A boot from "test" partition will occur if there is a GPIO input pulled low while device starts up. See settings below.

Found in Bootloader config

CONFIG_BOOTLOADER_NUM_PIN_APP_TEST
Number of the GPIO input to boot TEST partition

The selected GPIO will be configured as an input with internal pull-up enabled. To trigger a test app, this GPIO must be pulled low on reset. After the GPIO input is deactivated and the device reboots, the old application will boot. (factory or OTA[x]). Note that GPIO34-39 do not have an internal pullup and an external one must be provided.

Found in Bootloader config

CONFIG_BOOTLOADER_HOLD_TIME_GPIO
Hold time of GPIO for reset/test mode (seconds)

The GPIO must be held low continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.

Found in Bootloader config

Security features

- CONFIG_SECURE_SIGNED_ON_BOOT
- CONFIG_SECURE_SIGNED_ON_UPDATE
- CONFIG_SECURE_SIGNED_APPS
- *CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT*
- *CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT*
- *CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT*
- *CONFIG_SECURE_BOOT_ENABLED*
- *CONFIG_SECURE_BOOTLOADER_MODE*
- *CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES*
- *CONFIG_SECURE_BOOT_SIGNING_KEY*
- *CONFIG_SECURE_BOOT_VERIFICATION_KEY*
- *CONFIG_SECURE_BOOTLOADER_KEY_ENCODING*
- *CONFIG_SECURE_BOOT_INSECURE*
- *CONFIG_FLASH_ENCRYPTION_ENABLED*
- *CONFIG_FLASH_ENCRYPTION_INSECURE*
- *CONFIG_SECURE_BOOT_ALLOW_ROM_BASIC*
- *CONFIG_SECURE_BOOT_ALLOW_JTAG*
- *CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION*
- *CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_ENCRYPT*
- *CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_DECRYPT*
- *CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_CACHE*
- *CONFIG_SECURE_BOOT_TEST_MODE*
- *CONFIG_FLASH_ENCRYPTION_DISABLE_PLAINTEXT*

CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT

Require signed app images

Require apps to be signed to verify their integrity.

This option uses the same app signature scheme as hardware secure boot, but unlike hardware secure boot it does not prevent the bootloader from being physically updated. This means that the device can be secured against remote network access, but not physical access. Compared to using hardware Secure Boot this option is much simpler to implement.

Found in Security features

CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT
Bootloader verifies app signatures

If this option is set, the bootloader will be compiled with code to verify that an app is signed before booting it.

If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option doesn't add significant security by itself so most users will want to leave it disabled.

Found in Security features

CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT
Verify app signature on update

If this option is set, any OTA updated apps will have the signature verified before being considered valid.

When enabled, the signature is automatically checked whenever the esp_ota_ops.h APIs are used for OTA updates, or esp_image_format.h APIs are used to verify apps.

If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option still adds significant security against network-based attackers by preventing spoofing of OTA updates.

Found in Security features

CONFIG_SECURE_BOOT_ENABLED
Enable hardware secure boot in bootloader (READ DOCS FIRST)

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://docs.espressif.com/projects/esp-idf/en/latest/security/secure-boot.html before enabling.

Found in Security features

CONFIG_SECURE_BOOTLOADER_MODE
Secure bootloader mode

Found in Security features

Available options:

- One-time flash (*CONFIG_SECURE_BOOTLOADER_ONE_TIME_FLASH*)

  On first boot, the bootloader will generate a key which is not readable externally or by software. A digest is generated from the bootloader image itself. This digest will be verified on each subsequent boot.

  Enabling this option means that the bootloader cannot be changed after the first time it is booted.

- Reflashable (*CONFIG_SECURE_BOOTLOADER_REFLASHABLE*)

  Generate a reusable secure bootloader key, derived (via SHA-256) from the secure boot signing key.

  This allows the secure bootloader to be re-flashed by anyone with access to the secure boot signing key.

  This option is less secure than one-time flash, because a leak of the digest key from one device allows reflashing of any device that uses it.

CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES
Sign binaries during build

Once secure boot or signed app requirement is enabled, app images are required to be signed.

If enabled (default), these binary files are signed as part of the build process. The file named in "Secure boot private signing key" will be used to sign the image.

If disabled, unsigned app/partition data will be built. They must be signed manually using espsecure.py (for example, on a remote signing server.)

Found in Security features

CONFIG_SECURE_BOOT_SIGNING_KEY
Secure boot private signing key

Path to the key file used to sign app images.

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.

Found in Security features

CONFIG_SECURE_BOOT_VERIFICATION_KEY
Secure boot public signature verification key

Path to a public key file used to verify signed images. This key is compiled into the bootloader and/or app, to verify app images.

Key file is in raw binary format, and can be extracted from a PEM formatted private key using the espsecure.py extract_public_key command.

Refer to https://docs.espressif.com/projects/esp-idf/en/latest/security/secure-boot.html before enabling.

Found in Security features

CONFIG_SECURE_BOOTLOADER_KEY_ENCODING
Hardware Key Encoding

In reflashable secure bootloader mode, a hardware key is derived from the signing key (with SHA-256) and can be written to efuse with espefuse.py.

Normally this is a 256-bit key, but if 3/4 Coding Scheme is used on the device then the efuse key is truncated to 192 bits.

This configuration item doesn't change any firmware code, it only changes the size of key binary which is generated at build time.

Found in Security features

Available options:

- No encoding (256 bit key) (*CONFIG_SECURE_BOOTLOADER_KEY_ENCODING_256BIT*)
- 3/4 encoding (192 bit key) (*CONFIG_SECURE_BOOTLOADER_KEY_ENCODING_192BIT*)

CONFIG_SECURE_BOOT_INSECURE
Allow potentially insecure options

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://docs.espressif.com/projects/esp-idf/en/latest/security/secure-boot.html before enabling.

Found in Security features

CONFIG_FLASH_ENCRYPTION_ENABLED

Enable flash encryption on boot (READ DOCS FIRST)

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://docs.espressif.com/projects/esp-idf/en/latest/security/flash-encryption.html before enabling.

Found in Security features

CONFIG_FLASH_ENCRYPTION_INSECURE

Allow potentially insecure options

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.

Found in Security features

Potentially insecure options

CONFIG_SECURE_BOOT_ALLOW_ROM_BASIC

Leave ROM BASIC Interpreter available on reset

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.

Found in Security features > Potentially insecure options

CONFIG_SECURE_BOOT_ALLOW_JTAG

Allow JTAG Debugging

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.

Found in Security features > Potentially insecure options

CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION

Allow app partition length not 64KB aligned

If not set (default), app partition size must be a multiple of 64KB. App images are padded to 64KB length, and the bootloader checks any trailing bytes after the signature (before the next 64KB boundary) have not been

written. This is because flash cache maps entire 64KB pages into the address space. This prevents an attacker from appending unverified data after the app image in the flash, causing it to be mapped into the address space.

Setting this option allows the app partition length to be unaligned, and disables padding of the app image to this length. It is generally not recommended to set this option, unless you have a legacy partitioning scheme which doesn't support 64KB aligned partition lengths.

Found in Security features > Potentially insecure options

CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_ENCRYPT

Leave UART bootloader encryption enabled

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.

Found in Security features > Potentially insecure options

CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_DECRYPT

Leave UART bootloader decryption enabled

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.

Found in Security features > Potentially insecure options

CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_CACHE

Leave UART bootloader flash cache enabled

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.

Found in Security features > Potentially insecure options

CONFIG_SECURE_BOOT_TEST_MODE

Secure boot test mode: don't permanently set any efuses

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.

Found in Security features > Potentially insecure options

CONFIG_FLASH_ENCRYPTION_DISABLE_PLAINTEXT

Disable serial reflashing of plaintext firmware

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 Reflashable 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.

Found in Security features

Partition Table

- *CONFIG_PARTITION_TABLE_TYPE*
- *CONFIG_PARTITION_TABLE_CUSTOM_FILENAME*
- CONFIG_PARTITION_TABLE_FILENAME
- *CONFIG_PARTITION_TABLE_OFFSET*
- *CONFIG_PARTITION_TABLE_MD5*

CONFIG_PARTITION_TABLE_TYPE

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.

Found in Partition Table

Available options:

- Single factory app, no OTA (*CONFIG_PARTITION_TABLE_SINGLE_APP*)
- Factory app, two OTA definitions (*CONFIG_PARTITION_TABLE_TWO_OTA*)
- Custom partition table CSV (*CONFIG_PARTITION_TABLE_CUSTOM*)

CONFIG_PARTITION_TABLE_CUSTOM_FILENAME

Custom partition CSV file

Name of the custom partition CSV filename. This path is evaluated relative to the project root directory.

Found in Partition Table

CONFIG_PARTITION_TABLE_OFFSET

Offset of partition table

The address of partition table (by default 0x8000). Allows you to move the partition table, it gives more space for the bootloader. Note that the bootloader and app will both need to be compiled with the same PARTITION_TABLE_OFFSET value.

Found in Partition Table

CONFIG_PARTITION_TABLE_MD5

Generate an MD5 checksum for the partition table

Generate an MD5 checksum for the partition table for protecting the integrity of the table. The generation should be turned off for legacy bootloaders which cannot recognize the MD5 checksum in the partition table.

Found in Partition Table

Compiler options

- *CONFIG_OPTIMIZATION_COMPILER*
- *CONFIG_OPTIMIZATION_ASSERTION_LEVEL*
- *CONFIG_CXX_EXCEPTIONS_EMG_POOL_SIZE*
- *CONFIG_STACK_CHECK_MODE*
- CONFIG_STACK_CHECK

- *CONFIG_WARN_WRITE_STRINGS*

CONFIG_OPTIMIZATION_COMPILER

Optimization Level

This option sets compiler optimization level (gcc -O argument).

- for "Release" setting, -Os flag is added to CFLAGS.
- for "Debug" setting, -Og flag is added to CFLAGS.

"Release" with -Os produces smaller & faster compiled code but it may be harder to correlated code addresses to source files when debugging.

To add custom optimization settings, set CFLAGS and/or CPPFLAGS in project makefile, before including $(IDF_PATH)/make/project.mk. Note that custom optimization levels may be unsupported.

Found in Compiler options

Available options:

- Debug (-Og) (*CONFIG_OPTIMIZATION_LEVEL_DEBUG*)
- Release (-Os) (*CONFIG_OPTIMIZATION_LEVEL_RELEASE*)

CONFIG_OPTIMIZATION_ASSERTION_LEVEL

Assertion level

Assertions can be:

- Enabled. Failure will print verbose assertion details. This is the default.
- Set to "silent" to save code size (failed assertions will abort() but user needs to use the aborting address to find the line number with the failed assertion.)
- Disabled entirely (not recommended for most configurations.) -DNDEBUG is added to CPPFLAGS in this case.

Found in Compiler options

Available options:

- Enabled (*CONFIG_OPTIMIZATION_ASSERTIONS_ENABLED*)

  Enable assertions. Assertion content and line number will be printed on failure.

- Silent (saves code size) (*CONFIG_OPTIMIZATION_ASSERTIONS_SILENT*)

  Enable silent assertions. Failed assertions will abort(), user needs to use the aborting address to find the line number with the failed assertion.

- Disabled (sets -DNDEBUG) (*CONFIG_OPTIMIZATION_ASSERTIONS_DISABLED*)

  If assertions are disabled, -DNDEBUG is added to CPPFLAGS.

CONFIG_CXX_EXCEPTIONS

Enable C++ exceptions

Enable C++ exceptions

Enabling this option compiles all IDF C++ files with exception support enabled.

Disabling this option disables C++ exception support in all compiled files, and any libstdc++ code which throws an exception will abort instead.

Enabling this option currently adds an additional ~500 bytes of heap overhead when an exception is thrown in user code for the first time.

CONFIG_CXX_EXCEPTIONS_EMG_POOL_SIZE

Emergency Pool Size

Size (in bytes) of the emergency memory pool for C++ exceptions. This pool will be used to allocate memory for thrown exceptions when there is not enough memory on the heap.

Found in Compiler options > Enable C++ exceptions

CONFIG_STACK_CHECK_MODE

Stack smashing protection mode

Stack smashing protection mode. Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, program is halted. Protection has the following modes:

- In NORMAL mode (GCC flag: -fstack-protector) only functions that call alloca, and functions with buffers larger than 8 bytes are protected.
- STRONG mode (GCC flag: -fstack-protector-strong) is like NORMAL, but includes additional functions to be protected – those that have local array definitions, or have references to local frame addresses.
- In OVERALL mode (GCC flag: -fstack-protector-all) all functions are protected.

Modes have the following impact on code performance and coverage:

- performance: NORMAL > STRONG > OVERALL
- coverage: NORMAL < STRONG < OVERALL

Found in Compiler options

Available options:

- None (*CONFIG_STACK_CHECK_NONE*)
- Normal (*CONFIG_STACK_CHECK_NORM*)
- Strong (*CONFIG_STACK_CHECK_STRONG*)
- Overall (*CONFIG_STACK_CHECK_ALL*)

CONFIG_WARN_WRITE_STRINGS

Enable -Wwrite-strings warning flag

Adds -Wwrite-strings flag for the C/C++ compilers.

For C, this gives string constants the type `const char[]` so that copying the address of one into a non-const `char *` pointer produces a warning. This warning helps to find at compile time code that tries to write into a string constant.

For C++, this warns about the deprecated conversion from string literals to `char *`.

Found in Compiler options

Component config

- *CONFIG_OPENSSL_DEBUG*
- *CONFIG_OPENSSL_DEBUG_LEVEL*

- *CONFIG_OPENSSL_LOWLEVEL_DEBUG*
- *CONFIG_OPENSSL_ASSERT*
- *CONFIG_SUPPRESS_SELECT_DEBUG_OUTPUT*
- *CONFIG_MBEDTLS_SSL_MAX_CONTENT_LEN*
- *CONFIG_MBEDTLS_DEBUG*
- *CONFIG_MBEDTLS_HARDWARE_AES*
- *CONFIG_MBEDTLS_HARDWARE_MPI*
- *CONFIG_MBEDTLS_MPI_USE_INTERRUPT*
- *CONFIG_MBEDTLS_HARDWARE_SHA*
- *CONFIG_MBEDTLS_HAVE_TIME*
- *CONFIG_MBEDTLS_HAVE_TIME_DATE*
- *CONFIG_MBEDTLS_TLS_MODE*
- CONFIG_MBEDTLS_TLS_SERVER
- CONFIG_MBEDTLS_TLS_CLIENT
- CONFIG_MBEDTLS_TLS_ENABLED
- *CONFIG_MBEDTLS_PSK_MODES*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_PSK*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_PSK*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_PSK*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_RSA_PSK*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_RSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_RSA*
- *CONFIG_MBEDTLS_SSL_RENEGOTIATION*
- *CONFIG_MBEDTLS_SSL_PROTO_SSL3*
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1*
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1_1*
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1_2*
- *CONFIG_MBEDTLS_SSL_PROTO_DTLS*
- *CONFIG_MBEDTLS_SSL_ALPN*
- *CONFIG_MBEDTLS_SSL_SESSION_TICKETS*
- *CONFIG_MBEDTLS_AES_C*

- *CONFIG_MBEDTLS_CAMELLIA_C*
- *CONFIG_MBEDTLS_DES_C*
- *CONFIG_MBEDTLS_RC4_MODE*
- *CONFIG_MBEDTLS_BLOWFISH_C*
- *CONFIG_MBEDTLS_XTEA_C*
- *CONFIG_MBEDTLS_CCM_C*
- *CONFIG_MBEDTLS_GCM_C*
- *CONFIG_MBEDTLS_RIPEMD160_C*
- *CONFIG_MBEDTLS_PEM_PARSE_C*
- *CONFIG_MBEDTLS_PEM_WRITE_C*
- *CONFIG_MBEDTLS_X509_CRL_PARSE_C*
- *CONFIG_MBEDTLS_X509_CSR_PARSE_C*
- *CONFIG_MBEDTLS_ECDH_C*
- *CONFIG_MBEDTLS_ECDSA_C*
- *CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED*
- *CONFIG_MBEDTLS_ECP_NIST_OPTIM*
- *CONFIG_HEAP_CORRUPTION_DETECTION*
- *CONFIG_HEAP_TRACING*
- *CONFIG_HEAP_TRACING_STACK_DEPTH*
- *CONFIG_HEAP_TASK_TRACKING*
- *CONFIG_ESP32_PTHREAD_TASK_PRIO_DEFAULT*
- *CONFIG_ESP32_PTHREAD_TASK_STACK_SIZE_DEFAULT*
- *CONFIG_ADC_FORCE_XPD_FSM*
- *CONFIG_ADC2_DISABLE_DAC*
- *CONFIG_SPI_MASTER_IN_IRAM*

- *CONFIG_SPI_MASTER_ISR_IN_IRAM*
- *CONFIG_LOG_DEFAULT_LEVEL*
- *CONFIG_LOG_DEFAULT_LEVEL*
- *CONFIG_LOG_COLORS*
- *CONFIG_AWS_IOT_MQTT_HOST*
- *CONFIG_AWS_IOT_MQTT_PORT*
- *CONFIG_AWS_IOT_MQTT_TX_BUF_LEN*
- *CONFIG_AWS_IOT_MQTT_RX_BUF_LEN*
- *CONFIG_AWS_IOT_MQTT_NUM_SUBSCRIBE_HANDLERS*
- *CONFIG_AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL*
- *CONFIG_AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL*
- *CONFIG_AWS_IOT_OVERRIDE_THING_SHADOW_RX_BUFFER*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_RX_BUFFER*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_UNIQUE_CLIENT_ID_BYTES*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_ACKS*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_THINGNAMES*
- *CONFIG_AWS_IOT_SHADOW_MAX_JSON_TOKEN_EXPECTED*
- *CONFIG_AWS_IOT_SHADOW_MAX_SHADOW_TOPIC_LENGTH_WITHOUT_THINGNAME*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_THING_NAME*
- *CONFIG_MP_BLOB_SUPPORT*
- *CONFIG_NVS_ENCRYPTION*
- *CONFIG_SPIFFS_MAX_PARTITIONS*
- *CONFIG_SPIFFS_CACHE*
- *CONFIG_SPIFFS_CACHE_WR*
- *CONFIG_SPIFFS_CACHE_STATS*
- *CONFIG_SPIFFS_PAGE_CHECK*
- *CONFIG_SPIFFS_GC_MAX_RUNS*
- *CONFIG_SPIFFS_GC_STATS*
- *CONFIG_SPIFFS_PAGE_SIZE*
- *CONFIG_SPIFFS_OBJ_NAME_LEN*
- *CONFIG_SPIFFS_USE_MAGIC*
- *CONFIG_SPIFFS_USE_MAGIC_LENGTH*
- *CONFIG_SPIFFS_META_LENGTH*
- *CONFIG_SPIFFS_USE_MTIME*
- *CONFIG_SPIFFS_DBG*
- *CONFIG_SPIFFS_API_DBG*

- *CONFIG_SPIFFS_GC_DBG*
- *CONFIG_SPIFFS_CACHE_DBG*
- *CONFIG_SPIFFS_CHECK_DBG*
- *CONFIG_SPIFFS_TEST_VISUALISATION*
- *CONFIG_WL_SECTOR_SIZE*
- *CONFIG_WL_SECTOR_SIZE*
- *CONFIG_WL_SECTOR_MODE*
- *CONFIG_WL_SECTOR_MODE*
- *CONFIG_IP_LOST_TIMER_INTERVAL*
- *CONFIG_FATFS_CHOOSE_CODEPAGE*
- CONFIG_FATFS_CODEPAGE
- *CONFIG_FATFS_LONG_FILENAMES*
- *CONFIG_FATFS_MAX_LFN*
- *CONFIG_FATFS_API_ENCODING*
- *CONFIG_FATFS_FS_LOCK*
- *CONFIG_FATFS_TIMEOUT_MS*
- *CONFIG_FATFS_PER_FILE_CACHE*
- *CONFIG_BT_ENABLED*
- *CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH*
- CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH_EFF
- *CONFIG_BTDM_CONTROLLER_AUTO_LATENCY*
- CONFIG_BTDM_CONTROLLER_AUTO_LATENCY_EFF
- *CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE_CHOICE*
- CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE
- *CONFIG_BTDM_CONTROLLER_HCI_MODE_CHOICE*
- *CONFIG_BT_HCI_UART_NO*
- *CONFIG_BT_HCI_UART_BAUDRATE*
- *CONFIG_BTDM_CONTROLLER_MODEM_SLEEP*
- *CONFIG_BTDM_MODEM_SLEEP_MODE*
- *CONFIG_BTDM_LOW_POWER_CLOCK*
- *CONFIG_BLUEDROID_PINNED_TO_CORE_CHOICE*
- CONFIG_BLUEDROID_PINNED_TO_CORE
- *CONFIG_BTC_TASK_STACK_SIZE*
- *CONFIG_BLUEDROID_MEM_DEBUG*
- *CONFIG_CLASSIC_BT_ENABLED*
- *CONFIG_A2DP_ENABLE*

- *CONFIG_A2DP_SINK_TASK_STACK_SIZE*
- *CONFIG_A2DP_SOURCE_TASK_STACK_SIZE*
- *CONFIG_BT_SPP_ENABLED*
- *CONFIG_HFP_ENABLE*
- *CONFIG_HFP_ROLE*
- *CONFIG_HFP_AUDIO_DATA_PATH*
- *CONFIG_GATTS_ENABLE*
- *CONFIG_GATTC_ENABLE*
- *CONFIG_GATTC_CACHE_NVS_FLASH*
- *CONFIG_BLE_SMP_ENABLE*
- *CONFIG_SMP_SLAVE_CON_PARAMS_UPD_ENABLE*
- *CONFIG_BT_STACK_NO_LOG*
- *CONFIG_HCI_INITIAL_TRACE_LEVEL*
- *CONFIG_HCI_INITIAL_TRACE_LEVEL*
- *CONFIG_BTM_INITIAL_TRACE_LEVEL*
- *CONFIG_BTM_INITIAL_TRACE_LEVEL*
- *CONFIG_L2CAP_INITIAL_TRACE_LEVEL*
- *CONFIG_L2CAP_INITIAL_TRACE_LEVEL*
- *CONFIG_RFCOMM_INITIAL_TRACE_LEVEL*
- *CONFIG_RFCOMM_INITIAL_TRACE_LEVEL*
- *CONFIG_SDP_INITIAL_TRACE_LEVEL*
- *CONFIG_SDP_INITIAL_TRACE_LEVEL*
- *CONFIG_GAP_INITIAL_TRACE_LEVEL*
- *CONFIG_GAP_INITIAL_TRACE_LEVEL*
- *CONFIG_BNEP_INITIAL_TRACE_LEVEL*
- *CONFIG_BNEP_INITIAL_TRACE_LEVEL*
- *CONFIG_PAN_INITIAL_TRACE_LEVEL*
- *CONFIG_PAN_INITIAL_TRACE_LEVEL*
- *CONFIG_A2D_INITIAL_TRACE_LEVEL*
- *CONFIG_A2D_INITIAL_TRACE_LEVEL*
- *CONFIG_AVDT_INITIAL_TRACE_LEVEL*
- *CONFIG_AVDT_INITIAL_TRACE_LEVEL*
- *CONFIG_AVCT_INITIAL_TRACE_LEVEL*
- *CONFIG_AVCT_INITIAL_TRACE_LEVEL*
- *CONFIG_AVRC_INITIAL_TRACE_LEVEL*
- *CONFIG_AVRC_INITIAL_TRACE_LEVEL*

- *CONFIG_MCA_INITIAL_TRACE_LEVEL*
- *CONFIG_MCA_INITIAL_TRACE_LEVEL*
- *CONFIG_HID_INITIAL_TRACE_LEVEL*
- *CONFIG_HID_INITIAL_TRACE_LEVEL*
- *CONFIG_APPL_INITIAL_TRACE_LEVEL*
- *CONFIG_APPL_INITIAL_TRACE_LEVEL*
- *CONFIG_GATT_INITIAL_TRACE_LEVEL*
- *CONFIG_GATT_INITIAL_TRACE_LEVEL*
- *CONFIG_SMP_INITIAL_TRACE_LEVEL*
- *CONFIG_SMP_INITIAL_TRACE_LEVEL*
- *CONFIG_BTIF_INITIAL_TRACE_LEVEL*
- *CONFIG_BTIF_INITIAL_TRACE_LEVEL*
- *CONFIG_BTC_INITIAL_TRACE_LEVEL*
- *CONFIG_BTC_INITIAL_TRACE_LEVEL*
- *CONFIG_OSI_INITIAL_TRACE_LEVEL*
- *CONFIG_OSI_INITIAL_TRACE_LEVEL*
- *CONFIG_BLUFI_INITIAL_TRACE_LEVEL*
- *CONFIG_BLUFI_INITIAL_TRACE_LEVEL*
- *CONFIG_BT_ACL_CONNECTIONS*
- *CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST*
- *CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY*
- *CONFIG_BLE_HOST_QUEUE_CONGESTION_CHECK*
- *CONFIG_BLE_SCAN_DUPLICATE*
- *CONFIG_DUPLICATE_SCAN_CACHE_SIZE*
- *CONFIG_BLE_MESH_SCAN_DUPLICATE_EN*
- *CONFIG_MESH_DUPLICATE_SCAN_CACHE_SIZE*
- *CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_SUPPORTED*
- *CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_NUM*
- *CONFIG_BLE_ADV_REPORT_DISCARD_THRSHOLD*
- CONFIG_SMP_ENABLE
- *CONFIG_BLE_ACTIVE_SCAN_REPORT_ADV_SCAN_RSP_INDIVIDUALLY*
- *CONFIG_BLE_ESTABLISH_LINK_CONNECTION_TIMEOUT*
- CONFIG_BT_RESERVE_DRAM
- *CONFIG_LIBSODIUM_USE_MBEDTLS_SHA*
- *CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS*
- *CONFIG_L2_TO_L3_COPY*

- *CONFIG_LWIP_IRAM_OPTIMIZATION*
- *CONFIG_LWIP_MAX_SOCKETS*
- *CONFIG_USE_ONLY_LWIP_SELECT*
- *CONFIG_LWIP_SO_REUSE*
- *CONFIG_LWIP_SO_REUSE_RXTOALL*
- *CONFIG_LWIP_SO_RCVBUF*
- *CONFIG_LWIP_DHCP_MAX_NTP_SERVERS*
- *CONFIG_LWIP_IP_FRAG*
- *CONFIG_LWIP_IP_REASSEMBLY*
- *CONFIG_LWIP_STATS*
- *CONFIG_LWIP_ETHARP_TRUST_IP_MAC*
- *CONFIG_ESP_GRATUITOUS_ARP*
- *CONFIG_GARP_TMR_INTERVAL*
- *CONFIG_TCPIP_RECVMBOX_SIZE*
- *CONFIG_LWIP_DHCP_DOES_ARP_CHECK*
- *CONFIG_LWIP_DHCPS_LEASE_UNIT*
- *CONFIG_LWIP_DHCPS_MAX_STATION_NUM*
- *CONFIG_LWIP_AUTOIP_TRIES*
- *CONFIG_LWIP_AUTOIP_MAX_CONFLICTS*
- *CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL*
- *CONFIG_LWIP_IPV6_AUTOCONFIG*
- *CONFIG_LWIP_LOOPBACK_MAX_PBUFS*
- *CONFIG_LWIP_MAX_ACTIVE_TCP*
- *CONFIG_LWIP_MAX_LISTENING_TCP*
- *CONFIG_TCP_MAXRTX*
- *CONFIG_TCP_SYNMAXRTX*
- *CONFIG_TCP_MSS*
- *CONFIG_TCP_MSL*
- *CONFIG_TCP_SND_BUF_DEFAULT*
- *CONFIG_TCP_WND_DEFAULT*
- *CONFIG_TCP_RECVMBOX_SIZE*
- *CONFIG_TCP_QUEUE_OOSEQ*
- *CONFIG_ESP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES*
- *CONFIG_TCP_OVERSIZE*
- *CONFIG_LWIP_WND_SCALE*
- *CONFIG_TCP_RCV_SCALE*

- *CONFIG_LWIP_MAX_UDP_PCBS*
- *CONFIG_UDP_RECVMBOX_SIZE*
- *CONFIG_TCPIP_TASK_STACK_SIZE*
- *CONFIG_PPP_PAP_SUPPORT*
- *CONFIG_PPP_CHAP_SUPPORT*
- *CONFIG_PPP_MSCHAP_SUPPORT*
- *CONFIG_PPP_MPPE_SUPPORT*
- *CONFIG_PPP_DEBUG_ON*
- *CONFIG_LWIP_MULTICAST_PING*
- *CONFIG_LWIP_BROADCAST_PING*
- *CONFIG_LWIP_MAX_RAW_PCBS*
- *CONFIG_ESP32_APPTRACE_DESTINATION*
- CONFIG_ESP32_APPTRACE_ENABLE
- CONFIG_ESP32_APPTRACE_LOCK_ENABLE
- *CONFIG_ESP32_APPTRACE_ONPANIC_HOST_FLUSH_TMO*
- *CONFIG_ESP32_APPTRACE_POSTMORTEM_FLUSH_TRAX_THRESH*
- *CONFIG_ESP32_APPTRACE_PENDING_DATA_SIZE_MAX*
- *CONFIG_SYSVIEW_ENABLE*
- *CONFIG_SYSVIEW_TS_SOURCE*
- *CONFIG_SYSVIEW_EVT_OVERFLOW_ENABLE*
- *CONFIG_SYSVIEW_EVT_ISR_ENTER_ENABLE*
- *CONFIG_SYSVIEW_EVT_ISR_EXIT_ENABLE*
- *CONFIG_SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_START_EXEC_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_START_READY_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_STOP_READY_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_CREATE_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_TERMINATE_ENABLE*
- *CONFIG_SYSVIEW_EVT_IDLE_ENABLE*
- *CONFIG_SYSVIEW_EVT_TIMER_ENTER_ENABLE*
- *CONFIG_SYSVIEW_EVT_TIMER_EXIT_ENABLE*
- *CONFIG_ESP32_GCOV_ENABLE*
- *CONFIG_ESP32_REV_MIN*
- *CONFIG_ESP32_REV_MIN*
- CONFIG_ESP32_DPORT_WORKAROUND

- *CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ*
- *CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ*
- *CONFIG_SPIRAM_SUPPORT*
- *CONFIG_SPIRAM_BOOT_INIT*
- *CONFIG_SPIRAM_IGNORE_NOTFOUND*
- *CONFIG_SPIRAM_USE*
- *CONFIG_SPIRAM_TYPE*
- CONFIG_SPIRAM_SIZE
- *CONFIG_SPIRAM_SPEED*
- *CONFIG_SPIRAM_MEMTEST*
- *CONFIG_SPIRAM_CACHE_WORKAROUND*
- *CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL*
- *CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST*
- *CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL*
- *CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY*
- CONFIG_MEMMAP_TRACEMEM
- CONFIG_MEMMAP_TRACEMEM_TWOBANKS
- *CONFIG_ESP32_TRAX*
- *CONFIG_ESP32_TRAX_TWOBANKS*
- CONFIG_TRACEMEM_RESERVE_DRAM
- *CONFIG_ESP32_COREDUMP_TO_FLASH_OR_UART*
- CONFIG_ESP32_ENABLE_COREDUMP
- *CONFIG_ESP32_CORE_DUMP_UART_DELAY*
- *CONFIG_ESP32_CORE_DUMP_LOG_LEVEL*
- *CONFIG_NUMBER_OF_UNIVERSAL_MAC_ADDRESS*
- *CONFIG_NUMBER_OF_UNIVERSAL_MAC_ADDRESS*
- *CONFIG_SYSTEM_EVENT_QUEUE_SIZE*
- *CONFIG_SYSTEM_EVENT_TASK_STACK_SIZE*
- *CONFIG_MAIN_TASK_STACK_SIZE*
- *CONFIG_IPC_TASK_STACK_SIZE*
- *CONFIG_TIMER_TASK_STACK_SIZE*
- *CONFIG_NEWLIB_STDOUT_LINE_ENDING*
- *CONFIG_NEWLIB_STDIN_LINE_ENDING*
- *CONFIG_NEWLIB_NANO_FORMAT*
- *CONFIG_CONSOLE_UART*
- *CONFIG_CONSOLE_UART_NUM*

- *CONFIG_CONSOLE_UART_NUM*
- *CONFIG_CONSOLE_UART_TX_GPIO*
- *CONFIG_CONSOLE_UART_RX_GPIO*
- *CONFIG_CONSOLE_UART_BAUDRATE*
- *CONFIG_ULP_COPROC_ENABLED*
- *CONFIG_ULP_COPROC_RESERVE_MEM*
- *CONFIG_ESP32_PANIC*
- *CONFIG_ESP32_DEBUG_OCDAWARE*
- *CONFIG_ESP32_DEBUG_STUBS_ENABLE*
- *CONFIG_INT_WDT*
- *CONFIG_INT_WDT_TIMEOUT_MS*
- *CONFIG_INT_WDT_CHECK_CPU1*
- *CONFIG_TASK_WDT*
- *CONFIG_TASK_WDT_PANIC*
- *CONFIG_TASK_WDT_TIMEOUT_S*
- *CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU0*
- *CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU1*
- *CONFIG_BROWNOUT_DET*
- *CONFIG_BROWNOUT_DET_LVL_SEL*
- CONFIG_BROWNOUT_DET_LVL
- *CONFIG_ESP32_TIME_SYSCALL*
- *CONFIG_ESP32_RTC_CLOCK_SOURCE*
- *CONFIG_ESP32_RTC_EXTERNAL_CRYSTAL_ADDITIONAL_CURRENT*
- *CONFIG_ESP32_RTC_CLK_CAL_CYCLES*
- *CONFIG_ESP32_RTC_XTAL_BOOTSTRAP_CYCLES*
- *CONFIG_ESP32_DEEP_SLEEP_WAKEUP_DELAY*
- *CONFIG_ESP32_XTAL_FREQ_SEL*
- CONFIG_ESP32_XTAL_FREQ
- *CONFIG_DISABLE_BASIC_ROM_CONSOLE*
- *CONFIG_NO_BLOBS*
- *CONFIG_ESP_TIMER_PROFILING*
- *CONFIG_COMPATIBLE_PRE_V2_1_BOOTLOADERS*
- *CONFIG_ESP_ERR_TO_NAME_LOOKUP*
- *CONFIG_SW_COEXIST_ENABLE*
- *CONFIG_SW_COEXIST_PREFERENCE*
- CONFIG_SW_COEXIST_PREFERENCE_VALUE

- *CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_TX_BUFFER*
- CONFIG_ESP32_WIFI_TX_BUFFER_TYPE
- *CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_CSI_ENABLED*
- *CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED*
- *CONFIG_ESP32_WIFI_TX_BA_WIN*
- *CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED*
- *CONFIG_ESP32_WIFI_RX_BA_WIN*
- *CONFIG_ESP32_WIFI_NVS_ENABLED*
- *CONFIG_ESP32_WIFI_TASK_CORE_ID*
- *CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN*
- *CONFIG_ESP32_WIFI_IRAM_OPT*
- *CONFIG_ESP32_WIFI_RX_IRAM_OPT*
- *CONFIG_ESP32_WIFI_MGMT_SBUF_NUM*
- *CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE*
- *CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION*
- *CONFIG_ESP32_PHY_MAX_WIFI_TX_POWER*
- CONFIG_ESP32_PHY_MAX_TX_POWER
- *CONFIG_ESP32_DPORT_DIS_INTERRUPT_LVL*
- *CONFIG_PM_ENABLE*
- *CONFIG_PM_DFS_INIT_AUTO*
- *CONFIG_PM_USE_RTC_TIMER_REF*
- *CONFIG_PM_PROFILING*
- *CONFIG_PM_TRACE*
- *CONFIG_FREERTOS_UNICORE*
- *CONFIG_FREERTOS_CORETIMER*
- *CONFIG_FREERTOS_HZ*
- *CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION*
- *CONFIG_FREERTOS_CHECK_STACKOVERFLOW*
- *CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK*
- *CONFIG_FREERTOS_INTERRUPT_BACKTRACE*
- *CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS*
- *CONFIG_FREERTOS_ASSERT*

- *CONFIG_FREERTOS_IDLE_TASK_STACKSIZE*
- *CONFIG_FREERTOS_ISR_STACKSIZE*
- *CONFIG_FREERTOS_LEGACY_HOOKS*
- *CONFIG_FREERTOS_LEGACY_IDLE_HOOK*
- *CONFIG_FREERTOS_LEGACY_TICK_HOOK*
- *CONFIG_FREERTOS_MAX_TASK_NAME_LEN*
- *CONFIG_SUPPORT_STATIC_ALLOCATION*
- *CONFIG_ENABLE_STATIC_TASK_CLEAN_UP_HOOK*
- *CONFIG_TIMER_TASK_PRIORITY*
- *CONFIG_TIMER_TASK_STACK_DEPTH*
- *CONFIG_TIMER_QUEUE_LENGTH*
- *CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE*
- *CONFIG_FREERTOS_USE_TRACE_FACILITY*
- *CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS*
- *CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS*
- *CONFIG_FREERTOS_RUN_TIME_STATS_CLK*
- *CONFIG_FREERTOS_USE_TICKLESS_IDLE*
- *CONFIG_FREERTOS_IDLE_TIME_BEFORE_SLEEP*
- *CONFIG_FREERTOS_PORTMUX_DEBUG*
- *CONFIG_FREERTOS_PORTMUX_DEBUG_RECURSIVE*
- *CONFIG_DMA_RX_BUF_NUM*
- *CONFIG_DMA_TX_BUF_NUM*
- *CONFIG_EMAC_L2_TO_L3_RX_BUF_MODE*
- *CONFIG_EMAC_TASK_PRIORITY*
- *CONFIG_EMAC_TASK_STACK_SIZE*
- *CONFIG_ADC_CAL_EFUSE_TP_ENABLE*
- *CONFIG_ADC_CAL_EFUSE_VREF_ENABLE*
- *CONFIG_ADC_CAL_LUT_ENABLE*
- *CONFIG_SPI_FLASH_VERIFY_WRITE*
- *CONFIG_SPI_FLASH_LOG_FAILED_WRITE*
- *CONFIG_SPI_FLASH_WARN_SETTING_ZERO_TO_ONE*
- *CONFIG_SPI_FLASH_ENABLE_COUNTERS*
- *CONFIG_SPI_FLASH_ROM_DRIVER_PATCH*
- *CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS*
- *CONFIG_SPI_FLASH_YIELD_DURING_ERASE*
- *CONFIG_SPI_FLASH_ERASE_YIELD_DURATION_MS*

- *CONFIG_SPI_FLASH_ERASE_YIELD_TICKS*

OpenSSL

CONFIG_OPENSSL_DEBUG

Enable OpenSSL debugging

Enable OpenSSL debugging function.

If the option is enabled, "SSL_DEBUG" works.

Found in Component config > OpenSSL

CONFIG_OPENSSL_DEBUG_LEVEL

OpenSSL debugging level

OpenSSL debugging level.

Only function whose debugging level is higher than "OPENSSL_DEBUG_LEVEL" works.

For example: If OPENSSL_DEBUG_LEVEL = 2, you use function "SSL_DEBUG(1, "malloc failed")". Because 1 < 2, it will not print.

Found in Component config > OpenSSL

CONFIG_OPENSSL_LOWLEVEL_DEBUG

Enable OpenSSL low-level module debugging

If the option is enabled, low-level module debugging function of OpenSSL is enabled, e.g. mbedtls internal debugging function.

Found in Component config > OpenSSL

CONFIG_OPENSSL_ASSERT

Select OpenSSL assert function

OpenSSL function needs "assert" function to check if input parameters are valid.

If you want to use assert debugging function, "OPENSSL_DEBUG" should be enabled.

Found in Component config > OpenSSL

Available options:

- Do nothing (*CONFIG_OPENSSL_ASSERT_DO_NOTHING*)

  Do nothing and "SSL_ASSERT" does not work.

- Check and exit (*CONFIG_OPENSSL_ASSERT_EXIT*)

  Enable assert exiting, it will check and return error code.

- Show debugging message (*CONFIG_OPENSSL_ASSERT_DEBUG*)

  Enable assert debugging, it will check and show debugging message.

- Show debugging message and exit (*CONFIG_OPENSSL_ASSERT_DEBUG_EXIT*)

  Enable assert debugging and exiting, it will check, show debugging message and return error code.

- Show debugging message and block (*CONFIG_OPENSSL_ASSERT_DEBUG_BLOCK*)

  Enable assert debugging and blocking, it will check, show debugging message and block by "while (1);".

Virtual file system

CONFIG_SUPPRESS_SELECT_DEBUG_OUTPUT

Suppress select() related debug outputs

Select() related functions might produce an unconveniently lot of debug outputs when one sets the default log level to DEBUG or higher. It is possible to suppress these debug outputs by enabling this option.

Found in Component config > Virtual file system

mbedTLS

CONFIG_MBEDTLS_SSL_MAX_CONTENT_LEN

TLS maximum message content length

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).

Found in Component config > mbedTLS

CONFIG_MBEDTLS_DEBUG

Enable mbedTLS debugging

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.

Found in Component config > mbedTLS

CONFIG_MBEDTLS_HARDWARE_AES

Enable hardware AES acceleration

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.

Found in Component config > mbedTLS

CONFIG_MBEDTLS_HARDWARE_MPI

Enable hardware MPI (bignum) acceleration

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.

Found in Component config > mbedTLS

CONFIG_MBEDTLS_MPI_USE_INTERRUPT

Use interrupt for MPI operations

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.

Found in Component config > mbedTLS

CONFIG_MBEDTLS_HARDWARE_SHA
Enable hardware SHA acceleration

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.

Found in Component config > mbedTLS

CONFIG_MBEDTLS_HAVE_TIME
Enable mbedtls time

System has time.h and time(). The time does not need to be correct, only time differences are used,

Found in Component config > mbedTLS

CONFIG_MBEDTLS_HAVE_TIME_DATE
Enable mbedtls time data

System has time.h and time(), gmtime() and the clock is correct. The time needs to be correct (not necesarily 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".

Found in Component config > mbedTLS

CONFIG_MBEDTLS_TLS_MODE
TLS Protocol Role

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.

Found in Component config > mbedTLS

Available options:

- Server & Client (*CONFIG_MBEDTLS_TLS_SERVER_AND_CLIENT*)
- Server (*CONFIG_MBEDTLS_TLS_SERVER_ONLY*)
- Client (*CONFIG_MBEDTLS_TLS_CLIENT_ONLY*)
- None (*CONFIG_MBEDTLS_TLS_DISABLED*)

TLS Key Exchange Methods

- *CONFIG_MBEDTLS_PSK_MODES*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_PSK*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_PSK*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_PSK*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_RSA_PSK*

- *CONFIG_MBEDTLS_KEY_EXCHANGE_RSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA*
- *CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_RSA*

CONFIG_MBEDTLS_PSK_MODES
Enable pre-shared-key ciphersuites

Enable to show configuration for different types of pre-shared-key TLS authentatication methods.

Leaving this options disabled will save code size if they are not used.

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_PSK
Enable PSK based ciphersuite modes

Enable to support symmetric key PSK (pre-shared-key) TLS key exchange modes.

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_PSK
Enable DHE-PSK based ciphersuite modes

Enable to support Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_PSK
Enable ECDHE-PSK based ciphersuite modes

Enable to support Elliptic-Curve-Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_RSA_PSK
Enable RSA-PSK based ciphersuite modes

Enable to support RSA PSK (pre-shared-key) TLS authentication modes.

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_RSA
Enable RSA-only based ciphersuite modes

Enable to support ciphersuites with prefix TLS-RSA-WITH-

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA
Enable DHE-RSA based ciphersuite modes

Enable to support ciphersuites with prefix TLS-DHE-RSA-WITH-

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE
Support Elliptic Curve based ciphersuites

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.

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA
Enable ECDHE-RSA based ciphersuite modes

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA
Enable ECDHE-ECDSA based ciphersuite modes

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA
Enable ECDH-ECDSA based ciphersuite modes

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_RSA
Enable ECDH-RSA based ciphersuite modes

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

Found in Component config > mbedTLS > TLS Key Exchange Methods

CONFIG_MBEDTLS_SSL_RENEGOTIATION
Support TLS renegotiation

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.

Found in Component config > mbedTLS

CONFIG_MBEDTLS_SSL_PROTO_SSL3
Legacy SSL 3.0 support

Support the legacy SSL 3.0 protocol. Most servers will speak a newer TLS protocol these days.

Found in Component config > mbedTLS

CONFIG_MBEDTLS_SSL_PROTO_TLS1
Support TLS 1.0 protocol

Found in Component config > mbedTLS

CONFIG_MBEDTLS_SSL_PROTO_TLS1_1
Support TLS 1.1 protocol

Found in Component config > mbedTLS

CONFIG_MBEDTLS_SSL_PROTO_TLS1_2
Support TLS 1.2 protocol

Found in Component config > mbedTLS

CONFIG_MBEDTLS_SSL_PROTO_DTLS
Support DTLS protocol (all versions)

Requires TLS 1.1 to be enabled for DTLS 1.0 Requires TLS 1.2 to be enabled for DTLS 1.2

Found in Component config > mbedTLS

CONFIG_MBEDTLS_SSL_ALPN
Support ALPN (Application Layer Protocol Negotiation)

Disabling this option will save some code size if it is not needed.

Found in Component config > mbedTLS

CONFIG_MBEDTLS_SSL_SESSION_TICKETS
TLS: Support RFC 5077 SSL session tickets

Support RFC 5077 session tickets. See mbedTLS documentation for more details.

Disabling this option will save some code size.

Found in Component config > mbedTLS

Symmetric Ciphers

- *CONFIG_MBEDTLS_AES_C*
- *CONFIG_MBEDTLS_CAMELLIA_C*
- *CONFIG_MBEDTLS_DES_C*
- *CONFIG_MBEDTLS_RC4_MODE*
- *CONFIG_MBEDTLS_BLOWFISH_C*
- *CONFIG_MBEDTLS_XTEA_C*
- *CONFIG_MBEDTLS_CCM_C*
- *CONFIG_MBEDTLS_GCM_C*

CONFIG_MBEDTLS_AES_C
AES block cipher

Found in Component config > mbedTLS > Symmetric Ciphers

CONFIG_MBEDTLS_CAMELLIA_C
Camellia block cipher

Found in Component config > mbedTLS > Symmetric Ciphers

CONFIG_MBEDTLS_DES_C
DES block cipher (legacy, insecure)

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.

Found in Component config > mbedTLS > Symmetric Ciphers

CONFIG_MBEDTLS_RC4_MODE
RC4 Stream Cipher (legacy, insecure)

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.

Found in Component config > mbedTLS > Symmetric Ciphers

Available options:

- Disabled (*CONFIG_MBEDTLS_RC4_DISABLED*)
- Enabled, not in default ciphersuites (*CONFIG_MBEDTLS_RC4_ENABLED_NO_DEFAULT*)
- Enabled (*CONFIG_MBEDTLS_RC4_ENABLED*)

CONFIG_MBEDTLS_BLOWFISH_C

Blowfish block cipher (read help)

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.

Found in Component config > mbedTLS > Symmetric Ciphers

CONFIG_MBEDTLS_XTEA_C

XTEA block cipher

Enables the XTEA block cipher.

Found in Component config > mbedTLS > Symmetric Ciphers

CONFIG_MBEDTLS_CCM_C

CCM (Counter with CBC-MAC) block cipher modes

Enable Counter with CBC-MAC (CCM) modes for AES and/or Camellia ciphers.

Disabling this option saves some code size.

Found in Component config > mbedTLS > Symmetric Ciphers

CONFIG_MBEDTLS_GCM_C

GCM (Galois/Counter) block cipher modes

Enable Galois/Counter Mode for AES and/or Camellia ciphers.

This option is generally faster than CCM.

Found in Component config > mbedTLS > Symmetric Ciphers

CONFIG_MBEDTLS_RIPEMD160_C

Enable RIPEMD-160 hash algorithm

Enable the RIPEMD-160 hash algorithm.

Found in Component config > mbedTLS

Certificates

- *CONFIG_MBEDTLS_PEM_PARSE_C*
- *CONFIG_MBEDTLS_PEM_WRITE_C*
- *CONFIG_MBEDTLS_X509_CRL_PARSE_C*
- *CONFIG_MBEDTLS_X509_CSR_PARSE_C*

CONFIG_MBEDTLS_PEM_PARSE_C

Read & Parse PEM formatted certificates

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.

Found in Component config > mbedTLS > Certificates

CONFIG_MBEDTLS_PEM_WRITE_C
Write PEM formatted certificates

Enable writing of PEM formatted certificates.

If writing certificate data only in DER format, disabling this option will save some code size.

Found in Component config > mbedTLS > Certificates

CONFIG_MBEDTLS_X509_CRL_PARSE_C
X.509 CRL parsing

Support for parsing X.509 Certifificate Revocation Lists.

Found in Component config > mbedTLS > Certificates

CONFIG_MBEDTLS_X509_CSR_PARSE_C
X.509 CSR parsing

Support for parsing X.509 Certifificate Signing Requests

Found in Component config > mbedTLS > Certificates

CONFIG_MBEDTLS_ECP_C

Elliptic Curve Ciphers

- *CONFIG_MBEDTLS_ECDH_C*
- *CONFIG_MBEDTLS_ECDSA_C*
- *CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED*
- *CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED*
- *CONFIG_MBEDTLS_ECP_NIST_OPTIM*

Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECDH_C
Elliptic Curve Diffie-Hellman (ECDH)

Enable ECDH. Needed to use ECDHE-xxx TLS ciphersuites.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECDSA_C
Elliptic Curve DSA

Enable ECDSA. Needed to use ECDSA-xxx TLS ciphersuites.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED
Enable SECP192R1 curve

Enable support for SECP192R1 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED
Enable SECP224R1 curve

Enable support for SECP224R1 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED
Enable SECP256R1 curve

Enable support for SECP256R1 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED
Enable SECP384R1 curve

Enable support for SECP384R1 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED
Enable SECP521R1 curve

Enable support for SECP521R1 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED
Enable SECP192K1 curve

Enable support for SECP192K1 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED
Enable SECP224K1 curve

Enable support for SECP224K1 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED
Enable SECP256K1 curve

Enable support for SECP256K1 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED
Enable BP256R1 curve

support for DP Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED
Enable BP384R1 curve

support for DP Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED
Enable BP512R1 curve

support for DP Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED
Enable CURVE25519 curve

Enable support for CURVE25519 Elliptic Curve.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

CONFIG_MBEDTLS_ECP_NIST_OPTIM
NIST 'modulo p' optimisations

NIST 'modulo p' optimisations increase Elliptic Curve operation performance.

Disabling this option saves some code size.

Found in Component config > mbedTLS > Elliptic Curve Ciphers

Heap memory debugging

CONFIG_HEAP_CORRUPTION_DETECTION
Heap corruption detection

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.

Found in Component config > Heap memory debugging

Available options:

- Basic (no poisoning) (*CONFIG_HEAP_POISONING_DISABLED*)
- Light impact (*CONFIG_HEAP_POISONING_LIGHT*)
- Comprehensive (*CONFIG_HEAP_POISONING_COMPREHENSIVE*)

CONFIG_HEAP_TRACING
Enable heap tracing

Enables the heap tracing API defined in esp_heap_trace.h.

This function causes a moderate increase in IRAM code side and a minor increase in heap function (malloc/free/realloc) CPU overhead, even when the tracing feature is not used. So it's best to keep it disabled unless tracing is being used.

Found in Component config > Heap memory debugging

CONFIG_HEAP_TRACING_STACK_DEPTH
Heap tracing stack depth

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.

Found in Component config > Heap memory debugging

CONFIG_HEAP_TASK_TRACKING
Enable heap task tracking

Enables tracking the task responsible for each heap allocation.

This function depends on heap poisoning being enabled and adds four more bytes of overhead for each block allocated.

Found in Component config > Heap memory debugging

PThreads

CONFIG_ESP32_PTHREAD_TASK_PRIO_DEFAULT
Default task priority

Priority used to create new tasks with default pthread parameters.

Found in Component config > PThreads

CONFIG_ESP32_PTHREAD_TASK_STACK_SIZE_DEFAULT
Default task stack size

Stack size used to create new tasks with default pthread parameters.

Found in Component config > PThreads

Driver configurations

ADC configuration

- *CONFIG_ADC_FORCE_XPD_FSM*
- *CONFIG_ADC2_DISABLE_DAC*

CONFIG_ADC_FORCE_XPD_FSM
Use the FSM to control ADC power

ADC power can be controlled by the FSM instead of software. This allows the ADC to be shut off when it is not working leading to lower power consumption. However using the FSM control ADC power will increase the noise of ADC.

Found in Component config > Driver configurations > ADC configuration

CONFIG_ADC2_DISABLE_DAC
Disable DAC when ADC2 is used on GPIO 25 and 26

If this is set, the ADC2 driver will disables the output of the DAC corresponding to the specified channel. This is the default value.

For testing, disable this option so that we can measure the output of DAC by internal ADC.

Found in Component config > Driver configurations > ADC configuration

SPI master configuration

- *CONFIG_SPI_MASTER_IN_IRAM*
- *CONFIG_SPI_MASTER_ISR_IN_IRAM*

CONFIG_SPI_MASTER_IN_IRAM

Place transmitting functions of SPI master into IRAM

Normally only the ISR of SPI master is placed in the IRAM, so that it can work without the flash when interrupt is triggered. For other functions, there's some possibility that the flash cache miss when running inside and out of SPI functions, which may increase the interval of SPI transactions. Enable this to put `queue\_trans`, `get\_trans\_result` and `transmit` functions into the IRAM to avoid possible cache miss.

During unit test, this is enabled to measure the ideal case of api.

Found in Component config > Driver configurations > SPI master configuration

CONFIG_SPI_MASTER_ISR_IN_IRAM

Place SPI master ISR function into IRAM

Place the SPI master ISR in to IRAM to avoid possible cache miss.

Also you can forbid the ISR being disabled during flash writing access, by add ESP_INTR_FLAG_IRAM when initializing the driver.

Found in Component config > Driver configurations > SPI master configuration

Log output

CONFIG_LOG_DEFAULT_LEVEL

Default log verbosity

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.

Found in Component config > Log output

Available options:

- No output (*CONFIG_LOG_DEFAULT_LEVEL_NONE*)
- Error (*CONFIG_LOG_DEFAULT_LEVEL_ERROR*)
- Warning (*CONFIG_LOG_DEFAULT_LEVEL_WARN*)
- Info (*CONFIG_LOG_DEFAULT_LEVEL_INFO*)
- Debug (*CONFIG_LOG_DEFAULT_LEVEL_DEBUG*)
- Verbose (*CONFIG_LOG_DEFAULT_LEVEL_VERBOSE*)

CONFIG_LOG_COLORS

Use ANSI terminal colors in log output

Enable ANSI terminal color codes in bootloader output.

In order to view these, your terminal program must support ANSI color codes.

Found in Component config > Log output

CONFIG_AWS_IOT_SDK

Amazon Web Services IoT Platform

Amazon Web Services IoT Platform

Select this option to enable support for the AWS IoT platform, via the esp-idf component for the AWS IoT Device C SDK.

CONFIG_AWS_IOT_MQTT_HOST
AWS IoT Endpoint Hostname

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.

Found in Component config > Amazon Web Services IoT Platform

CONFIG_AWS_IOT_MQTT_PORT
AWS IoT MQTT Port

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.

Found in Component config > Amazon Web Services IoT Platform

CONFIG_AWS_IOT_MQTT_TX_BUF_LEN
MQTT TX Buffer Length

Maximum MQTT transmit buffer size. This is the maximum MQTT message length (including protocol overhead) which can be sent.

Sending longer messages will fail.

Found in Component config > Amazon Web Services IoT Platform

CONFIG_AWS_IOT_MQTT_RX_BUF_LEN
MQTT RX Buffer Length

Maximum MQTT receive buffer size. This is the maximum MQTT message length (including protocol overhead) which can be received.

Longer messages are dropped.

Found in Component config > Amazon Web Services IoT Platform

CONFIG_AWS_IOT_MQTT_NUM_SUBSCRIBE_HANDLERS
Maximum MQTT Topic Filters

Maximum number of concurrent MQTT topic filters.

Found in Component config > Amazon Web Services IoT Platform

CONFIG_AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL
Auto reconnect initial interval (ms)

Initial delay before making first reconnect attempt, if the AWS IoT connection fails. Client will perform exponential backoff, starting from this value.

Found in Component config > Amazon Web Services IoT Platform

CONFIG_AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL
Auto reconnect maximum interval (ms)

Maximum delay between reconnection attempts. If the exponentially increased delay interval reaches this value, the client will stop automatically attempting to reconnect.

Found in Component config > Amazon Web Services IoT Platform

Thing Shadow

- *CONFIG_AWS_IOT_OVERRIDE_THING_SHADOW_RX_BUFFER*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_RX_BUFFER*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_UNIQUE_CLIENT_ID_BYTES*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_ACKS*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_THINGNAMES*
- *CONFIG_AWS_IOT_SHADOW_MAX_JSON_TOKEN_EXPECTED*
- *CONFIG_AWS_IOT_SHADOW_MAX_SHADOW_TOPIC_LENGTH_WITHOUT_THINGNAME*
- *CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_THING_NAME*

CONFIG_AWS_IOT_OVERRIDE_THING_SHADOW_RX_BUFFER
Override Shadow RX buffer size

Allows setting a different Thing Shadow RX buffer size. This is the maximum size of a Thing Shadow message in bytes, plus one.

If not overridden, the default value is the MQTT RX Buffer length plus one. If overriden, do not set higher than the default value.

Found in Component config > Amazon Web Services IoT Platform > Thing Shadow

CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_RX_BUFFER
Maximum RX Buffer (bytes)

Allows setting a different Thing Shadow RX buffer size. This is the maximum size of a Thing Shadow message in bytes, plus one.

Found in Component config > Amazon Web Services IoT Platform > Thing Shadow

CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_UNIQUE_CLIENT_ID_BYTES
Maximum unique client ID size (bytes)

Maximum size of the Unique Client Id.

Found in Component config > Amazon Web Services IoT Platform > Thing Shadow

CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_ACKS
Maximum simultaneous responses

At any given time we will wait for this many responses. This will correlate to the rate at which the shadow actions are requested

Found in Component config > Amazon Web Services IoT Platform > Thing Shadow

CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_THINGNAMES
Maximum simultaneous Thing Name operations

We could perform shadow action on any thing Name and this is maximum Thing Names we can act on at any given time

Found in Component config > Amazon Web Services IoT Platform > Thing Shadow

CONFIG_AWS_IOT_SHADOW_MAX_JSON_TOKEN_EXPECTED
Maximum expected JSON tokens

These are the max tokens that is expected to be in the Shadow JSON document. Includes the metadata which is published

Found in Component config > Amazon Web Services IoT Platform > Thing Shadow

CONFIG_AWS_IOT_SHADOW_MAX_SHADOW_TOPIC_LENGTH_WITHOUT_THINGNAME
Maximum topic length (not including Thing Name)

All shadow actions have to be published or subscribed to a topic which is of the format $aws/things/{thingName}/shadow/update/accepted. This refers to the size of the topic without the Thing Name

Found in Component config > Amazon Web Services IoT Platform > Thing Shadow

CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_THING_NAME
Maximum Thing Name length

Maximum length of a Thing Name.

Found in Component config > Amazon Web Services IoT Platform > Thing Shadow

NVS

CONFIG_MP_BLOB_SUPPORT
Enable multi-page blob support

When enabled, blobs larger than the sector size are split and stored on multiple sectors. This removes the earlier limitation of 1984 bytes for storing data-blobs.

Found in Component config > NVS

CONFIG_NVS_ENCRYPTION
Enable NVS encryption

This option enables encryption for NVS. When enabled, AES-XTS is used to encrypt the complete NVS data, except the page headers. It requires XTS encryption keys to be stored in an encrypted partition. This means enabling flash encryption is a pre-requisite for this feature.

Found in Component config > NVS

SPIFFS Configuration

CONFIG_SPIFFS_MAX_PARTITIONS
Maximum Number of Partitions

Define maximum number of partitions that can be mounted.

Found in Component config > SPIFFS Configuration

SPIFFS Cache Configuration

- *CONFIG_SPIFFS_CACHE*
- *CONFIG_SPIFFS_CACHE_WR*
- *CONFIG_SPIFFS_CACHE_STATS*

CONFIG_SPIFFS_CACHE
Enable SPIFFS Cache

Enables/disable memory read caching of nucleus file system operations.

Found in Component config > SPIFFS Configuration > SPIFFS Cache Configuration

CONFIG_SPIFFS_CACHE_WR
Enable SPIFFS Write Caching

Enables memory write caching for file descriptors in hydrogen.

Found in Component config > SPIFFS Configuration > SPIFFS Cache Configuration

CONFIG_SPIFFS_CACHE_STATS
Enable SPIFFS Cache Statistics

Enable/disable statistics on caching. Debug/test purpose only.

Found in Component config > SPIFFS Configuration > SPIFFS Cache Configuration

CONFIG_SPIFFS_PAGE_CHECK
Enable SPIFFS Page Check

Always check header of each accessed page to ensure consistent state. If enabled it will increase number of reads from flash, especially if cache is disabled.

Found in Component config > SPIFFS Configuration

CONFIG_SPIFFS_GC_MAX_RUNS
Set Maximum GC Runs

Define maximum number of GC runs to perform to reach desired free pages.

Found in Component config > SPIFFS Configuration

CONFIG_SPIFFS_GC_STATS
Enable SPIFFS GC Statistics

Enable/disable statistics on gc. Debug/test purpose only.

Found in Component config > SPIFFS Configuration

CONFIG_SPIFFS_PAGE_SIZE
SPIFFS logical page size

Logical page size of SPIFFS partition, in bytes. Must be multiple of flash page size (which is usually 256 bytes). Larger page sizes reduce overhead when storing large files, and improve filesystem performance when reading large files. Smaller page sizes reduce overhead when storing small (< page size) files.

Found in Component config > SPIFFS Configuration

CONFIG_SPIFFS_OBJ_NAME_LEN
Set SPIFFS Maximum Name Length

Object name maximum length. Note that this length include the zero-termination character, meaning maximum string of characters can at most be SPIFFS_OBJ_NAME_LEN - 1.

SPIFFS_OBJ_NAME_LEN + SPIFFS_META_LENGTH should not exceed SPIFFS_PAGE_SIZE - 64.

Found in Component config > SPIFFS Configuration

CONFIG_SPIFFS_USE_MAGIC
Enable SPIFFS Filesystem Magic

Enable this to have an identifiable spiffs filesystem. This will look for a magic in all sectors to determine if this is a valid spiffs system or not at mount time.

Found in Component config > SPIFFS Configuration

CONFIG_SPIFFS_USE_MAGIC_LENGTH
Enable SPIFFS Filesystem Length Magic

If this option is enabled, the magic will also be dependent on the length of the filesystem. For example, a filesystem configured and formatted for 4 megabytes will not be accepted for mounting with a configuration defining the filesystem as 2 megabytes.

Found in Component config > SPIFFS Configuration

CONFIG_SPIFFS_META_LENGTH
Size of per-file metadata field

This option sets the number of extra bytes stored in the file header. These bytes can be used in an application-specific manner. Set this to at least 4 bytes to enable support for saving file modification time.

SPIFFS_OBJ_NAME_LEN + SPIFFS_META_LENGTH should not exceed SPIFFS_PAGE_SIZE - 64.

Found in Component config > SPIFFS Configuration

CONFIG_SPIFFS_USE_MTIME
Save file modification time

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.

Found in Component config > SPIFFS Configuration

Debug Configuration

- *CONFIG_SPIFFS_DBG*
- *CONFIG_SPIFFS_API_DBG*
- *CONFIG_SPIFFS_GC_DBG*
- *CONFIG_SPIFFS_CACHE_DBG*
- *CONFIG_SPIFFS_CHECK_DBG*
- *CONFIG_SPIFFS_TEST_VISUALISATION*

CONFIG_SPIFFS_DBG
Enable general SPIFFS debug

Enabling this option will print general debug mesages to the console.

Found in Component config > SPIFFS Configuration > Debug Configuration

CONFIG_SPIFFS_API_DBG
Enable SPIFFS API debug

Enabling this option will print API debug mesages to the console.

Found in Component config > SPIFFS Configuration > Debug Configuration

CONFIG_SPIFFS_GC_DBG
Enable SPIFFS Garbage Cleaner debug

Enabling this option will print GC debug mesages to the console.

Found in Component config > SPIFFS Configuration > Debug Configuration

CONFIG_SPIFFS_CACHE_DBG
Enable SPIFFS Cache debug

Enabling this option will print cache debug mesages to the console.

Found in Component config > SPIFFS Configuration > Debug Configuration

CONFIG_SPIFFS_CHECK_DBG
Enable SPIFFS Filesystem Check debug

Enabling this option will print Filesystem Check debug mesages to the console.

Found in Component config > SPIFFS Configuration > Debug Configuration

CONFIG_SPIFFS_TEST_VISUALISATION
Enable SPIFFS Filesystem Visualization

Enable this option to enable SPIFFS_vis function in the API.

Found in Component config > SPIFFS Configuration > Debug Configuration

Wear Levelling

CONFIG_WL_SECTOR_SIZE
Wear Levelling library sector size

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).

Found in Component config > Wear Levelling

Available options:

- 512 (*CONFIG_WL_SECTOR_SIZE_512*)
- 4096 (*CONFIG_WL_SECTOR_SIZE_4096*)

CONFIG_WL_SECTOR_MODE
Sector store mode

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.

Found in Component config > Wear Levelling

Available options:

- Perfomance (*CONFIG_WL_SECTOR_MODE_PERF*)
- Safety (*CONFIG_WL_SECTOR_MODE_SAFE*)

tcpip adapter

CONFIG_IP_LOST_TIMER_INTERVAL

IP Address lost timer interval (seconds)

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.

Found in Component config > tcpip adapter

FAT Filesystem support

CONFIG_FATFS_CHOOSE_CODEPAGE

OEM Code Page

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.

Found in Component config > FAT Filesystem support

Available options:

- Dynamic (all code pages supported) (*CONFIG_FATFS_CODEPAGE_DYNAMIC*)
- US (CP437) (*CONFIG_FATFS_CODEPAGE_437*)
- Arabic (CP720) (*CONFIG_FATFS_CODEPAGE_720*)
- Greek (CP737) (*CONFIG_FATFS_CODEPAGE_737*)
- KBL (CP771) (*CONFIG_FATFS_CODEPAGE_771*)
- Baltic (CP775) (*CONFIG_FATFS_CODEPAGE_775*)
- Latin 1 (CP850) (*CONFIG_FATFS_CODEPAGE_850*)
- Latin 2 (CP852) (*CONFIG_FATFS_CODEPAGE_852*)
- Cyrillic (CP855) (*CONFIG_FATFS_CODEPAGE_855*)
- Turkish (CP857) (*CONFIG_FATFS_CODEPAGE_857*)
- Portugese (CP860) (*CONFIG_FATFS_CODEPAGE_860*)
- Icelandic (CP861) (*CONFIG_FATFS_CODEPAGE_861*)
- Hebrew (CP862) (*CONFIG_FATFS_CODEPAGE_862*)
- Canadian French (CP863) (*CONFIG_FATFS_CODEPAGE_863*)
- Arabic (CP864) (*CONFIG_FATFS_CODEPAGE_864*)
- Nordic (CP865) (*CONFIG_FATFS_CODEPAGE_865*)
- Russian (CP866) (*CONFIG_FATFS_CODEPAGE_866*)
- Greek 2 (CP869) (*CONFIG_FATFS_CODEPAGE_869*)
- Japanese (DBCS) (CP932) (*CONFIG_FATFS_CODEPAGE_932*)
- Simplified Chinese (DBCS) (CP936) (*CONFIG_FATFS_CODEPAGE_936*)

- Korean (DBCS) (CP949) (*CONFIG_FATFS_CODEPAGE_949*)
- Traditional Chinese (DBCS) (CP950) (*CONFIG_FATFS_CODEPAGE_950*)

CONFIG_FATFS_LONG_FILENAMES

Long filename 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.

Found in Component config > FAT Filesystem support

Available options:

- No long filenames (*CONFIG_FATFS_LFN_NONE*)
- Long filename buffer in heap (*CONFIG_FATFS_LFN_HEAP*)
- Long filename buffer on stack (*CONFIG_FATFS_LFN_STACK*)

CONFIG_FATFS_MAX_LFN

Max long filename length

Maximum long filename length. Can be reduced to save RAM.

Found in Component config > FAT Filesystem support

CONFIG_FATFS_API_ENCODING

API character encoding

Choose encoding for character and string arguments/returns when using FATFS APIs. The encoding of arguments will usually depend on text editor settings.

Found in Component config > FAT Filesystem support

Available options:

- API uses ANSI/OEM encoding (*CONFIG_FATFS_API_ENCODING_ANSI_OEM*)
- API uses UTF-16 encoding (*CONFIG_FATFS_API_ENCODING_UTF_16*)
- API uses UTF-8 encoding (*CONFIG_FATFS_API_ENCODING_UTF_8*)

CONFIG_FATFS_FS_LOCK

Number of simultaneously open files protected by lock function

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.

Found in Component config > FAT Filesystem support

CONFIG_FATFS_TIMEOUT_MS

Timeout for acquiring a file lock, ms

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 lenghty operation, another task will wait for the first task to release the lock, and time out after amount of time set by this option.

Found in Component config > FAT Filesystem support

CONFIG_FATFS_PER_FILE_CACHE
Use separate cache for each file

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.

Found in Component config > FAT Filesystem support

Bluetooth

CONFIG_BT_ENABLED
Bluetooth

Select this option to enable Bluetooth and show the submenu with Bluetooth configuration choices.

Found in Component config > Bluetooth

CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH
BR/EDR Sync(SCO/eSCO) default data path

SCO data path, i.e. HCI or PCM. SCO data can be sent/received through HCI synchronous packets, or the data can be routed to on-chip PCM module on ESP32. PCM input/output signals can be "matrixed" to GPIOs. The default data path can also be set using API "esp_bredr_sco_datapath_set"

Found in Component config > Bluetooth

Available options:

- HCI (*CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH_HCI*)
- PCM (*CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH_PCM*)

CONFIG_BTDM_CONTROLLER_AUTO_LATENCY
Auto latency

BLE auto latency, used to enhance classic BT performance while classic BT and BLE are enabled at the same time.

Found in Component config > Bluetooth

CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE_CHOICE
The cpu core which bluetooth controller run

Specify the cpu core to run bluetooth controller. Can not specify no-affinity.

Found in Component config > Bluetooth

Available options:

- Core 0 (PRO CPU) (*CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE_0*)
- Core 1 (APP CPU) (*CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE_1*)

CONFIG_BTDM_CONTROLLER_HCI_MODE_CHOICE

HCI mode

Speicify HCI mode as VHCI or UART(H4)

Found in Component config > Bluetooth

Available options:

- VHCI (*CONFIG_BTDM_CONTROLLER_HCI_MODE_VHCI*)

  Normal option. Mostly, choose this VHCI when bluetooth host run on ESP32, too.

- UART(H4) (*CONFIG_BTDM_CONTROLLER_HCI_MODE_UART_H4*)

  If use external bluetooth host which run on other hardware and use UART as the HCI interface, choose this option.

HCI UART(H4) Options

- *CONFIG_BT_HCI_UART_NO*
- *CONFIG_BT_HCI_UART_BAUDRATE*

CONFIG_BT_HCI_UART_NO

UART Number for HCI

Uart number for HCI. The available uart is UART1 and UART2.

Found in Component config > Bluetooth > HCI UART(H4) Options

CONFIG_BT_HCI_UART_BAUDRATE

UART Baudrate for HCI

UART Baudrate for HCI. Please use standard baudrate.

Found in Component config > Bluetooth > HCI UART(H4) Options

MODEM SLEEP Options

- *CONFIG_BTDM_CONTROLLER_MODEM_SLEEP*
- *CONFIG_BTDM_MODEM_SLEEP_MODE*
- *CONFIG_BTDM_LOW_POWER_CLOCK*

CONFIG_BTDM_CONTROLLER_MODEM_SLEEP

Bluetooth modem sleep

Enable/disable bluetooth controller low power mode.

Found in Component config > Bluetooth > MODEM SLEEP Options

CONFIG_BTDM_MODEM_SLEEP_MODE

Bluetooth Modem sleep mode

To select which strategy to use for modem sleep

Found in Component config > Bluetooth > MODEM SLEEP Options

Available options:

- ORIG Mode(sleep with low power clock) (*CONFIG_BTDM_MODEM_SLEEP_MODE_ORIG*)

  ORIG mode is a bluetooth sleep mode that can be used for dual mode controller. In this mode, bluetooth controller sleeps between BR/EDR frames and BLE events. A low power clock is used to maintain bluetooth reference clock.

- EVED Mode(For internal test only) (*CONFIG_BTDM_MODEM_SLEEP_MODE_EVED*)

  EVED mode is for BLE only and is only for internal test. Do not use it for production. this mode is not compatible with DFS nor light sleep

CONFIG_BTDM_LOW_POWER_CLOCK

Bluetooth low power clock

Select the low power clock source for bluetooth controller

Found in Component config > Bluetooth > MODEM SLEEP Options

Available options:

- Main crystal (*CONFIG_BTDM_LPCLK_SEL_MAIN_XTAL*)

  Main crystal can be used as low power clock for bluetooth modem sleep. If this option is selected, bluetooth modem sleep can work under Dynamic Frequency Scaling(DFS) enabled, but cannot work when light sleep is enabled. Main crystal has a relatively better performance than other bluetooth low power clock sources.

- External 32kHz crystal (*CONFIG_BTDM_LPCLK_SEL_EXT_32K_XTAL*)

CONFIG_BLUEDROID_ENABLED

Bluedroid Enable

- *CONFIG_BLUEDROID_PINNED_TO_CORE_CHOICE*
- CONFIG_BLUEDROID_PINNED_TO_CORE
- *CONFIG_BTC_TASK_STACK_SIZE*
- *CONFIG_BLUEDROID_MEM_DEBUG*
- *CONFIG_CLASSIC_BT_ENABLED*
- *CONFIG_A2DP_ENABLE*
- *CONFIG_A2DP_SINK_TASK_STACK_SIZE*
- *CONFIG_A2DP_SOURCE_TASK_STACK_SIZE*
- *CONFIG_BT_SPP_ENABLED*
- *CONFIG_HFP_ENABLE*
- *CONFIG_HFP_ROLE*
- *CONFIG_HFP_AUDIO_DATA_PATH*
- *CONFIG_GATTS_ENABLE*
- *CONFIG_GATTC_ENABLE*
- *CONFIG_GATTC_CACHE_NVS_FLASH*
- *CONFIG_BLE_SMP_ENABLE*
- *CONFIG_SMP_SLAVE_CON_PARAMS_UPD_ENABLE*

- *CONFIG_BT_STACK_NO_LOG*
- *CONFIG_HCI_INITIAL_TRACE_LEVEL*
- *CONFIG_HCI_INITIAL_TRACE_LEVEL*
- *CONFIG_BTM_INITIAL_TRACE_LEVEL*
- *CONFIG_BTM_INITIAL_TRACE_LEVEL*
- *CONFIG_L2CAP_INITIAL_TRACE_LEVEL*
- *CONFIG_L2CAP_INITIAL_TRACE_LEVEL*
- *CONFIG_RFCOMM_INITIAL_TRACE_LEVEL*
- *CONFIG_RFCOMM_INITIAL_TRACE_LEVEL*
- *CONFIG_SDP_INITIAL_TRACE_LEVEL*
- *CONFIG_SDP_INITIAL_TRACE_LEVEL*
- *CONFIG_GAP_INITIAL_TRACE_LEVEL*
- *CONFIG_GAP_INITIAL_TRACE_LEVEL*
- *CONFIG_BNEP_INITIAL_TRACE_LEVEL*
- *CONFIG_BNEP_INITIAL_TRACE_LEVEL*
- *CONFIG_PAN_INITIAL_TRACE_LEVEL*
- *CONFIG_PAN_INITIAL_TRACE_LEVEL*
- *CONFIG_A2D_INITIAL_TRACE_LEVEL*
- *CONFIG_A2D_INITIAL_TRACE_LEVEL*
- *CONFIG_AVDT_INITIAL_TRACE_LEVEL*
- *CONFIG_AVDT_INITIAL_TRACE_LEVEL*
- *CONFIG_AVCT_INITIAL_TRACE_LEVEL*
- *CONFIG_AVCT_INITIAL_TRACE_LEVEL*
- *CONFIG_AVRC_INITIAL_TRACE_LEVEL*
- *CONFIG_AVRC_INITIAL_TRACE_LEVEL*
- *CONFIG_MCA_INITIAL_TRACE_LEVEL*
- *CONFIG_MCA_INITIAL_TRACE_LEVEL*
- *CONFIG_HID_INITIAL_TRACE_LEVEL*
- *CONFIG_HID_INITIAL_TRACE_LEVEL*
- *CONFIG_APPL_INITIAL_TRACE_LEVEL*
- *CONFIG_APPL_INITIAL_TRACE_LEVEL*
- *CONFIG_GATT_INITIAL_TRACE_LEVEL*
- *CONFIG_GATT_INITIAL_TRACE_LEVEL*
- *CONFIG_SMP_INITIAL_TRACE_LEVEL*
- *CONFIG_SMP_INITIAL_TRACE_LEVEL*
- *CONFIG_BTIF_INITIAL_TRACE_LEVEL*

- *CONFIG_BTIF_INITIAL_TRACE_LEVEL*
- *CONFIG_BTC_INITIAL_TRACE_LEVEL*
- *CONFIG_BTC_INITIAL_TRACE_LEVEL*
- *CONFIG_OSI_INITIAL_TRACE_LEVEL*
- *CONFIG_OSI_INITIAL_TRACE_LEVEL*
- *CONFIG_BLUFI_INITIAL_TRACE_LEVEL*
- *CONFIG_BLUFI_INITIAL_TRACE_LEVEL*
- *CONFIG_BT_ACL_CONNECTIONS*
- *CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST*
- *CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY*
- *CONFIG_BLE_HOST_QUEUE_CONGESTION_CHECK*
- *CONFIG_BLE_SCAN_DUPLICATE*
- *CONFIG_DUPLICATE_SCAN_CACHE_SIZE*
- *CONFIG_BLE_MESH_SCAN_DUPLICATE_EN*
- *CONFIG_MESH_DUPLICATE_SCAN_CACHE_SIZE*
- *CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_SUPPORTED*
- *CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_NUM*
- *CONFIG_BLE_ADV_REPORT_DISCARD_THRSHOLD*
- CONFIG_SMP_ENABLE
- *CONFIG_BLE_ACTIVE_SCAN_REPORT_ADV_SCAN_RSP_INDIVIDUALLY*
- *CONFIG_BLE_ESTABLISH_LINK_CONNECTION_TIMEOUT*

Bluedroid Enable

This enables the default Bluedroid Bluetooth stack

CONFIG_BLUEDROID_PINNED_TO_CORE_CHOICE

The cpu core which Bluedroid run

Which the cpu core to run Bluedroid. Can choose core0 and core1. Can not specify no-affinity.

Found in Component config > Bluetooth > Bluedroid Enable

Available options:

- Core 0 (PRO CPU) (*CONFIG_BLUEDROID_PINNED_TO_CORE_0*)
- Core 1 (APP CPU) (*CONFIG_BLUEDROID_PINNED_TO_CORE_1*)

CONFIG_BTC_TASK_STACK_SIZE

Bluetooth event (callback to application) task stack size

This select btc task stack size

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLUEDROID_MEM_DEBUG

Bluedroid memory debug

Bluedroid memory debug

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_CLASSIC_BT_ENABLED
Classic Bluetooth

For now this option needs "SMP_ENABLE" to be set to yes

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_A2DP_ENABLE
A2DP

Advanced Audio Distrubution Profile

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_A2DP_SINK_TASK_STACK_SIZE
A2DP sink (audio stream decoding) task stack size

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_A2DP_SOURCE_TASK_STACK_SIZE
A2DP source (audio stream encoding) task stack size

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BT_SPP_ENABLED
SPP

This enables the Serial Port Profile

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_HFP_ENABLE
Hands Free/Handset Profile

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_HFP_ROLE
Hands-free Profile Role configuration

Found in Component config > Bluetooth > Bluedroid Enable

Available options:

- Hands Free Unit (*CONFIG_HFP_CLIENT_ENABLE*)

CONFIG_HFP_AUDIO_DATA_PATH
audio(SCO) data path

SCO data path, i.e. HCI or PCM. This option is set using API "esp_bredr_sco_datapath_set" in Bluetooth host. Default SCO data path can also be set in Bluetooth Controller.

Found in Component config > Bluetooth > Bluedroid Enable

Available options:

- PCM (*CONFIG_HFP_AUDIO_DATA_PATH_PCM*)
- HCI (*CONFIG_HFP_AUDIO_DATA_PATH_HCI*)

CONFIG_GATTS_ENABLE
Include GATT server module(GATTS)

This option can be disabled when the app work only on gatt client mode

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_GATTC_ENABLE
Include GATT client module(GATTC)

This option can be close when the app work only on gatt server mode

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_GATTC_CACHE_NVS_FLASH
Save gattc cache data to nvs flash

This select can save gattc cache data to nvs flash

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_SMP_ENABLE
Include BLE security module(SMP)

This option can be close when the app not used the ble security connect.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_SMP_SLAVE_CON_PARAMS_UPD_ENABLE
Slave enable connection parameters update during pairing

In order to reduce the pairing time, slave actively initiates connection parameters update during pairing.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BT_STACK_NO_LOG
Disable BT debug logs (minimize bin size)

This select can save the rodata code size

Found in Component config > Bluetooth > Bluedroid Enable

BT DEBUG LOG LEVEL

CONFIG_HCI_INITIAL_TRACE_LEVEL
HCI layer

Define BT trace level for HCI layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_HCI_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_HCI_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_HCI_TRACE_LEVEL_WARNING*)
- API (*CONFIG_HCI_TRACE_LEVEL_API*)
- EVENT (*CONFIG_HCI_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_HCI_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_HCI_TRACE_LEVEL_VERBOSE*)

CONFIG_BTM_INITIAL_TRACE_LEVEL
BTM layer

Define BT trace level for BTM layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_BTM_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_BTM_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_BTM_TRACE_LEVEL_WARNING*)
- API (*CONFIG_BTM_TRACE_LEVEL_API*)
- EVENT (*CONFIG_BTM_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_BTM_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_BTM_TRACE_LEVEL_VERBOSE*)

CONFIG_L2CAP_INITIAL_TRACE_LEVEL

L2CAP layer

Define BT trace level for L2CAP layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_L2CAP_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_L2CAP_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_L2CAP_TRACE_LEVEL_WARNING*)
- API (*CONFIG_L2CAP_TRACE_LEVEL_API*)
- EVENT (*CONFIG_L2CAP_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_L2CAP_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_L2CAP_TRACE_LEVEL_VERBOSE*)

CONFIG_RFCOMM_INITIAL_TRACE_LEVEL

RFCOMM layer

Define BT trace level for RFCOMM layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_RFCOMM_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_RFCOMM_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_RFCOMM_TRACE_LEVEL_WARNING*)
- API (*CONFIG_RFCOMM_TRACE_LEVEL_API*)
- EVENT (*CONFIG_RFCOMM_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_RFCOMM_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_RFCOMM_TRACE_LEVEL_VERBOSE*)

CONFIG_SDP_INITIAL_TRACE_LEVEL

SDP layer

Define BT trace level for SDP layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_SDP_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_SDP_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_SDP_TRACE_LEVEL_WARNING*)
- API (*CONFIG_SDP_TRACE_LEVEL_API*)
- EVENT (*CONFIG_SDP_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_SDP_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_SDP_TRACE_LEVEL_VERBOSE*)

CONFIG_GAP_INITIAL_TRACE_LEVEL

GAP layer

Define BT trace level for GAP layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_GAP_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_GAP_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_GAP_TRACE_LEVEL_WARNING*)
- API (*CONFIG_GAP_TRACE_LEVEL_API*)
- EVENT (*CONFIG_GAP_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_GAP_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_GAP_TRACE_LEVEL_VERBOSE*)

CONFIG_BNEP_INITIAL_TRACE_LEVEL

BNEP layer

Define BT trace level for BNEP layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_BNEP_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_BNEP_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_BNEP_TRACE_LEVEL_WARNING*)
- API (*CONFIG_BNEP_TRACE_LEVEL_API*)
- EVENT (*CONFIG_BNEP_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_BNEP_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_BNEP_TRACE_LEVEL_VERBOSE*)

CONFIG_PAN_INITIAL_TRACE_LEVEL

PAN layer

Define BT trace level for PAN layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_PAN_TRACE_LEVEL_NONE*)

- ERROR (*CONFIG_PAN_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_PAN_TRACE_LEVEL_WARNING*)
- API (*CONFIG_PAN_TRACE_LEVEL_API*)
- EVENT (*CONFIG_PAN_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_PAN_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_PAN_TRACE_LEVEL_VERBOSE*)

CONFIG_A2D_INITIAL_TRACE_LEVEL

A2D layer

Define BT trace level for A2D layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_A2D_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_A2D_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_A2D_TRACE_LEVEL_WARNING*)
- API (*CONFIG_A2D_TRACE_LEVEL_API*)
- EVENT (*CONFIG_A2D_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_A2D_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_A2D_TRACE_LEVEL_VERBOSE*)

CONFIG_AVDT_INITIAL_TRACE_LEVEL

AVDT layer

Define BT trace level for AVDT layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_AVDT_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_AVDT_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_AVDT_TRACE_LEVEL_WARNING*)
- API (*CONFIG_AVDT_TRACE_LEVEL_API*)
- EVENT (*CONFIG_AVDT_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_AVDT_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_AVDT_TRACE_LEVEL_VERBOSE*)

CONFIG_AVCT_INITIAL_TRACE_LEVEL

AVCT layer

Define BT trace level for AVCT layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_AVCT_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_AVCT_TRACE_LEVEL_ERROR*)

- WARNING (*CONFIG_AVCT_TRACE_LEVEL_WARNING*)
- API (*CONFIG_AVCT_TRACE_LEVEL_API*)
- EVENT (*CONFIG_AVCT_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_AVCT_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_AVCT_TRACE_LEVEL_VERBOSE*)

CONFIG_AVRC_INITIAL_TRACE_LEVEL

AVRC layer

Define BT trace level for AVRC layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_AVRC_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_AVRC_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_AVRC_TRACE_LEVEL_WARNING*)
- API (*CONFIG_AVRC_TRACE_LEVEL_API*)
- EVENT (*CONFIG_AVRC_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_AVRC_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_AVRC_TRACE_LEVEL_VERBOSE*)

CONFIG_MCA_INITIAL_TRACE_LEVEL

MCA layer

Define BT trace level for MCA layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_MCA_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_MCA_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_MCA_TRACE_LEVEL_WARNING*)
- API (*CONFIG_MCA_TRACE_LEVEL_API*)
- EVENT (*CONFIG_MCA_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_MCA_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_MCA_TRACE_LEVEL_VERBOSE*)

CONFIG_HID_INITIAL_TRACE_LEVEL

HID layer

Define BT trace level for HID layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_HID_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_HID_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_HID_TRACE_LEVEL_WARNING*)

- API (*CONFIG_HID_TRACE_LEVEL_API*)
- EVENT (*CONFIG_HID_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_HID_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_HID_TRACE_LEVEL_VERBOSE*)

CONFIG_APPL_INITIAL_TRACE_LEVEL

APPL layer

Define BT trace level for APPL layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_APPL_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_APPL_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_APPL_TRACE_LEVEL_WARNING*)
- API (*CONFIG_APPL_TRACE_LEVEL_API*)
- EVENT (*CONFIG_APPL_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_APPL_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_APPL_TRACE_LEVEL_VERBOSE*)

CONFIG_GATT_INITIAL_TRACE_LEVEL

GATT layer

Define BT trace level for GATT layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_GATT_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_GATT_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_GATT_TRACE_LEVEL_WARNING*)
- API (*CONFIG_GATT_TRACE_LEVEL_API*)
- EVENT (*CONFIG_GATT_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_GATT_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_GATT_TRACE_LEVEL_VERBOSE*)

CONFIG_SMP_INITIAL_TRACE_LEVEL

SMP layer

Define BT trace level for SMP layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_SMP_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_SMP_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_SMP_TRACE_LEVEL_WARNING*)
- API (*CONFIG_SMP_TRACE_LEVEL_API*)

- EVENT (*CONFIG_SMP_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_SMP_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_SMP_TRACE_LEVEL_VERBOSE*)

CONFIG_BTIF_INITIAL_TRACE_LEVEL

BTIF layer

Define BT trace level for BTIF layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_BTIF_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_BTIF_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_BTIF_TRACE_LEVEL_WARNING*)
- API (*CONFIG_BTIF_TRACE_LEVEL_API*)
- EVENT (*CONFIG_BTIF_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_BTIF_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_BTIF_TRACE_LEVEL_VERBOSE*)

CONFIG_BTC_INITIAL_TRACE_LEVEL

BTC layer

Define BT trace level for BTC layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_BTC_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_BTC_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_BTC_TRACE_LEVEL_WARNING*)
- API (*CONFIG_BTC_TRACE_LEVEL_API*)
- EVENT (*CONFIG_BTC_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_BTC_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_BTC_TRACE_LEVEL_VERBOSE*)

CONFIG_OSI_INITIAL_TRACE_LEVEL

OSI layer

Define BT trace level for OSI layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_OSI_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_OSI_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_OSI_TRACE_LEVEL_WARNING*)
- API (*CONFIG_OSI_TRACE_LEVEL_API*)
- EVENT (*CONFIG_OSI_TRACE_LEVEL_EVENT*)

- DEBUG (*CONFIG_OSI_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_OSI_TRACE_LEVEL_VERBOSE*)

CONFIG_BLUFI_INITIAL_TRACE_LEVEL
BLUFI layer

Define BT trace level for BLUFI layer

Found in Component config > Bluetooth > Bluedroid Enable > BT DEBUG LOG LEVEL

Available options:

- NONE (*CONFIG_BLUFI_TRACE_LEVEL_NONE*)
- ERROR (*CONFIG_BLUFI_TRACE_LEVEL_ERROR*)
- WARNING (*CONFIG_BLUFI_TRACE_LEVEL_WARNING*)
- API (*CONFIG_BLUFI_TRACE_LEVEL_API*)
- EVENT (*CONFIG_BLUFI_TRACE_LEVEL_EVENT*)
- DEBUG (*CONFIG_BLUFI_TRACE_LEVEL_DEBUG*)
- VERBOSE (*CONFIG_BLUFI_TRACE_LEVEL_VERBOSE*)

CONFIG_BT_ACL_CONNECTIONS
BT/BLE MAX ACL CONNECTIONS(1~7)

Maximum BT/BLE connection count

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST
BT/BLE will first malloc the memory from the PSRAM

This select can save the internal RAM if there have the PSRAM

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY
Use dynamic memory allocation in BT/BLE stack

This select can make the allocation of memory will become more flexible

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_HOST_QUEUE_CONGESTION_CHECK
BLE queue congestion check

When scanning and scan duplicate is not enabled, if there are a lot of adv packets around or application layer handling adv packets is slow, it will cause the controller memory to run out. if enabled, adv packets will be lost when host queue is congested.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_SCAN_DUPLICATE
BLE Scan Duplicate Options

This select enables parameters setting of BLE scan duplicate.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_DUPLICATE_SCAN_CACHE_SIZE
Maximum number of devices in scan duplicate filter

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.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_MESH_SCAN_DUPLICATE_EN
Special duplicate scan mechanism for BLE Mesh scan

This enables the BLE scan duplicate for special BLE Mesh scan.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_MESH_DUPLICATE_SCAN_CACHE_SIZE
Maximum number of Mesh adv packets in scan duplicate filter

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.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_SUPPORTED
BLE adv report flow control supported

The function is mainly used to enable flow control for advertising reports. When it is enabled, advertising reports will be discarded by the controller if the number of unprocessed advertising reports exceeds the size of BLE adv report flow control.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_NUM
BLE adv report flow control number

The number of unprocessed advertising report that Bluedroid can save.If you set *BLE_ADV_REPORT_FLOW_CONTROL_NUM* to a small value, this may cause adv packets lost. If you set *BLE_ADV_REPORT_FLOW_CONTROL_NUM* to a large value, Bluedroid may cache a lot of adv packets and this may cause system memory run out. For example, if you set it to 50, the maximum memory consumed by host is 35 * 50 bytes. Please set *BLE_ADV_REPORT_FLOW_CONTROL_NUM* according to your system free memory and handle adv packets as fast as possible, otherwise it will cause adv packets lost.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_ADV_REPORT_DISCARD_THRSHOLD
BLE adv lost event threshold value

When adv report flow control is enabled, The ADV lost event will be generated when the number of ADV packets lost in the controller reaches this threshold. It is better to set a larger value. If you set *BLE_ADV_REPORT_DISCARD_THRSHOLD* to a small value or printf every adv lost event, it may cause adv packets lost more.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_ACTIVE_SCAN_REPORT_ADV_SCAN_RSP_INDIVIDUALLY
Report adv data and scan response individually when BLE active scan

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.

Found in Component config > Bluetooth > Bluedroid Enable

CONFIG_BLE_ESTABLISH_LINK_CONNECTION_TIMEOUT
Timeout of BLE connection establishment

Bluetooth Connection establishment maximum time, if connection time exceeds this value, the connection establishment fails, ESP_GATTC_OPEN_EVT or ESP_GATTS_OPEN_EVT is triggered.

Found in Component config > Bluetooth > Bluedroid Enable

libsodium

CONFIG_LIBSODIUM_USE_MBEDTLS_SHA

Use mbedTLS SHA256 & SHA512 implementations

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).

Found in Component config > libsodium

ESP HTTP client

CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS

Enable https

This option will enable https protocol by linking mbedtls library and initializing SSL transport

Found in Component config > ESP HTTP client

LWIP

CONFIG_L2_TO_L3_COPY

Enable copy between Layer2 and Layer3 packets

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.

Found in Component config > LWIP

CONFIG_LWIP_IRAM_OPTIMIZATION

Enable LWIP IRAM optimization

If this feature is enabled, some functions relating to RX/TX in LWIP will be put into IRAM, it can improve UDP/TCP throughput by >10% for single core mode, it doesn't help too much for dual core mode. On the other hand, it needs about 10KB IRAM for these optimizations.

If this feature is disabled, all lwip functions will be put into FLASH.

Found in Component config > LWIP

CONFIG_LWIP_MAX_SOCKETS

Max number of open sockets

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.

Found in Component config > LWIP

CONFIG_USE_ONLY_LWIP_SELECT
Support LWIP socket select() only

The virtual filesystem layer of select() redirects sockets to lwip_select() and non-socket file descriptors to their respective driver implementations. If this option is enabled then all calls of select() will be redirected to lwip_select(), therefore, select can be used for sockets only.

Found in Component config > LWIP

CONFIG_LWIP_SO_REUSE
Enable SO_REUSEADDR option

Enabling this option allows binding to a port which remains in TIME_WAIT.

Found in Component config > LWIP

CONFIG_LWIP_SO_REUSE_RXTOALL
SO_REUSEADDR copies broadcast/multicast to all matches

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.

Found in Component config > LWIP

CONFIG_LWIP_SO_RCVBUF
Enable SO_RCVBUF option

Enabling this option allows checking for available data on a netconn.

Found in Component config > LWIP

CONFIG_LWIP_DHCP_MAX_NTP_SERVERS
Maximum number of NTP servers

Set maximum number of NTP servers used by LwIP SNTP module. First argument of sntp_setserver/sntp_setservername functions is limited to this value.

Found in Component config > LWIP

CONFIG_LWIP_IP_FRAG
Enable fragment outgoing IP packets

Enabling this option allows fragmenting outgoing IP packets if their size exceeds MTU.

Found in Component config > LWIP

CONFIG_LWIP_IP_REASSEMBLY
Enable reassembly incoming fragmented IP packets

Enabling this option allows reassemblying incoming fragmented IP packets.

Found in Component config > LWIP

CONFIG_LWIP_STATS
Enable LWIP statistics

Enabling this option allows LWIP statistics

Found in Component config > LWIP

CONFIG_LWIP_ETHARP_TRUST_IP_MAC
Enable LWIP ARP trust

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.

Found in Component config > LWIP

CONFIG_ESP_GRATUITOUS_ARP
Send gratuitous ARP periodically

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.

Found in Component config > LWIP

CONFIG_GARP_TMR_INTERVAL
GARP timer interval(seconds)

Set the timer interval for gratuitous ARP. The default value is 60s

Found in Component config > LWIP

CONFIG_TCPIP_RECVMBOX_SIZE
TCPIP task receive mail box size

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.

Found in Component config > LWIP

CONFIG_LWIP_DHCP_DOES_ARP_CHECK
DHCP: Perform ARP check on any offered address

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.

Found in Component config > LWIP

DHCP server

- *CONFIG_LWIP_DHCPS_LEASE_UNIT*

- *CONFIG_LWIP_DHCPS_MAX_STATION_NUM*

CONFIG_LWIP_DHCPS_LEASE_UNIT
Multiplier for lease time, in seconds

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.

Found in Component config > LWIP > DHCP server

CONFIG_LWIP_DHCPS_MAX_STATION_NUM
Maximum number of stations

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.

Found in Component config > LWIP > DHCP server

CONFIG_LWIP_AUTOIP

Enable IPV4 Link-Local Addressing (AUTOIP)

- *CONFIG_LWIP_AUTOIP_TRIES*
- *CONFIG_LWIP_AUTOIP_MAX_CONFLICTS*
- *CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL*

Enable IPV4 Link-Local Addressing (AUTOIP)

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.

CONFIG_LWIP_AUTOIP_TRIES
DHCP Probes before self-assigning IPv4 LL address

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.)

Found in Component config > LWIP > Enable IPV4 Link-Local Addressing (AUTOIP)

CONFIG_LWIP_AUTOIP_MAX_CONFLICTS
Max IP conflicts before rate limiting

If the AUTOIP functionality detects this many IP conflicts while self-assigning an address, it will go into a rate limited mode.

Found in Component config > LWIP > Enable IPV4 Link-Local Addressing (AUTOIP)

CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL
Rate limited interval (seconds)

If rate limiting self-assignment requests, wait this long between each request.

Found in Component config > LWIP > Enable IPV4 Link-Local Addressing (AUTOIP)

CONFIG_LWIP_IPV6_AUTOCONFIG
Enable IPV6 stateless address autoconfiguration

Enabling this option allows the devices to IPV6 stateless address autoconfiguration.

See RFC 4862.

Found in Component config > LWIP

CONFIG_LWIP_NETIF_LOOPBACK

Support per-interface loopback

- *CONFIG_LWIP_LOOPBACK_MAX_PBUFS*

  Support per-interface loopback

  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.

CONFIG_LWIP_LOOPBACK_MAX_PBUFS

Max queued loopback packets per interface

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.

Found in Component config > LWIP > Support per-interface loopback

TCP

- *CONFIG_LWIP_MAX_ACTIVE_TCP*
- *CONFIG_LWIP_MAX_LISTENING_TCP*
- *CONFIG_TCP_MAXRTX*
- *CONFIG_TCP_SYNMAXRTX*
- *CONFIG_TCP_MSS*
- *CONFIG_TCP_MSL*
- *CONFIG_TCP_SND_BUF_DEFAULT*
- *CONFIG_TCP_WND_DEFAULT*
- *CONFIG_TCP_RECVMBOX_SIZE*
- *CONFIG_TCP_QUEUE_OOSEQ*
- *CONFIG_ESP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES*
- *CONFIG_TCP_OVERSIZE*
- *CONFIG_LWIP_WND_SCALE*
- *CONFIG_TCP_RCV_SCALE*

CONFIG_LWIP_MAX_ACTIVE_TCP

Maximum active TCP Connections

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.

Found in Component config > LWIP > TCP

CONFIG_LWIP_MAX_LISTENING_TCP
Maximum listening TCP Connections

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.

Found in Component config > LWIP > TCP

CONFIG_TCP_MAXRTX
Maximum number of retransmissions of data segments

Set maximum number of retransmissions of data segments.

Found in Component config > LWIP > TCP

CONFIG_TCP_SYNMAXRTX
Maximum number of retransmissions of SYN segments

Set maximum number of retransmissions of SYN segments.

Found in Component config > LWIP > TCP

CONFIG_TCP_MSS
Maximum Segment Size (MSS)

Set maximum segment size for TCP transmission.

Can be set lower to save RAM, the default value 1436 will give best throughput.

Found in Component config > LWIP > TCP

CONFIG_TCP_MSL
Maximum segment lifetime (MSL)

Set maximum segment lifetime in in milliseconds.

Found in Component config > LWIP > TCP

CONFIG_TCP_SND_BUF_DEFAULT
Default send buffer size

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.

Found in Component config > LWIP > TCP

CONFIG_TCP_WND_DEFAULT
Default receive window size

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.

Found in Component config > LWIP > TCP

CONFIG_TCP_RECVMBOX_SIZE
Default TCP receive mail box size

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 caches maximum TCP_RECCVMBOX_SIZE packets for each TCP socket, so the maximum possible cached TCP packets for all TCP sockets is TCP_RECCVMBOX_SIZE multiples the maximum TCP socket number. In other words, the bigger TCP_RECVMBOX_SIZE means more memory. On the other hand, if the receiv 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.

Found in Component config > LWIP > TCP

CONFIG_TCP_QUEUE_OOSEQ

Queue incoming out-of-order segments

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.

Found in Component config > LWIP > TCP

CONFIG_ESP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES

Keep TCP connections when IP changed

This option is enabled when the following scenario happen: network dropped and reconnected, IP changes is like: 192.168.0.2->0.0.0.0->192.168.0.2

Disable this option to keep consistent with the original LWIP code behavior.

Found in Component config > LWIP > TCP

CONFIG_TCP_OVERSIZE

Pre-allocate transmit PBUF size

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.

Found in Component config > LWIP > TCP

Available options:

- MSS (*CONFIG_TCP_OVERSIZE_MSS*)
- 25% MSS (*CONFIG_TCP_OVERSIZE_QUARTER_MSS*)
- Disabled (*CONFIG_TCP_OVERSIZE_DISABLE*)

CONFIG_LWIP_WND_SCALE

Support TCP window scale

Enable this feature to support TCP window scaling.

Found in Component config > LWIP > TCP

CONFIG_TCP_RCV_SCALE
Set TCP receiving window scaling factor

Enable this feature to support TCP window scaling.

Found in Component config > LWIP > TCP

UDP

- *CONFIG_LWIP_MAX_UDP_PCBS*
- *CONFIG_UDP_RECVMBOX_SIZE*

CONFIG_LWIP_MAX_UDP_PCBS
Maximum active UDP control blocks

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.

Found in Component config > LWIP > UDP

CONFIG_UDP_RECVMBOX_SIZE
Default UDP receive mail box size

Set UDP receive mail box size. The recommended value is 6.

UDP receive mail box is a per socket mail box, when the application receives packets from UDP socket, LWIP core firstly posts the packets to UDP receive mail box and the application then fetches the packets from mail box. It means LWIP can caches maximum UDP_RECCVMBOX_SIZE packets for each UDP socket, so the maximum possible cached UDP packets for all UDP sockets is UDP_RECCVMBOX_SIZE multiples the maximum UDP socket number. In other words, the bigger UDP_RECVMBOX_SIZE means more memory. On the other hand, if the receiv 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.

Found in Component config > LWIP > UDP

CONFIG_TCPIP_TASK_STACK_SIZE
TCP/IP Task Stack Size

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.

Found in Component config > LWIP

CONFIG_PPP_SUPPORT

Enable PPP support (new/experimental)

- *CONFIG_PPP_PAP_SUPPORT*
- *CONFIG_PPP_CHAP_SUPPORT*
- *CONFIG_PPP_MSCHAP_SUPPORT*
- *CONFIG_PPP_MPPE_SUPPORT*
- *CONFIG_PPP_DEBUG_ON*

Enable PPP support (new/experimental)

Enable PPP stack. Now only PPP over serial is possible.

PPP over serial support is experimental and unsupported.

CONFIG_PPP_PAP_SUPPORT
Enable PAP support

Enable Password Authentication Protocol (PAP) support

Found in Component config > LWIP > Enable PPP support (new/experimental)

CONFIG_PPP_CHAP_SUPPORT
Enable CHAP support

Enable Challenge Handshake Authentication Protocol (CHAP) support

Found in Component config > LWIP > Enable PPP support (new/experimental)

CONFIG_PPP_MSCHAP_SUPPORT
Enable MSCHAP support

Enable Microsoft version of the Challenge-Handshake Authentication Protocol (MSCHAP) support

Found in Component config > LWIP > Enable PPP support (new/experimental)

CONFIG_PPP_MPPE_SUPPORT
Enable MPPE support

Enable Microsoft Point-to-Point Encryption (MPPE) support

Found in Component config > LWIP > Enable PPP support (new/experimental)

CONFIG_PPP_DEBUG_ON
Enable PPP debug log output

Enable PPP debug log output

Found in Component config > LWIP > Enable PPP support (new/experimental)

ICMP

- *CONFIG_LWIP_MULTICAST_PING*
- *CONFIG_LWIP_BROADCAST_PING*

CONFIG_LWIP_MULTICAST_PING
Respond to multicast pings

Found in Component config > LWIP > ICMP

CONFIG_LWIP_BROADCAST_PING
Respond to broadcast pings

Found in Component config > LWIP > ICMP

LWIP RAW API

- *CONFIG_LWIP_MAX_RAW_PCBS*

CONFIG_LWIP_MAX_RAW_PCBS
Maximum LWIP RAW PCBs

The maximum number of simultaneously active LWIP RAW protocol control blocks. The practical maximum limit is determined by available heap memory at runtime.

Found in Component config > LWIP > LWIP RAW API

Application Level Tracing

CONFIG_ESP32_APPTRACE_DESTINATION
Data Destination

Select destination for application trace: trace memory or none (to disable).

Found in Component config > Application Level Tracing

Available options:

- Trace memory (*CONFIG_ESP32_APPTRACE_DEST_TRAX*)
- None (*CONFIG_ESP32_APPTRACE_DEST_NONE*)

CONFIG_ESP32_APPTRACE_ONPANIC_HOST_FLUSH_TMO
Timeout for flushing last trace data to host on panic

Timeout for flushing last trace data to host in case of panic. In ms. Use -1 to disable timeout and wait forever.

Found in Component config > Application Level Tracing

CONFIG_ESP32_APPTRACE_POSTMORTEM_FLUSH_TRAX_THRESH
Threshold for flushing last trace data to host on panic

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.

Found in Component config > Application Level Tracing

CONFIG_ESP32_APPTRACE_PENDING_DATA_SIZE_MAX
Size of the pending data buffer

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.

Found in Component config > Application Level Tracing

FreeRTOS SystemView Tracing

- *CONFIG_SYSVIEW_ENABLE*
- *CONFIG_SYSVIEW_TS_SOURCE*
- *CONFIG_SYSVIEW_EVT_OVERFLOW_ENABLE*
- *CONFIG_SYSVIEW_EVT_ISR_ENTER_ENABLE*
- *CONFIG_SYSVIEW_EVT_ISR_EXIT_ENABLE*
- *CONFIG_SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_START_EXEC_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE*

- *CONFIG_SYSVIEW_EVT_TASK_START_READY_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_STOP_READY_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_CREATE_ENABLE*
- *CONFIG_SYSVIEW_EVT_TASK_TERMINATE_ENABLE*
- *CONFIG_SYSVIEW_EVT_IDLE_ENABLE*
- *CONFIG_SYSVIEW_EVT_TIMER_ENTER_ENABLE*
- *CONFIG_SYSVIEW_EVT_TIMER_EXIT_ENABLE*

CONFIG_SYSVIEW_ENABLE

SystemView Tracing Enable

Enables supporrt for SEGGER SystemView tracing functionality.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_TS_SOURCE

Timer to use as timestamp source

SystemView needs to use a hardware timer as the source of timestamps when tracing. This option selects the timer for it.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Available options:

- CPU cycle counter (CCOUNT) (*CONFIG_SYSVIEW_TS_SOURCE_CCOUNT*)
- Timer 0, Group 0 (*CONFIG_SYSVIEW_TS_SOURCE_TIMER_00*)
- Timer 1, Group 0 (*CONFIG_SYSVIEW_TS_SOURCE_TIMER_01*)
- Timer 0, Group 1 (*CONFIG_SYSVIEW_TS_SOURCE_TIMER_10*)
- Timer 1, Group 1 (*CONFIG_SYSVIEW_TS_SOURCE_TIMER_11*)
- esp_timer high resolution timer (*CONFIG_SYSVIEW_TS_SOURCE_ESP_TIMER*)

CONFIG_SYSVIEW_EVT_OVERFLOW_ENABLE

Trace Buffer Overflow Event

Enables "Trace Buffer Overflow" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_ISR_ENTER_ENABLE

ISR Enter Event

Enables "ISR Enter" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_ISR_EXIT_ENABLE

ISR Exit Event

Enables "ISR Exit" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE

ISR Exit to Scheduler Event

Enables "ISR to Scheduler" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_TASK_START_EXEC_ENABLE
Task Start Execution Event

Enables "Task Start Execution" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE
Task Stop Execution Event

Enables "Task Stop Execution" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_TASK_START_READY_ENABLE
Task Start Ready State Event

Enables "Task Start Ready State" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_TASK_STOP_READY_ENABLE
Task Stop Ready State Event

Enables "Task Stop Ready State" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_TASK_CREATE_ENABLE
Task Create Event

Enables "Task Create" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_TASK_TERMINATE_ENABLE
Task Terminate Event

Enables "Task Terminate" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_IDLE_ENABLE
System Idle Event

Enables "System Idle" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_TIMER_ENTER_ENABLE
Timer Enter Event

Enables "Timer Enter" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_SYSVIEW_EVT_TIMER_EXIT_ENABLE
Timer Exit Event

Enables "Timer Exit" event.

Found in Component config > Application Level Tracing > FreeRTOS SystemView Tracing

CONFIG_ESP32_GCOV_ENABLE
GCOV to Host Enable

Enables support for GCOV data transfer to host.

Found in Component config > Application Level Tracing

ESP32-specific

CONFIG_ESP32_REV_MIN

Minimum Supported ESP32 Revision

Minimum revision that ESP-IDF would support. ESP-IDF performs different strategy on different esp32 revision.

Found in Component config > ESP32-specific

Available options:

- Rev 0 (*CONFIG_ESP32_REV_MIN_0*)
- Rev 1 (*CONFIG_ESP32_REV_MIN_1*)
- Rev 2 (*CONFIG_ESP32_REV_MIN_2*)
- Rev 3 (*CONFIG_ESP32_REV_MIN_3*)

CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ

CPU frequency

CPU frequency to be set on application startup.

Found in Component config > ESP32-specific

Available options:

- 80 MHz (*CONFIG_ESP32_DEFAULT_CPU_FREQ_80*)
- 160 MHz (*CONFIG_ESP32_DEFAULT_CPU_FREQ_160*)
- 240 MHz (*CONFIG_ESP32_DEFAULT_CPU_FREQ_240*)

CONFIG_SPIRAM_SUPPORT

Support for external, SPI-connected RAM

This enables support for an external SPI RAM chip, connected in parallel with the main SPI flash chip.

Found in Component config > ESP32-specific

SPI RAM config

- *CONFIG_SPIRAM_BOOT_INIT*
- *CONFIG_SPIRAM_IGNORE_NOTFOUND*
- *CONFIG_SPIRAM_USE*
- *CONFIG_SPIRAM_TYPE*
- CONFIG_SPIRAM_SIZE
- *CONFIG_SPIRAM_SPEED*
- *CONFIG_SPIRAM_MEMTEST*
- *CONFIG_SPIRAM_CACHE_WORKAROUND*
- *CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL*

- *CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST*
- *CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL*
- *CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY*

CONFIG_SPIRAM_BOOT_INIT

Initialize SPI RAM when booting the ESP32

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.

Found in Component config > ESP32-specific > SPI RAM config

CONFIG_SPIRAM_IGNORE_NOTFOUND

Ignore PSRAM when not found

Normally, if psram initialization is enabled during compile time but not found at runtime, it is seen as an error making the ESP32 panic. If this is enabled, the ESP32 will keep on running but will not add the (non-existing) RAM to any allocator.

Found in Component config > ESP32-specific > SPI RAM config

CONFIG_SPIRAM_USE

SPI RAM access method

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.

Found in Component config > ESP32-specific > SPI RAM config

Available options:

- Integrate RAM into ESP32 memory map (*CONFIG_SPIRAM_USE_MEMMAP*)
- Make RAM allocatable using heap_caps_malloc(..., MALLOC_CAP_SPIRAM) (*CONFIG_SPIRAM_USE_CAPS_ALLOC*)
- Make RAM allocatable using malloc() as well (*CONFIG_SPIRAM_USE_MALLOC*)

CONFIG_SPIRAM_TYPE

Type of SPI RAM chip in use

Found in Component config > ESP32-specific > SPI RAM config

Available options:

- ESP-PSRAM32 or IS25WP032 (*CONFIG_SPIRAM_TYPE_ESPPSRAM32*)

CONFIG_SPIRAM_SPEED

Set RAM clock speed

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)

Found in Component config > ESP32-specific > SPI RAM config

Available options:

- 40MHz clock speed (*CONFIG_SPIRAM_SPEED_40M*)
- 80MHz clock speed (*CONFIG_SPIRAM_SPEED_80M*)

CONFIG_SPIRAM_MEMTEST

Run memory test on SPI RAM initialization

Runs a rudimentary memory test on initialization. Aborts when memory test fails. Disable this for slightly faster startop.

Found in Component config > ESP32-specific > SPI RAM config

CONFIG_SPIRAM_CACHE_WORKAROUND

Enable workaround for bug in SPI RAM cache for Rev1 ESP32s

Revision 1 of the ESP32 has a bug that can cause a write to PSRAM not to take place in some situations when the cache line needs to be fetched from external RAM and an interrupt occurs. This enables a fix in the compiler (-mfix-esp32-psram-cache-issue) that makes sure the specific code that is vulnerable to this will not be emitted.

This will also not use any bits of newlib that are located in ROM, opting for a version that is compiled with the workaround and located in flash instead.

The workaround is not required for ESP32 revision 3 and above.

Found in Component config > ESP32-specific > SPI RAM config

CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL

Maximum malloc() size, in bytes, to always put in internal memory

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.

Found in Component config > ESP32-specific > SPI RAM config

CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST

Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, allocate internal memory

Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, try to allocate internal memory then.

Found in Component config > ESP32-specific > SPI RAM config

CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL

Reserve this amount of bytes for data that specifically needs to be in DMA or internal memory

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.

Found in Component config > ESP32-specific > SPI RAM config

CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY
Allow external memory as an argument to xTaskCreateStatic

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.

Found in Component config > ESP32-specific > SPI RAM config

CONFIG_ESP32_TRAX
Use TRAX tracing feature

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.

Found in Component config > ESP32-specific

CONFIG_ESP32_TRAX_TWOBANKS
Reserve memory for tracing both pro as well as app cpu execution

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.

Found in Component config > ESP32-specific

CONFIG_ESP32_COREDUMP_TO_FLASH_OR_UART
Core dump destination

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.

Found in Component config > ESP32-specific

Available options:

- Flash (*CONFIG_ESP32_ENABLE_COREDUMP_TO_FLASH*)
- UART (*CONFIG_ESP32_ENABLE_COREDUMP_TO_UART*)
- None (*CONFIG_ESP32_ENABLE_COREDUMP_TO_NONE*)

CONFIG_ESP32_CORE_DUMP_UART_DELAY
Core dump print to UART delay

Config delay (in ms) before printing core dump to UART. Delay can be interrupted by pressing Enter key.

Found in Component config > ESP32-specific

CONFIG_ESP32_CORE_DUMP_LOG_LEVEL
Core dump module logging level

Config core dump module logging level (0-5).

Found in Component config > ESP32-specific

CONFIG_NUMBER_OF_UNIVERSAL_MAC_ADDRESS
Number of universally administered (by IEEE) MAC address

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.)

Found in Component config > ESP32-specific

Available options:

- Two (*CONFIG_TWO_UNIVERSAL_MAC_ADDRESS*)
- Four (*CONFIG_FOUR_UNIVERSAL_MAC_ADDRESS*)

CONFIG_SYSTEM_EVENT_QUEUE_SIZE

System event queue size

Config system event queue size in different application.

Found in Component config > ESP32-specific

CONFIG_SYSTEM_EVENT_TASK_STACK_SIZE

Event loop task stack size

Config system event task stack size in different application.

Found in Component config > ESP32-specific

CONFIG_MAIN_TASK_STACK_SIZE

Main task stack size

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.

Found in Component config > ESP32-specific

CONFIG_IPC_TASK_STACK_SIZE

Inter-Processor Call (IPC) task stack size

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.

Found in Component config > ESP32-specific

CONFIG_TIMER_TASK_STACK_SIZE

High-resolution timer task stack size

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 seing 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.

Found in Component config > ESP32-specific

CONFIG_NEWLIB_STDOUT_LINE_ENDING

Line ending for UART output

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 occurence of LF is replaced with CR

This option doesn't affect behavior of the UART driver (drivers/uart.h).

Found in Component config > ESP32-specific

Available options:

- CRLF (*CONFIG_NEWLIB_STDOUT_LINE_ENDING_CRLF*)
- LF (*CONFIG_NEWLIB_STDOUT_LINE_ENDING_LF*)
- CR (*CONFIG_NEWLIB_STDOUT_LINE_ENDING_CR*)

CONFIG_NEWLIB_STDIN_LINE_ENDING

Line ending for UART input

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 occurence of CR is replaced with LF

This option doesn't affect behavior of the UART driver (drivers/uart.h).

Found in Component config > ESP32-specific

Available options:

- CRLF (*CONFIG_NEWLIB_STDIN_LINE_ENDING_CRLF*)
- LF (*CONFIG_NEWLIB_STDIN_LINE_ENDING_LF*)
- CR (*CONFIG_NEWLIB_STDIN_LINE_ENDING_CR*)

CONFIG_NEWLIB_NANO_FORMAT

Enable 'nano' formatting options for printf/scanf family

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.

Found in Component config > ESP32-specific

CONFIG_CONSOLE_UART
UART for console output

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.

Found in Component config > ESP32-specific

Available options:

- Default: UART0, TX=GPIO1, RX=GPIO3 (*CONFIG_CONSOLE_UART_DEFAULT*)
- Custom (*CONFIG_CONSOLE_UART_CUSTOM*)
- None (*CONFIG_CONSOLE_UART_NONE*)

CONFIG_CONSOLE_UART_NUM
UART peripheral to use for console output (0-1)

Due of a ROM bug, UART2 is not supported for console output via ets_printf.

Found in Component config > ESP32-specific

Available options:

- UART0 (*CONFIG_CONSOLE_UART_CUSTOM_NUM_0*)
- UART1 (*CONFIG_CONSOLE_UART_CUSTOM_NUM_1*)

CONFIG_CONSOLE_UART_TX_GPIO
UART TX on GPIO#

Found in Component config > ESP32-specific

CONFIG_CONSOLE_UART_RX_GPIO
UART RX on GPIO#

Found in Component config > ESP32-specific

CONFIG_CONSOLE_UART_BAUDRATE
UART console baud rate

Found in Component config > ESP32-specific

CONFIG_ULP_COPROC_ENABLED
Enable Ultra Low Power (ULP) Coprocessor

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.

Found in Component config > ESP32-specific

CONFIG_ULP_COPROC_RESERVE_MEM
RTC slow memory reserved for coprocessor

Bytes of memory to reserve for ULP coprocessor firmware & data.

Data is reserved at the beginning of RTC slow memory.

Found in Component config > ESP32-specific

CONFIG_ESP32_PANIC
Panic handler behaviour

If FreeRTOS detects unexpected behaviour or an unhandled exception, the panic handler is invoked. Configure the panic handlers action here.

Found in Component config > ESP32-specific

Available options:

- Print registers and halt (*CONFIG_ESP32_PANIC_PRINT_HALT*)

  Outputs the relevant registers over the serial port and halt the processor. Needs a manual reset to restart.

- Print registers and reboot (*CONFIG_ESP32_PANIC_PRINT_REBOOT*)

  Outputs the relevant registers over the serial port and immediately reset the processor.

- Silent reboot (*CONFIG_ESP32_PANIC_SILENT_REBOOT*)

  Just resets the processor without outputting anything

- Invoke GDBStub (*CONFIG_ESP32_PANIC_GDBSTUB*)

  Invoke gdbstub on the serial port, allowing for gdb to attach to it to do a postmortem of the crash.

CONFIG_ESP32_DEBUG_OCDAWARE
Make exception and panic handlers JTAG/OCD aware

The FreeRTOS panic and unhandled exception handers can detect a JTAG OCD debugger and instead of panicking, have the debugger stop on the offending instruction.

Found in Component config > ESP32-specific

CONFIG_ESP32_DEBUG_STUBS_ENABLE
OpenOCD debug stubs

Debug stubs are used by OpenOCD to execute pre-compiled onboard code which does some useful debugging, e.g. GCOV data dump.

Found in Component config > ESP32-specific

CONFIG_INT_WDT
Interrupt watchdog

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.

Found in Component config > ESP32-specific

CONFIG_INT_WDT_TIMEOUT_MS
Interrupt watchdog timeout (ms)

The timeout of the watchdog, in miliseconds. Make this higher than the FreeRTOS tick rate.

Found in Component config > ESP32-specific

CONFIG_INT_WDT_CHECK_CPU1
Also watch CPU1 tick interrupt

Also detect if interrupts on CPU 1 are disabled for too long.

Found in Component config > ESP32-specific

CONFIG_TASK_WDT
Initialize Task Watchdog Timer on startup

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)

Found in Component config > ESP32-specific

CONFIG_TASK_WDT_PANIC
Invoke panic handler on Task Watchdog timeout

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)

Found in Component config > ESP32-specific

CONFIG_TASK_WDT_TIMEOUT_S
Task Watchdog timeout period (seconds)

Timeout period configuration for the Task Watchdog Timer in seconds. This is also configurable at run time (see Task Watchdog Timer API Reference)

Found in Component config > ESP32-specific

CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU0
Watch CPU0 Idle Task

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.

Found in Component config > ESP32-specific

CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU1
Watch CPU1 Idle Task

If this option is enabled, the Task Wtachdog Timer will wach the CPU1 Idle Task.

Found in Component config > ESP32-specific

CONFIG_BROWNOUT_DET
Hardware brownout detect & reset

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.

Found in Component config > ESP32-specific

CONFIG_BROWNOUT_DET_LVL_SEL
Brownout voltage level

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.

Found in Component config > ESP32-specific

Available options:

- 2.43V +/- 0.05 (*CONFIG_BROWNOUT_DET_LVL_SEL_0*)
- 2.48V +/- 0.05 (*CONFIG_BROWNOUT_DET_LVL_SEL_1*)
- 2.58V +/- 0.05 (*CONFIG_BROWNOUT_DET_LVL_SEL_2*)

- 2.62V +/- 0.05 (*CONFIG_BROWNOUT_DET_LVL_SEL_3*)
- 2.67V +/- 0.05 (*CONFIG_BROWNOUT_DET_LVL_SEL_4*)
- 2.70V +/- 0.05 (*CONFIG_BROWNOUT_DET_LVL_SEL_5*)
- 2.77V +/- 0.05 (*CONFIG_BROWNOUT_DET_LVL_SEL_6*)
- 2.80V +/- 0.05 (*CONFIG_BROWNOUT_DET_LVL_SEL_7*)

CONFIG_ESP32_TIME_SYSCALL

Timers used for gettimeofday function

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.

Found in Component config > ESP32-specific

Available options:

- RTC and high-resolution timer (*CONFIG_ESP32_TIME_SYSCALL_USE_RTC_FRC1*)
- RTC (*CONFIG_ESP32_TIME_SYSCALL_USE_RTC*)
- High-resolution timer (*CONFIG_ESP32_TIME_SYSCALL_USE_FRC1*)
- None (*CONFIG_ESP32_TIME_SYSCALL_USE_NONE*)

CONFIG_ESP32_RTC_CLOCK_SOURCE

RTC clock source

Choose which clock is used as RTC clock source.

Found in Component config > ESP32-specific

Available options:

- Internal 150kHz RC oscillator (*CONFIG_ESP32_RTC_CLOCK_SOURCE_INTERNAL_RC*)
- External 32kHz crystal (*CONFIG_ESP32_RTC_CLOCK_SOURCE_EXTERNAL_CRYSTAL*)

CONFIG_ESP32_RTC_EXTERNAL_CRYSTAL_ADDITIONAL_CURRENT

Additional current for external 32kHz crystal

Choose which additional current is used for rtc external crystal.

- With some 32kHz crystal configurations, the X32N and X32P pins may not have enough drive strength to keep the crystal oscillating during deep sleep. If this option is enabled, additional current from touchpad 9 is provided internally to drive the 32kHz crystal. If this option is enabled, deep sleep current is slightly higher (4-5uA) and the touchpad and ULP wakeup sources are not available.

Found in Component config > ESP32-specific

CONFIG_ESP32_RTC_CLK_CAL_CYCLES

Number of cycles for RTC_SLOW_CLK calibration

When the startup code initializes RTC_SLOW_CLK, it can perform calibration by comparing the RTC_SLOW_CLK frequency with main XTAL frequency. This option sets the number of RTC_SLOW_CLK cycles measured by the calibration routine. Higher numbers increase calibration precision, which may be important for applications which spend a lot of time in deep sleep. Lower numbers reduce startup time.

When this option is set to 0, clock calibration will not be performed at startup, and approximate clock frequencies will be assumed:

- 150000 Hz if internal RC oscillator is used as clock source. For this use value 1024.
- 32768 Hz if the 32k crystal oscillator is used. For this use value 3000 or more. In case more value will help improve the definition of the launch of the crystal. If the crystal could not start, it will be switched to internal RC.

Found in Component config > ESP32-specific

CONFIG_ESP32_RTC_XTAL_BOOTSTRAP_CYCLES

Bootstrap cycles for external 32kHz crystal

To reduce the startup time of an external RTC crystal, we bootstrap it with a 32kHz square wave for a fixed number of cycles. Setting 0 will disable bootstrapping (if disabled, the crystal may take longer to start up or fail to oscillate under some conditions).

If this value is too high, a faulty crystal may initially start and then fail. If this value is too low, an otherwise good crystal may not start.

To accurately determine if the crystal has started, set a larger "Number of cycles for RTC_SLOW_CLK calibration" (about 3000).

Found in Component config > ESP32-specific

CONFIG_ESP32_DEEP_SLEEP_WAKEUP_DELAY

Extra delay in deep sleep wake stub (in us)

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.

Found in Component config > ESP32-specific

CONFIG_ESP32_XTAL_FREQ_SEL

Main XTAL frequency

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.

Found in Component config > ESP32-specific

Available options:

- 40 MHz (*CONFIG_ESP32_XTAL_FREQ_40*)
- 26 MHz (*CONFIG_ESP32_XTAL_FREQ_26*)
- Autodetect (*CONFIG_ESP32_XTAL_FREQ_AUTO*)

CONFIG_DISABLE_BASIC_ROM_CONSOLE

Permanently disable BASIC ROM Console

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.)

Found in Component config > ESP32-specific

CONFIG_NO_BLOBS

No Binary Blobs

If enabled, this disables the linking of binary libraries in the application build. Note that after enabling this Wi-Fi/Bluetooth will not work.

Found in Component config > ESP32-specific

CONFIG_ESP_TIMER_PROFILING

Enable esp_timer profiling features

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.

Found in Component config > ESP32-specific

CONFIG_COMPATIBLE_PRE_V2_1_BOOTLOADERS

App compatible with bootloaders before IDF v2.1

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.

Found in Component config > ESP32-specific

CONFIG_ESP_ERR_TO_NAME_LOOKUP

Enable lookup of error code strings

Functions esp_err_to_name() and esp_err_to_name_r() return string representations of error codes from a pre-generated lookup table. This option can be used to turn off the use of the look-up table in order to save memory but this comes at the price of sacrificing distinguishable (meaningful) output string representations.

Found in Component config > ESP32-specific

Wi-Fi

CONFIG_SW_COEXIST_ENABLE

Software controls WiFi/Bluetooth coexistence

If enabled, WiFi & Bluetooth coexistence is controlled by software rather than hardware. Recommended for heavy traffic scenarios. Both coexistence configuration options are automatically managed, no user intervention is required.

Found in Component config > Wi-Fi

CONFIG_SW_COEXIST_PREFERENCE

WiFi/Bluetooth coexistence performance preference

Choose Bluetooth/WiFi/Balance for different preference. If choose WiFi, it will make WiFi performance better. Such, keep WiFi Audio more fluent. If choose Bluetooth, it will make Bluetooth performance better. Such, keep Bluetooth(A2DP) Audio more fluent. If choose Balance, the performance of WiFi and bluetooth will be balance. It's default. Normally, just choose balance, the A2DP audio can play fluently, too. Except config preference in menuconfig, you can also call esp_coex_preference_set() dynamically.

Found in Component config > Wi-Fi

Available options:

- WiFi (*CONFIG_SW_COEXIST_PREFERENCE_WIFI*)
- Bluetooth(include BR/EDR and BLE) (*CONFIG_SW_COEXIST_PREFERENCE_BT*)
- Balance (*CONFIG_SW_COEXIST_PREFERENCE_BALANCE*)

CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM

Max number of WiFi static RX buffers

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.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM

Max number of WiFi dynamic RX buffers

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.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_TX_BUFFER

Type of WiFi TX buffers

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.

Found in Component config > Wi-Fi

Available options:

- Static (*CONFIG_ESP32_WIFI_STATIC_TX_BUFFER*)
- Dynamic (*CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER*)

CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM

Max number of WiFi static TX buffers

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.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM

Max number of WiFi dynamic TX buffers

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.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_CSI_ENABLED

WiFi CSI(Channel State Information)

Select this option to enable CSI(Channel State Information) feature. CSI takes about CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM KB of RAM. If CSI is not used, it is better to disable this feature in order to save memory.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED

WiFi AMPDU TX

Select this option to enable AMPDU TX feature

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_TX_BA_WIN

WiFi AMPDU TX BA window size

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.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED
WiFi AMPDU RX

Select this option to enable AMPDU RX feature

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_RX_BA_WIN
WiFi AMPDU RX BA window size

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 prefered to allocat in PSRAM first, the default and minimum value should be 16 to achieve better throughput and compatibility with both stations and APs.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_NVS_ENABLED
WiFi NVS flash

Select this option to enable WiFi NVS flash

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_TASK_CORE_ID
WiFi Task Core ID

Pinned WiFi task to core 0 or core 1.

Found in Component config > Wi-Fi

Available options:

- Core 0 (*CONFIG_ESP32_WIFI_TASK_PINNED_TO_CORE_0*)
- Core 1 (*CONFIG_ESP32_WIFI_TASK_PINNED_TO_CORE_1*)

CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN
Max length of WiFi SoftAP Beacon

ESP-MESH utilizes beacon frames to detect and resolve root node conflicts (see documentation). However the default length of a beacon frame can simultaneously hold only five root node identifier structures, meaning that a root node conflict of up to five nodes can be detected at one time. In the occurence of more root nodes conflict involving more than five root nodes, the conflict resolution process will detect five of the root nodes, resolve the conflict, and re-detect more root nodes. This process will repeat until all root node conflicts are resolved. However this process can generally take a very long time.

To counter this situation, the beacon frame length can be increased such that more root nodes can be detected simultaneously. Each additional root node will require 36 bytes and should be added ontop of the default beacon frame length of 752 bytes. For example, if you want to detect 10 root nodes simultaneously, you need to set the beacon frame length as 932 (752+36*5).

Setting a longer beacon length also assists with debugging as the conflicting root nodes can be identified more quickly.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_IRAM_OPT
WiFi IRAM speed optimization

Select this option to place frequently called Wi-Fi library functions in IRAM. When this option is disabled, more than 10Kbytes of IRAM memory will be saved but Wi-Fi throughput will be reduced.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_RX_IRAM_OPT
WiFi RX IRAM speed optimization

Select this option to place frequently called Wi-Fi library RX functions in IRAM.When this option is disabled, more than 17Kbytes of IRAM memory will be saved but Wi-Fi performance will be reduced.

Found in Component config > Wi-Fi

CONFIG_ESP32_WIFI_MGMT_SBUF_NUM
WiFi mgmt short buffer number

Set the number of WiFi management short buffer.

Found in Component config > Wi-Fi

PHY

CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE
Store phy calibration data in NVS

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'.

Found in Component config > PHY

CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION
Use a partition to store PHY init data

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'.

Found in Component config > PHY

CONFIG_ESP32_PHY_MAX_WIFI_TX_POWER
Max WiFi TX power (dBm)

Set maximum transmit power for WiFi radio. Actual transmit power for high data rates may be lower than this setting.

Found in Component config > PHY

CONFIG_ESP32_DPORT_DIS_INTERRUPT_LVL
Disable the interrupt level for the DPORT workarounds

To prevent interrupting DPORT workarounds, need to disable interrupt with a maximum used level in the system.

Found in Component config

Power Management

CONFIG_PM_ENABLE
Support for 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.

Found in Component config > Power Management

CONFIG_PM_DFS_INIT_AUTO
Enable dynamic frequency scaling (DFS) at startup

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.

Found in Component config > Power Management

CONFIG_PM_USE_RTC_TIMER_REF
Use RTC timer to prevent time drift (EXPERIMENTAL)

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.

Found in Component config > Power Management

CONFIG_PM_PROFILING
Enable profiling counters for PM locks

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.

Found in Component config > Power Management

CONFIG_PM_TRACE
Enable debug tracing of PM using GPIOs

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.

Found in Component config > Power Management

FreeRTOS

CONFIG_FREERTOS_UNICORE
Run FreeRTOS only on first core

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.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_CORETIMER

Xtensa timer to use as the FreeRTOS tick source

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

Found in Component config > FreeRTOS

Available options:

- Timer 0 (int 6, level 1) (*CONFIG_FREERTOS_CORETIMER_0*)

  Select this to use timer 0

- Timer 1 (int 15, level 3) (*CONFIG_FREERTOS_CORETIMER_1*)

  Select this to use timer 1

CONFIG_FREERTOS_HZ

Tick rate (Hz)

Select the tick rate at which FreeRTOS does pre-emptive context switching.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION

Halt when an SMP-untested function is called

Some functions in FreeRTOS have not been thoroughly tested yet when moving to the SMP implementation of FreeRTOS. When this option is enabled, these fuctions will throw an assert().

Found in Component config > FreeRTOS

CONFIG_FREERTOS_CHECK_STACKOVERFLOW

Check for stack overflow

FreeRTOS can check for stack overflows in threads and trigger an user function called vApplicationStackOverflowHook when this happens.

Found in Component config > FreeRTOS

Available options:

- No checking (*CONFIG_FREERTOS_CHECK_STACKOVERFLOW_NONE*)

  Do not check for stack overflows (configCHECK_FOR_STACK_OVERFLOW=0)

- Check by stack pointer value (*CONFIG_FREERTOS_CHECK_STACKOVERFLOW_PTRVAL*)

  Check for stack overflows on each context switch by checking if the stack pointer is in a valid range. Quick but does not detect stack overflows that happened between context switches (configCHECK_FOR_STACK_OVERFLOW=1)

- Check using canary bytes (*CONFIG_FREERTOS_CHECK_STACKOVERFLOW_CANARY*)

  Places some magic bytes at the end of the stack area and on each context switch, check if these bytes are still intact. More thorough than just checking the pointer, but also slightly slower. (configCHECK_FOR_STACK_OVERFLOW=2)

CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK

Set a debug watchpoint as a stack overflow check

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 on the stack of a task are overwritten. The side effect is that using gdb, you effectively only have one watchpoint; the 2nd one is overwritten as soon as a task switch happens.

This check only triggers if the stack overflow writes within 4 bytes of the end of the stack, rather than overshooting further, so it is worth combining this approach with one of the other stack overflow check methods.

When this watchpoint is hit, gdb will stop with a SIGTRAP message. When no JTAG OCD is attached, esp-idf will panic on an unhandled debug exception.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_INTERRUPT_BACKTRACE

Enable backtrace from interrupt to task context

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: higer level interrupt stack can be traced back to the lower level interrupt. This option adds 4 instructions to the interrupt dispatching code.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS

Number of thread local storage pointers

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.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_ASSERT

FreeRTOS assertions

Failed FreeRTOS configASSERT() assertions can be configured to behave in different ways.

Found in Component config > FreeRTOS

Available options:

- abort() on failed assertions (*CONFIG_FREERTOS_ASSERT_FAIL_ABORT*)

  If a FreeRTOS configASSERT() fails, FreeRTOS will abort() and halt execution. The panic handler can be configured to handle the outcome of an abort() in different ways.

- Print and continue failed assertions (*CONFIG_FREERTOS_ASSERT_FAIL_PRINT_CONTINUE*)

  If a FreeRTOS assertion fails, print it out and continue.

- Disable FreeRTOS assertions (*CONFIG_FREERTOS_ASSERT_DISABLE*)

  FreeRTOS configASSERT() will not be compiled into the binary.

CONFIG_FREERTOS_IDLE_TASK_STACKSIZE

Idle Task stack size

The idle task has its own stack, sized in bytes. The default size is enough for most uses. Size can be reduced to 768 bytes if no (or simple) FreeRTOS idle hooks are used and pthread local storage or FreeRTOS local storage cleanup callbacks are not used.

The stack size may need to be increased above the default if the app installs idle or thread local storage cleanup hooks that use a lot of stack memory.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_ISR_STACKSIZE
ISR stack size

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.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_LEGACY_HOOKS
Use FreeRTOS legacy hooks

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.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_LEGACY_IDLE_HOOK
Enable legacy idle hook

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.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_LEGACY_TICK_HOOK
Enable legacy tick hook

If enabled, FreeRTOS will call a function called vApplicationTickHook when a FreeRTOS tick is executed. Please make sure your code defines such a function.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_MAX_TASK_NAME_LEN
Maximum task name length

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.

Found in Component config > FreeRTOS

CONFIG_SUPPORT_STATIC_ALLOCATION
Enable FreeRTOS static allocation API

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).

Found in Component config > FreeRTOS

CONFIG_ENABLE_STATIC_TASK_CLEAN_UP_HOOK

Enable static task clean up hook

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
}
```

Found in Component config > FreeRTOS

CONFIG_TIMER_TASK_PRIORITY

FreeRTOS timer task priority

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.

Found in Component config > FreeRTOS

CONFIG_TIMER_TASK_STACK_DEPTH

FreeRTOS timer task stack size

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.

Found in Component config > FreeRTOS

CONFIG_TIMER_QUEUE_LENGTH

FreeRTOS timer queue length

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.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE

FreeRTOS queue registry size

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.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_USE_TRACE_FACILITY
Enable FreeRTOS trace facility

If enabled, configUSE_TRACE_FACILITY will be defined as 1 in FreeRTOS. This will allow the usage of trace facility functions such as uxTaskGetSystemState().

Found in Component config > FreeRTOS

CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS
Enable FreeRTOS stats formatting functions

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().

Found in Component config > FreeRTOS

CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS
Enable FreeRTOS to collect run time stats

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)

Found in Component config > FreeRTOS

CONFIG_FREERTOS_RUN_TIME_STATS_CLK
Choose the clock source for run time stats

Choose the clock source for FreeRTOS run time stats. Options are CPU0's CPU Clock or the ESP Timer. Both clock sources are 32 bits. The CPU Clock can run at a higher frequency hence provide a finer resolution but will overflow much quicker. Note that run time stats are only valid until the clock source overflows.

Found in Component config > FreeRTOS

Available options:

- Use ESP TIMER for run time stats (*CONFIG_FREERTOS_RUN_TIME_STATS_USING_ESP_TIMER*)

  ESP Timer will be used as the clock source for FreeRTOS run time stats. The ESP Timer runs at a frequency of 1MHz regardless of Dynamic Frequency Scaling. Therefore the ESP Timer will overflow in approximately 4290 seconds.

- Use CPU Clock for run time stats (*CONFIG_FREERTOS_RUN_TIME_STATS_USING_CPU_CLK*)

  CPU Clock will be used as the clock source for the generation of run time stats. The CPU Clock has a frequency dependent on ESP32_DEFAULT_CPU_FREQ_MHZ and Dynamic Frequency Scaling (DFS). Therefore the CPU Clock frequency can fluctuate between 80 to 240MHz. Run time stats generated using the CPU Clock represents the number of CPU cycles each task is allocated and DOES NOT reflect the amount of time each task runs for (as CPU clock frequency can change). If the CPU clock consistently runs at the maximum frequency of 240MHz, it will overflow in approximately 17 seconds.

CONFIG_FREERTOS_USE_TICKLESS_IDLE
Tickless idle support

If power management support is enabled, FreeRTOS will be able to put the system into light sleep mode when no tasks need to run for a number of ticks. This number can be set using FREERTOS_IDLE_TIME_BEFORE_SLEEP option. This feature is also known as "automatic light sleep".

Note that timers created using esp_timer APIs may prevent the system from entering sleep mode, even when no tasks need to run.

If disabled, automatic light sleep support will be disabled.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_IDLE_TIME_BEFORE_SLEEP
Minimum number of ticks to enter sleep mode for

FreeRTOS will enter light sleep mode if no tasks need to run for this number of ticks.

Found in Component config > FreeRTOS

CONFIG_FREERTOS_DEBUG_INTERNALS

Debug FreeRTOS internals

- *CONFIG_FREERTOS_PORTMUX_DEBUG*
- *CONFIG_FREERTOS_PORTMUX_DEBUG_RECURSIVE*

Debug FreeRTOS internals

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.

CONFIG_FREERTOS_PORTMUX_DEBUG
Debug portMUX portENTER_CRITICAL/portEXIT_CRITICAL

If enabled, debug information (including integrity checks) will be printed to UART for the port-specific MUX implementation.

Found in Component config > FreeRTOS > Debug FreeRTOS internals

CONFIG_FREERTOS_PORTMUX_DEBUG_RECURSIVE
Debug portMUX Recursion

If enabled, additional debug information will be printed for recursive portMUX usage.

Found in Component config > FreeRTOS > Debug FreeRTOS internals

Ethernet

CONFIG_DMA_RX_BUF_NUM
Number of DMA RX buffers

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 .

Found in Component config > Ethernet

CONFIG_DMA_TX_BUF_NUM
Number of DMA TX buffers

Number of DMA transmit buffers. Each buffer is 1600 bytes. Buffers are allocated statically. Larger number of buffers increases throughput.

Found in Component config > Ethernet

CONFIG_EMAC_L2_TO_L3_RX_BUF_MODE
Enable copy between Layer2 and Layer3

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 incomming 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.

Found in Component config > Ethernet

CONFIG_EMAC_TASK_PRIORITY
EMAC_TASK_PRIORITY

Ethernet MAC task priority.

Found in Component config > Ethernet

CONFIG_EMAC_TASK_STACK_SIZE
Stack Size of EMAC Task

Stack Size of Ethernet MAC task.

Found in Component config > Ethernet

ADC-Calibration

CONFIG_ADC_CAL_EFUSE_TP_ENABLE
Use Two Point Values

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.

Found in Component config > ADC-Calibration

CONFIG_ADC_CAL_EFUSE_VREF_ENABLE
Use eFuse Vref

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.

Found in Component config > ADC-Calibration

CONFIG_ADC_CAL_LUT_ENABLE
Use Lookup Tables

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.

Found in Component config > ADC-Calibration

SPI Flash driver

CONFIG_SPI_FLASH_VERIFY_WRITE
Verify SPI flash writes

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.

Found in Component config > SPI Flash driver

CONFIG_SPI_FLASH_LOG_FAILED_WRITE
Log errors if verification fails

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.

Found in Component config > SPI Flash driver

CONFIG_SPI_FLASH_WARN_SETTING_ZERO_TO_ONE
Log warning if writing zero bits to ones

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.

Found in Component config > SPI Flash driver

CONFIG_SPI_FLASH_ENABLE_COUNTERS
Enable operation counters

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.

Found in Component config > SPI Flash driver

CONFIG_SPI_FLASH_ROM_DRIVER_PATCH
Enable SPI flash ROM driver patched functions

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.

Found in Component config > SPI Flash driver

CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS
Writing to dangerous flash regions

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.

Found in Component config > SPI Flash driver

Available options:

- Aborts (*CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_ABORTS*)
- Fails (*CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_FAILS*)
- Allowed (*CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_ALLOWED*)

CONFIG_SPI_FLASH_YIELD_DURING_ERASE

Enables yield operation during flash erase

This allows to yield the CPUs between erase commands. Prevents starvation of other tasks.

Found in Component config > SPI Flash driver

CONFIG_SPI_FLASH_ERASE_YIELD_DURATION_MS

Duration of erasing to yield CPUs (ms)

If a duration of one erase command is large then it will yield CPUs after finishing a current command.

Found in Component config > SPI Flash driver

CONFIG_SPI_FLASH_ERASE_YIELD_TICKS

CPU release time (tick)

Defines how many ticks will be before returning to continue a erasing.

Found in Component config > SPI Flash driver

3.9.4 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.

3.10 Error Codes Reference

This section lists various error code constants defined in ESP-IDF.

For general information about error codes in ESP-IDF, see *Error Handling*.

ESP_FAIL (-1): Generic esp_err_t code indicating failure

ESP_OK (0): esp_err_t value indicating success (no error)

ESP_ERR_NO_MEM **(0x101)**: Out of memory

ESP_ERR_INVALID_ARG **(0x102)**: Invalid argument

ESP_ERR_INVALID_STATE **(0x103)**: Invalid state

ESP_ERR_INVALID_SIZE **(0x104)**: Invalid size

ESP_ERR_NOT_FOUND **(0x105)**: Requested resource not found

ESP_ERR_NOT_SUPPORTED **(0x106)**: Operation or feature not supported

ESP_ERR_TIMEOUT **(0x107)**: Operation timed out

ESP_ERR_INVALID_RESPONSE **(0x108)**: Received response was invalid

ESP_ERR_INVALID_CRC **(0x109)**: CRC or checksum was invalid

ESP_ERR_INVALID_VERSION **(0x10a)**: Version was invalid

ESP_ERR_INVALID_MAC **(0x10b)**: MAC address was invalid

ESP_ERR_NVS_BASE **(0x1100)**: Starting number of error codes

ESP_ERR_NVS_NOT_INITIALIZED **(0x1101)**: The storage driver is not initialized

ESP_ERR_NVS_NOT_FOUND **(0x1102)**: Id namespace doesn't exist yet and mode is NVS_READONLY

ESP_ERR_NVS_TYPE_MISMATCH **(0x1103)**: The type of set or get operation doesn't match the type of value stored in NVS

ESP_ERR_NVS_READ_ONLY **(0x1104)**: Storage handle was opened as read only

ESP_ERR_NVS_NOT_ENOUGH_SPACE **(0x1105)**: There is not enough space in the underlying storage to save the value

ESP_ERR_NVS_INVALID_NAME **(0x1106)**: Namespace name doesn't satisfy constraints

ESP_ERR_NVS_INVALID_HANDLE **(0x1107)**: Handle has been closed or is NULL

ESP_ERR_NVS_REMOVE_FAILED **(0x1108)**: The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

ESP_ERR_NVS_KEY_TOO_LONG **(0x1109)**: Key name is too long

ESP_ERR_NVS_PAGE_FULL **(0x110a)**: Internal error; never returned by nvs API functions

ESP_ERR_NVS_INVALID_STATE **(0x110b)**: NVS is in an inconsistent state due to a previous error. Call nvs_flash_init and nvs_open again, then retry.

ESP_ERR_NVS_INVALID_LENGTH **(0x110c)**: String or blob length is not sufficient to store data

ESP_ERR_NVS_NO_FREE_PAGES **(0x110d)**: NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call nvs_flash_init again.

ESP_ERR_NVS_VALUE_TOO_LONG **(0x110e)**: String or blob length is longer than supported by the implementation

ESP_ERR_NVS_PART_NOT_FOUND **(0x110f)**: Partition with specified name is not found in the partition table

ESP_ERR_NVS_NEW_VERSION_FOUND **(0x1110)**: NVS partition contains data in new format and cannot be recognized by this version of code

ESP_ERR_NVS_XTS_ENCR_FAILED **(0x1111)**: XTS encryption failed while writing NVS entry

ESP_ERR_NVS_XTS_DECR_FAILED **(0x1112)**: XTS decryption failed while reading NVS entry

ESP_ERR_NVS_XTS_CFG_FAILED **(0x1113)**: XTS configuration setting failed

ESP_ERR_NVS_XTS_CFG_NOT_FOUND **(0x1114)**: XTS configuration not found

ESP_ERR_NVS_ENCR_NOT_SUPPORTED **(0x1115)**: NVS encryption is not supported in this version

ESP_ERR_NVS_KEYS_NOT_INITIALIZED **(0x1116)**: NVS key partition is uninitialized

ESP_ERR_NVS_CORRUPT_KEY_PART **(0x1117)**: NVS key partition is corrupt

ESP_ERR_ULP_BASE **(0x1200)**: Offset for ULP-related error codes

ESP_ERR_ULP_SIZE_TOO_BIG **(0x1201)**: Program doesn't fit into RTC memory reserved for the ULP

ESP_ERR_ULP_INVALID_LOAD_ADDR **(0x1202)**: Load address is outside of RTC memory reserved for the ULP

ESP_ERR_ULP_DUPLICATE_LABEL **(0x1203)**: More than one label with the same number was defined

ESP_ERR_ULP_UNDEFINED_LABEL **(0x1204)**: Branch instructions references an undefined label

ESP_ERR_ULP_BRANCH_OUT_OF_RANGE **(0x1205)**: Branch target is out of range of B instruction (try replacing with BX)

ESP_ERR_OTA_BASE **(0x1500)**: Base error code for ota_ops api

ESP_ERR_OTA_PARTITION_CONFLICT **(0x1501)**: Error if request was to write or erase the current running partition

ESP_ERR_OTA_SELECT_INFO_INVALID **(0x1502)**: Error if OTA data partition contains invalid content

ESP_ERR_OTA_VALIDATE_FAILED **(0x1503)**: Error if OTA app image is invalid

ESP_ERR_IMAGE_BASE **(0x2000)**

ESP_ERR_IMAGE_FLASH_FAIL **(0x2001)**

ESP_ERR_IMAGE_INVALID **(0x2002)**

ESP_ERR_WIFI_BASE **(0x3000)**: Starting number of WiFi error codes

ESP_ERR_WIFI_NOT_INIT **(0x3001)**: WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED **(0x3002)**: WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED **(0x3003)**: WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF **(0x3004)**: WiFi interface error

ESP_ERR_WIFI_MODE **(0x3005)**: WiFi mode error

ESP_ERR_WIFI_STATE **(0x3006)**: WiFi internal state error

ESP_ERR_WIFI_CONN **(0x3007)**: WiFi internal control block of station or soft-AP error

ESP_ERR_WIFI_NVS **(0x3008)**: WiFi internal NVS module error

ESP_ERR_WIFI_MAC **(0x3009)**: MAC address is invalid

ESP_ERR_WIFI_SSID **(0x300a)**: SSID is invalid

ESP_ERR_WIFI_PASSWORD **(0x300b)**: Password is invalid

ESP_ERR_WIFI_TIMEOUT **(0x300c)**: Timeout error

ESP_ERR_WIFI_WAKE_FAIL **(0x300d)**: WiFi is in sleep state(RF closed) and wakeup fail

ESP_ERR_WIFI_WOULD_BLOCK **(0x300e)**: The caller would block

ESP_ERR_WIFI_NOT_CONNECT **(0x300f)**: Station still in disconnect status

ESP_ERR_WIFI_POST **(0x3012)**: Failed to post the event to WiFi task

ESP_ERR_WIFI_INIT_STATE **(0x3013)**: Invalod WiFi state when init/deinit is called

ESP_ERR_WIFI_STOP_STATE **(0x3014)**: Returned when WiFi is stopping

ESP_ERR_WIFI_REGISTRAR **(0x3033)**: WPS registrar is not supported

ESP_ERR_WIFI_WPS_TYPE **(0x3034)**: WPS type error

ESP_ERR_WIFI_WPS_SM **(0x3035)**: WPS state machine is not initialized

ESP_ERR_ESPNOW_BASE **(0x3064)**: ESPNOW error number base.

ESP_ERR_ESPNOW_NOT_INIT **(0x3065)**: ESPNOW is not initialized.

ESP_ERR_ESPNOW_ARG **(0x3066)**: Invalid argument

ESP_ERR_ESPNOW_NO_MEM **(0x3067)**: Out of memory

ESP_ERR_ESPNOW_FULL **(0x3068)**: ESPNOW peer list is full

ESP_ERR_ESPNOW_NOT_FOUND **(0x3069)**: ESPNOW peer is not found

ESP_ERR_ESPNOW_INTERNAL **(0x306a)**: Internal error

ESP_ERR_ESPNOW_EXIST **(0x306b)**: ESPNOW peer has existed

ESP_ERR_ESPNOW_IF **(0x306c)**: Interface error

ESP_ERR_MESH_BASE **(0x4000)**: Starting number of MESH error codes

ESP_ERR_MESH_WIFI_NOT_START **(0x4001)**

ESP_ERR_MESH_NOT_INIT **(0x4002)**

ESP_ERR_MESH_NOT_CONFIG **(0x4003)**

ESP_ERR_MESH_NOT_START **(0x4004)**

ESP_ERR_MESH_NOT_SUPPORT **(0x4005)**

ESP_ERR_MESH_NOT_ALLOWED **(0x4006)**

ESP_ERR_MESH_NO_MEMORY **(0x4007)**

ESP_ERR_MESH_ARGUMENT **(0x4008)**

ESP_ERR_MESH_EXCEED_MTU **(0x4009)**

ESP_ERR_MESH_TIMEOUT **(0x400a)**

ESP_ERR_MESH_DISCONNECTED **(0x400b)**

ESP_ERR_MESH_QUEUE_FAIL **(0x400c)**

ESP_ERR_MESH_QUEUE_FULL **(0x400d)**

ESP_ERR_MESH_NO_PARENT_FOUND **(0x400e)**

ESP_ERR_MESH_NO_ROUTE_FOUND **(0x400f)**

ESP_ERR_MESH_OPTION_NULL **(0x4010)**

ESP_ERR_MESH_OPTION_UNKNOWN **(0x4011)**

ESP_ERR_MESH_XON_NO_WINDOW **(0x4012)**

ESP_ERR_MESH_INTERFACE **(0x4013)**

ESP_ERR_MESH_DISCARD_DUPLICATE **(0x4014)**

ESP_ERR_MESH_DISCARD **(0x4015)**

ESP_ERR_MESH_VOTING **(0x4016)**

ESP_ERR_TCPIP_ADAPTER_BASE **(0x5000)**

ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS **(0x5001)**

ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY **(0x5002)**

ESP_ERR_TCPIP_ADAPTER_DHCPC_START_FAILED **(0x5003)**

ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STARTED **(0x5004)**

ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STOPPED **(0x5005)**

`ESP_ERR_TCPIP_ADAPTER_NO_MEM` **(0x5006)**

`ESP_ERR_TCPIP_ADAPTER_DHCP_NOT_STOPPED` **(0x5007)**

`ESP_ERR_PING_BASE` **(0x6000)**

`ESP_ERR_PING_INVALID_PARAMS` **(0x6001)**

`ESP_ERR_PING_NO_MEM` **(0x6002)**

`ESP_ERR_HTTP_BASE` **(0x7000)**: Starting number of HTTP error codes

`ESP_ERR_HTTP_MAX_REDIRECT` **(0x7001)**: The error exceeds the number of HTTP redirects

`ESP_ERR_HTTP_CONNECT` **(0x7002)**: Error open the HTTP connection

`ESP_ERR_HTTP_WRITE_DATA` **(0x7003)**: Error write HTTP data

`ESP_ERR_HTTP_FETCH_HEADER` **(0x7004)**: Error read HTTP header from server

`ESP_ERR_HTTP_INVALID_TRANSPORT` **(0x7005)**: There are no transport support for the input scheme

`ESP_ERR_FLASH_BASE` **(0x10010)**

`ESP_ERR_FLASH_OP_FAIL` **(0x10011)**

`ESP_ERR_FLASH_OP_TIMEOUT` **(0x10012)**

CHAPTER 4

ESP32 Hardware Reference

4.1 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*.

4.1.1 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.

–	Key Components				Dimensions [mm]		
Module	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

- U.FL - U.FL / IPEX antenna connector
- ESP32 Chip Datasheet (PDF)

ESP-WROOM-32

A basic and commonly adopted ESP32 module with ESP32-D0WDQ6 chip on board. The first one of the WROOM / WROVER family released to the market.

Fig. 1: ESP-WROOM-32 module (front and back)

Documentation

- ESP-WROOM-32 Schematic (PDF)
- ESP-WROOM-32 Datasheet (PDF)
- ESP32 Module Reference Design (ZIP) containing OrCAD schematic, PCB layout, gerbers and BOM

ESP-WROOM-32D / ESP32-WROOM-32U

Both modules have ESP32-D0WD chip on board of a smaller footprint than ESP32-D0WDQ6 installed in *ESP-WROOM-32*. Version "D" has a MIFA antenna. Version "U" has just an U.FL / IPEX antenna connector. That makes it 6.3 mm shorter comparing to "D", and also the smallest representative of the whole WROOM / WROVER family of modules.

Fig. 2: ESP-WROOM-32D module (back and front)

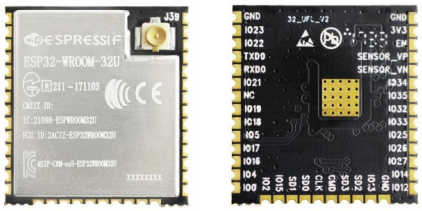


Fig. 3: ESP32-WROOM-32U module (back and front)

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*.

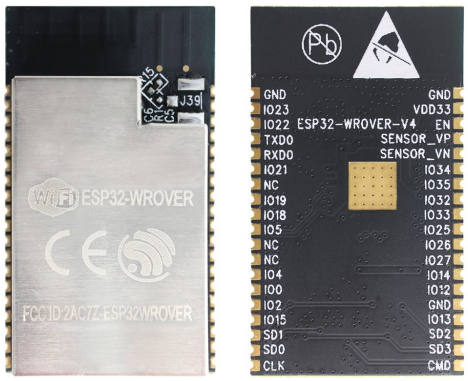


Fig. 4: ESP32-WROVER module (front and back)

Documentation

- ESP32-WROVER Datasheet (PDF)
- ESP-PSRAM32 Datasheet (PDF)
- ESP32-WROVER Reference Design PDF containing OrCAD schematic, PCB layout, gerbers and BOM

4.1.2 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.

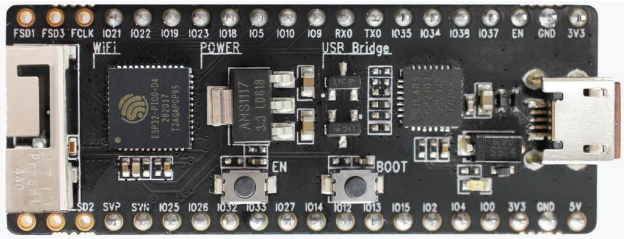


Fig. 5: ESP32-PICO-KIT V4 board

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*

4.1.3 ESP32 DevKitC V4

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. Comparing to the previous *ESP32 Core Board V2 / ESP32 DevKitC*, instead of ESP-WROOM-32 it can accommodate *ESP32-WROVER* module and has CP2102N chip that supports faster baud rates.

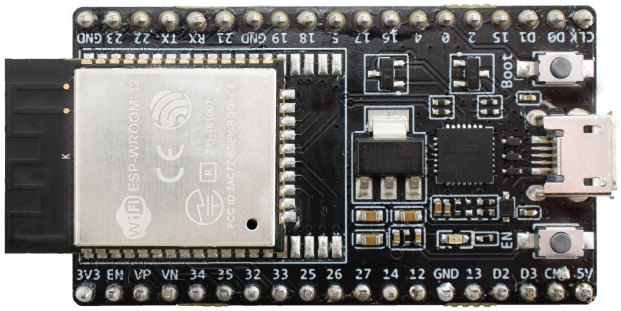


Fig. 6: ESP32 DevKitC V4 board

Documentation

- *ESP32-DevKitC V4 Getting Started Guide*
- ESP32-DevKitC schematic (PDF)
- ESP32-DevKitC Reference Design (ZIP) containing OrCAD schematic, PCB layout, gerbers and BOM
- CP210x USB to UART Bridge VCP Drivers

Previous Versions

- *ESP32 Core Board V2 / ESP32 DevKitC*

4.1.4 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*

4.1.5 Related Documents

- *Previous Versions of ESP32 Modules and Boards*

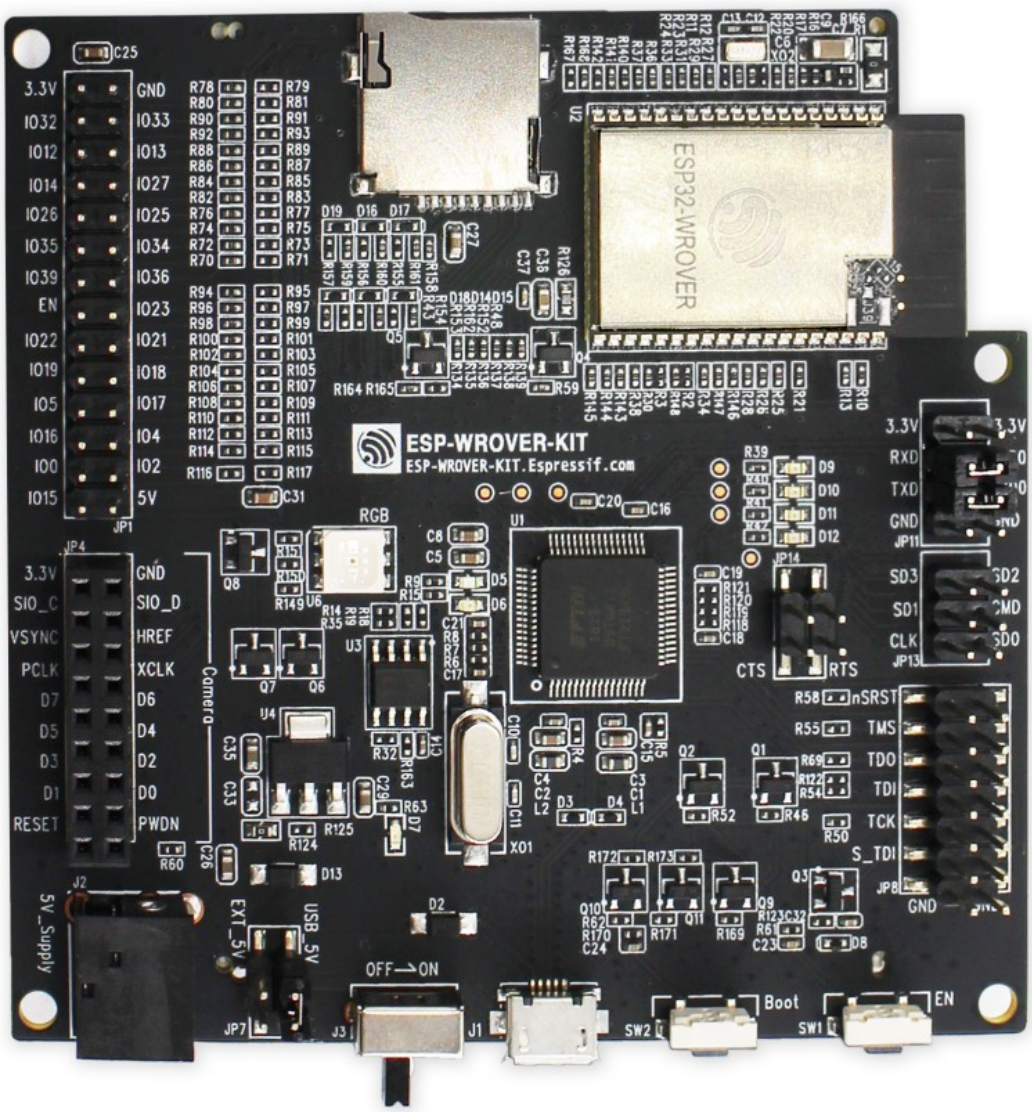


Fig. 7: ESP-WROVER-KIT V3 board

4.2 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*.

4.2.1 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.

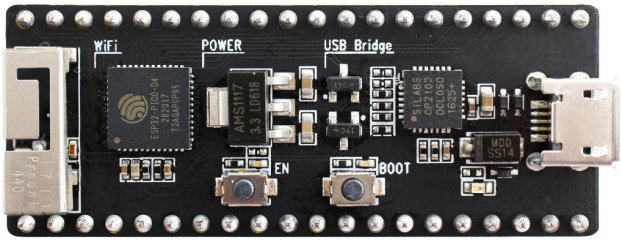


Fig. 8: ESP32-PICO-KIT V3 board

Documentation

- *ESP32-PICO-KIT V3 Getting Started Guide*
- ESP32-PICO-KIT V3 Schematic (PDF)
- ESP32-PICO-D4 Datasheet (PDF)

4.2.2 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.

Documentation

- *ESP32-DevKitC V2 Getting Started Guide*
- ESP32 DevKitC V2 Schematic (PDF)
- CP210x USB to UART Bridge VCP Drivers

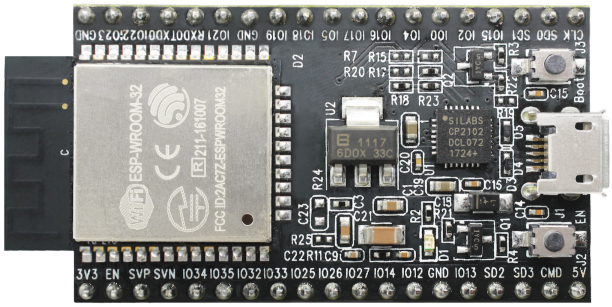


Fig. 9: ESP32 Core Board V2 / ESP32 DevKitC board

4.2.3 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

4.2.4 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.

Documentation

- *ESP-WROVER-KIT V2 Getting Started Guide*
- ESP-WROVER-KIT V2 Schematic (PDF)
- *JTAG Debugging*
- FTDI Virtual COM Port Drivers

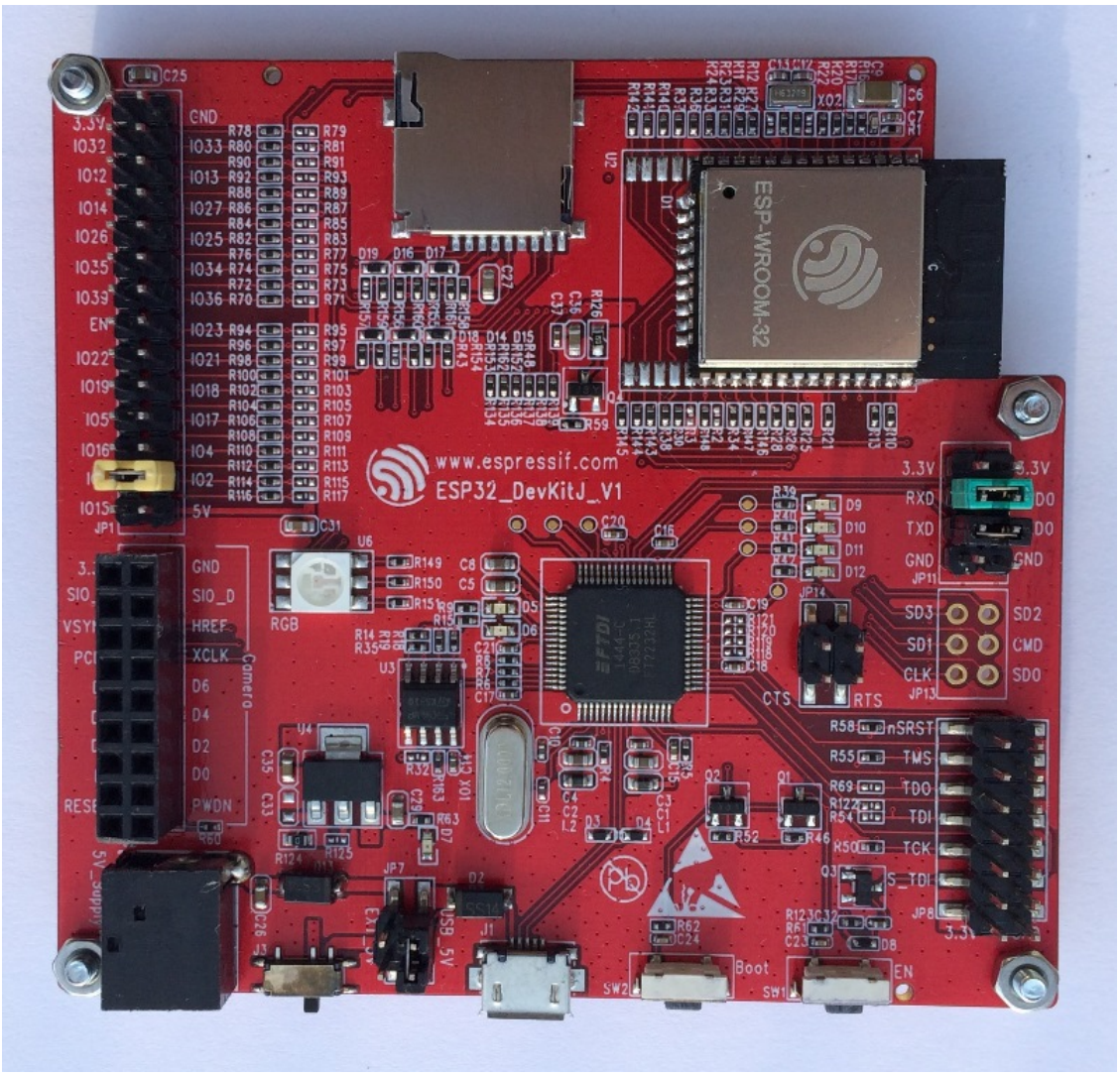


Fig. 10: ESP-WROVER-KIT V1 / ESP32 DevKitJ V1 board

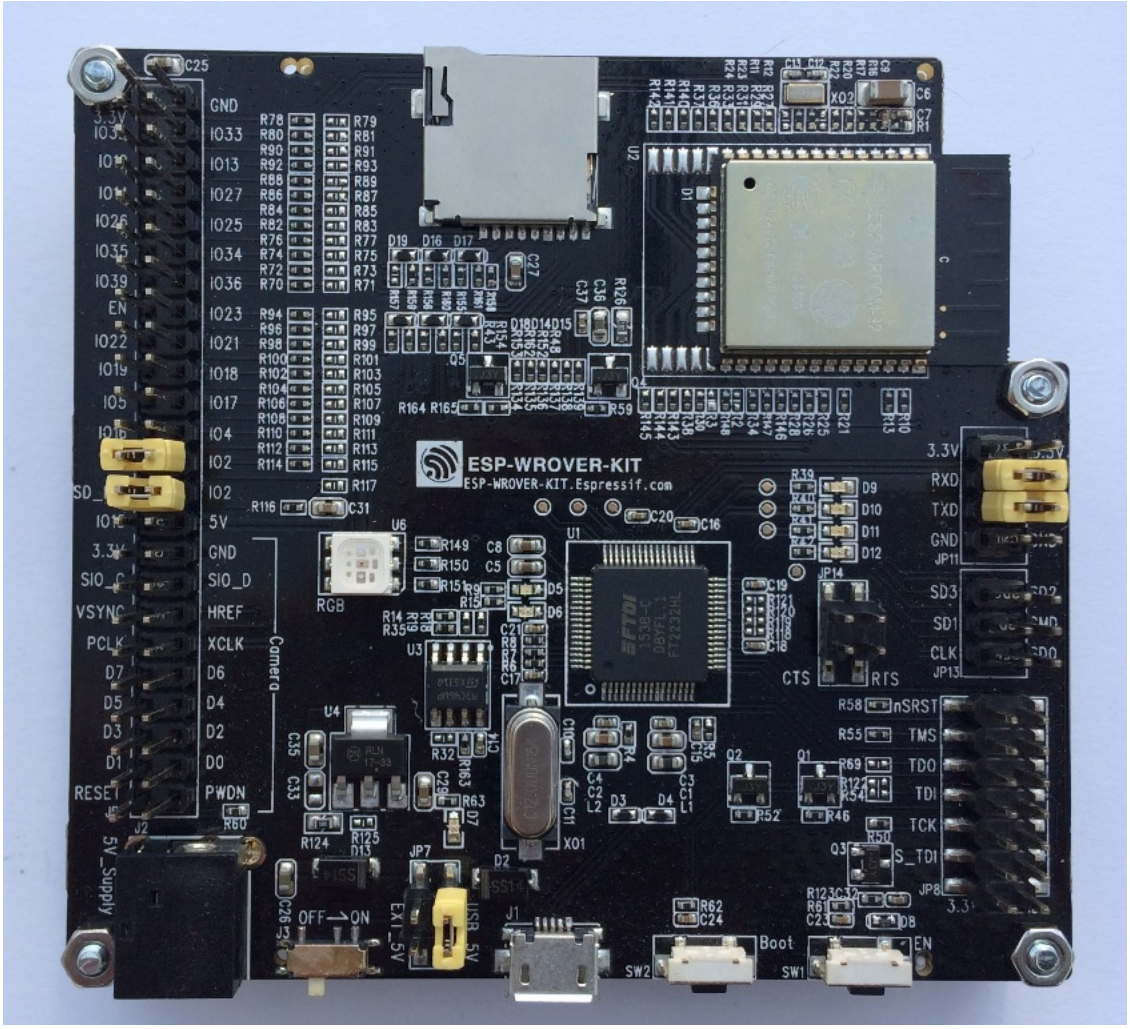


Fig. 11: ESP-WROVER-KIT V2 board

4.2.5 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.

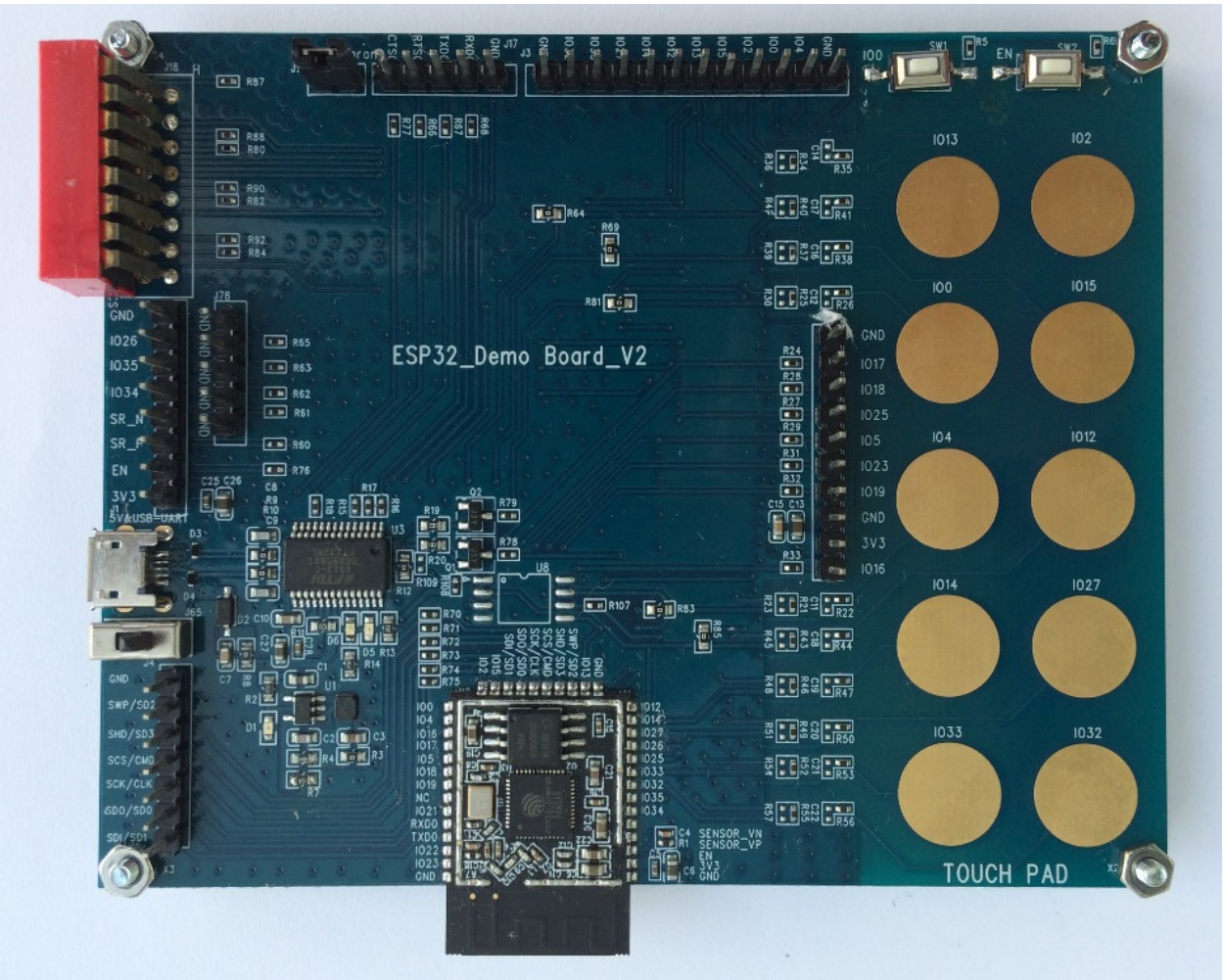


Fig. 12: ESP32 Demo Board V2

Production of this board is discontinued.

Documentation

- ESP32 Demo Board V2 Schematic (PDF)
- FTDI Virtual COM Port Drivers

4.2.6 Related Documents

- *ESP32 Modules and Boards*

CHAPTER 5

API Guides

5.1 General Notes About ESP-IDF Programming

5.1.1 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.

5.1.2 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.

The macro `__NOINIT_ATTR` can be used as attribute to place data into `.noinit` section. The values placed into this section will not be initialized at startup and keep its value after software restart.

Example:

```
__NOINIT_ATTR uint32_t noinit_data;
```

DROM (data stored in Flash)

By default, constant data is placed by the linker into a 4 MB region (`0x3F400000 -- 0x3F800000`) which is used to access external flash memory via Flash MMU and cache. Exceptions to this are literal constants which are embedded by the compiler into application code.

RTC slow memory

Global and static variables used by code which runs from RTC memory (i.e. deep sleep stub code) must be placed into RTC slow memory. Please check detailed description in *deep sleep* documentation.

The attribute macro named `RTC_NOINIT_ATTR` can be used to place data into this type of memory. The values placed into this section keep their value after waking from deep sleep.

Example:

```
RTC_NOINIT_ATTR uint32_t rtc_noinit_data;
```

5.1.3 DMA Capable Requirement

Most DMA controllers (e.g. SPI, sdmmc, etc.) have requirements that sending/receiving buffers should be placed in DRAM and word-aligned. We suggest to place DMA buffers in static variables rather than in the stack. Use macro `DMA_ATTR` to declare global/local static variables like:

```
DMA_ATTR uint8_t buffer[]="I want to send something";

void app_main()
{
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8*sizeof(buffer),
    };
    spi_device_transmit( spi, &temp );
    // other stuff
}
```

Or:

```
void app_main()
{
    DMA_ATTR static uint8_t buffer[]="I want to send something";
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8*sizeof(buffer),
    };
    spi_device_transmit( spi, &temp );
    // other stuff
}
```

Placing DMA buffers in the stack is still allowed, though you have to keep in mind:

1. Never try to do this if the stack is in the pSRAM. If the stack of a task is placed in the pSRAM, several steps have to be taken as described in *Support for external RAM* (at least `SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY` option enabled in the menuconfig). Make sure your task is not in the pSRAM.
2. Use macro `WORD_ALIGNED_ATTR` in functions before variables to place them in proper positions like:

```
void app_main()
{
    uint8_t stuff;
    WORD_ALIGNED_ATTR uint8_t buffer[]="I want to send something";   //or the␣
↪buffer will be placed right after stuff.
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8*sizeof(buffer),
    };
    spi_device_transmit( spi, &temp );
    // other stuff
}
```

5.2 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.

5.2.1 Using the Build System

The esp-idf README file contains a description of how to use the build system to build your project.

5.2.2 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 a text-based menu system to customize each component. After the components in the project are configured, the build process will compile the project.

Concepts

- A "project" is a directory that contains all the files and configuration to build a single "app" (executable), as well as additional supporting output such as a partition table, data/filesystem partitions, and a bootloader.
- "Project configuration" is held in a single file called sdkconfig in the root directory of the project. This configuration file is modified via `make menuconfig` to customise the configuration of the project. A single project contains exactly one project configuration.
- An "app" is an executable which is built by esp-idf. A single project will usually build two apps - a "project app" (the main executable, ie your custom firmware) and a "bootloader app" (the initial bootloader program which launches the project app).
- "components" are modular pieces of standalone code which are compiled into static libraries (.a files) and linked into an app. Some are provided by esp-idf itself, others may be sourced from other places.

Some things are not part of the project:

- "ESP-IDF" is not part of the project. Instead it is standalone, and linked to the project via the `IDF_PATH` environment variable which holds the path of the `esp-idf` directory. This allows the IDF framework to be decoupled from your project.

- The toolchain for compilation is not part of the project. The toolchain should be installed in the system command line PATH, or the path to the toolchain can be set as part of the compiler prefix in the project configuration.

Example Project

An example project directory tree might look like this:

```
- myProject/
             - Makefile
             - sdkconfig
             - components/ - component1/ - component.mk
                                         - Kconfig
                                         - src1.c
                           - component2/ - component.mk
                                         - Kconfig
                                         - src1.c
                                         - include/ - component2.h
             - main/       - src1.c
                           - src2.c
                           - component.mk

             - build/
```

This example "myProject" contains the following elements:

- A top-level project Makefile. This Makefile 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.
- `EXCLUDE_COMPONENTS`: Optional list of component names to exclude during the build process. Note that this decreases build time, but not binary size.

Any paths in these Makefile variables should be absolute paths. You can convert relative paths using `$(PROJECT_PATH)/xxx`, `$(IDF_PATH)/xxx`, or use the Make function `$(abspath xxx)`.

These variables should all be set before the line `include $(IDF_PATH)/make/project.mk` in the Makefile.

Component Makefiles

Each project contains one or more components, which can either be part of esp-idf or added from other component directories.

A component is any directory that contains a `component.mk` file.

Searching for Components

The list of directories in `COMPONENT_DIRS` is searched for the project's components. Directories in this list can either be components themselves (ie they contain a *component.mk* file), or they can be top-level directories whose subdirectories are components.

Running the `make list-components` target dumps many of these variables and can help debug the discovery of component directories.

Multiple components with the same name

When esp-idf is collecting all the components to compile, it will do this in the order specified by `COMPONENT_DIRS`; by default, this means the idf components first, the project components second and optionally the components in `EXTRA_COMPONENT_DIRS` last. If two or more of these directories contain component subdirectories with the same name, the component in the last place searched is used. This allows, for example, overriding esp-idf components with a modified version by simply copying the component from the esp-idf component directory to the project component tree and then modifying it there. If used in this way, the esp-idf directory itself can remain untouched.

Minimal Component Makefile

The minimal `component.mk` file is an empty file(!). If the file is empty, the default component behaviour is set:

- All source files in the same directory as the makefile (`*.c`, `*.cpp`, `*.cc`, `*.S`) will be compiled into the component library
- A sub-directory "include" will be added to the global include search path for all other components.
- The component library will be linked into the project app.

See *example component makefiles* for more complete component makefile examples.

Note that there is a difference between an empty `component.mk` file (which invokes default component build behaviour) and no `component.mk` file (which means no default component build behaviour will occur.) It is possible for a component to have no *component.mk* file, if it only contains other files which influence the project configuration or build process.

Preset Component Variables

The following component-specific variables are available for use inside `component.mk`, but should not be modified:

- `COMPONENT_PATH`: The component directory. Evaluates to the absolute path of the directory containing `component.mk`. The component path cannot contain spaces.
- `COMPONENT_NAME`: Name of the component. Defaults to the name of the component directory.
- `COMPONENT_BUILD_DIR`: The component build directory. Evaluates to the absolute path of a directory inside *$(BUILD_DIR_BASE)* where this component's source files are to be built. This is also the Current Working Directory any time the component is being built, so relative paths in make targets, etc. will be relative to this directory.
- `COMPONENT_LIBRARY`: Name of the static library file (relative to the component build directory) that will be built for this component. Defaults to `$(COMPONENT_NAME).a`.

The following variables are set at the project level, but exported for use in the component build:

- `PROJECT_NAME`: Name of the project, as set in project Makefile
- `PROJECT_PATH`: Absolute path of the project directory containing the project Makefile.
- `COMPONENTS`: Name of all components that are included in this build.
- `CONFIG_*`: Each value in the project configuration has a corresponding variable available in make. All names begin with `CONFIG_`.
- `CC`, `LD`, `AR`, `OBJCOPY`: Full paths to each tool from the gcc xtensa cross-toolchain.
- `HOSTCC`, `HOSTLD`, `HOSTAR`: Full names of each tool from the host native toolchain.

- `IDF_VER`: ESP-IDF version, retrieved from either `$(IDF_PATH)/version.txt` file (if present) else using git command `git describe`. Recommended format here is single liner that specifies major IDF release version, e.g. `v2.0` for a tagged release or `v2.0-275-g0efaa4f` for an arbitrary commit. Application can make use of this by calling `esp_get_idf_version()`.

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 contains configuration settings to add to the "make menuconfig" for this component.

These settings are found under the "Component Settings" menu when menuconfig is run.

To create a component KConfig file, it is easiest to start with one of the KConfig files distributed with esp-idf.

For an example, see *Adding conditional configuration*.

Preprocessor Definitions

ESP-IDF build systems adds the following C preprocessor definitions on the command line:

- `ESP_PLATFORM` — Can be used to detect that build happens within ESP-IDF.
- `IDF_VER` — ESP-IDF version, see *Preset Component Variables* for more details.

Build Process Internals

Top Level: Project Makefile

- "make" is always run from the project directory and the project makefile, typically named Makefile.
- The project makefile sets `PROJECT_NAME` and optionally customises other *optional project variables*

- The project makefile includes `$(IDF_PATH)/make/project.mk` which contains the project-level Make logic.
- `project.mk` fills in default project-level make variables and includes make variables from the project configuration. If the generated makefile containing project configuration is out of date, then it is regenerated (via targets in `project_config.mk`) and then the make process restarts from the top.
- `project.mk` builds a list of components to build, based on the default component directories or a custom list of components set in *optional project variables*.
- Each component can set some *optional project-wide component variables*. These are included via generated makefiles named `component_project_vars.mk` - there is one per component. These generated makefiles are included into `project.mk`. If any are missing or out of date, they are regenerated (via a recursive make call to the component makefile) and then the make process restarts from the top.
- *Makefile.projbuild* files from components are included into the make process, to add extra targets or configuration.
- By default, the project makefile also generates top-level build & clean targets for each component and sets up *app* and *clean* targets to invoke all of these sub-targets.
- In order to compile each component, a recursive make is performed for the component makefile.

To better understand the project make process, have a read through the `project.mk` file itself.

Second Level: Component Makefiles

- Each call to a component makefile goes via the `$(IDF_PATH)/make/component_wrapper.mk` wrapper makefile.
- This component wrapper includes all component `Makefile.componentbuild` files, making any recipes, variables etc in these files available to every component.
- The `component_wrapper.mk` is called with the current directory set to the component build directory, and the `COMPONENT_MAKEFILE` variable is set to the absolute path to `component.mk`.
- `component_wrapper.mk` sets default values for all *component variables*, then includes the *component.mk* file which can override or modify these.
- If `COMPONENT_OWNBUILDTARGET` and `COMPONENT_OWNCLEANTARGET` are not defined, default build and clean targets are created for the component's source files and the prerequisite `COMPONENT_LIBRARY` static library file.
- The `component_project_vars.mk` file has its own target in `component_wrapper.mk`, which is evaluated from `project.mk` if this file needs to be rebuilt due to changes in the component makefile or the project configuration.

To better understand the component make process, have a read through the `component_wrapper.mk` file and some of the `component.mk` files included with esp-idf.

Running Make Non-Interactively

When running `make` in a situation where you don't want interactive prompts (for example: inside an IDE or an automated build system) append `BATCH_BUILD=1` to the make arguments (or set it as an environment variable).

Setting `BATCH_BUILD` implies the following:

- Verbose output (same as `V=1`, see below). If you don't want verbose output, also set `V=0`.

- If the project configuration is missing new configuration items (from new components or esp-idf updates) then the project use the default values, instead of prompting the user for each item.
- If the build system needs to invoke `menuconfig`, an error is printed and the build fails.

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 LDFLAGS 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 explicitely, 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 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.)

Note: When using an external build process with PSRAM, remember to add `-mfix-esp32-psram-cache-issue` to the C compiler arguments. See *CONFIG_SPIRAM_CACHE_WORKAROUND* for details of this flag.

Custom sdkconfig defaults

For example projects or other projects where you don't want to specify a full sdkconfig configuration, but you do want to override some key values from the esp-idf defaults, it is possible to create a file `sdkconfig.defaults` in the project directory. This file will be used when running `make defconfig`, or creating a new config from scratch.

To override the name of this file, set the `SDKCONFIG_DEFAULTS` environment variable.

Save flash arguments

There're some scenarios that we want to flash the target board without IDF. For this case we want to save the built binaries, esptool.py and esptool write_flash arguments. It's simple to write a script to save binaries and esptool.py. We can use command `make print_flash_cmd`, it will print the flash arguments:

```
--flash_mode dio --flash_freq 40m --flash_size detect 0x1000 bootloader/bootloader.
↪bin 0x10000 example_app.bin 0x8000 partition_table_unit_test_app.bin
```

Then use flash arguments as the arguemnts for esptool write_flash arguments:

```
python esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 921600 --before default_
↪reset --after hard_reset write_flash -z --flash_mode dio --flash_freq 40m --flash_
↪size detect 0x1000 bootloader/bootloader.bin 0x10000 example_app.bin 0x8000␣
↪partition_table_unit_test_app.bin
```

5.2.3 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.

5.3 Build System (CMake)

Note: This is documentation for the CMake-based build system which is currently in preview release. If you encounter any gaps or bugs, please report them in the Issues section of the ESP-IDF repository.

The CMake-based build system will become the default build system in ESP-IDF V4.0. The existing GNU Make based build system will be deprecated in ESP-IDF V5.0.

Important: The following features are not yet supported with the CMake-based build system:

- Eclipse IDE Documentation
- Secure Boot
- Flash Encryption
- ULP toolchain.

Support for these features will be available before CMake becomes the default build system.

This document explains the implementation of the CMake-based ESP-IDF build system and the concept of "components". *Documentation for the GNU Make based build system* is also available

Read this document if you want to know how to organise and build a new ESP-IDF project or component using the CMake-based build system.

5.3.1 Overview

An ESP-IDF project can be seen as an amalgamation of a number of components. For example, for a webserver that shows the current humidity, there could be:

- The ESP32 base libraries (libc, ROM bindings, etc)
- The 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 system will look up all the components in the ESP-IDF directories, the project directories and (optionally) in additional custom component directories. It then allows the user to configure the ESP-IDF project using a a text-based menu system to customize each component. After the components in the project are configured, the build system will compile the project.

Concepts

- A "project" is a directory that contains all the files and configuration to build a single "app" (executable), as well as additional supporting elements such as a partition table, data/filesystem partitions, and a bootloader.
- "Project configuration" is held in a single file called `sdkconfig` in the root directory of the project. This configuration file is modified via `idf.py menuconfig` to customise the configuration of the project. A single project contains exactly one project configuration.
- An "app" is an executable which is built by ESP-IDF. A single project will usually build two apps - a "project app" (the main executable, ie your custom firmware) and a "bootloader app" (the initial bootloader program which launches the project app).
- "components" are modular pieces of standalone code which are compiled into static libraries (.a files) and linked into an app. Some are provided by ESP-IDF itself, others may be sourced from other places.

Some things are not part of the project:

- "ESP-IDF" is not part of the project. Instead it is standalone, and linked to the project via the `IDF_PATH` environment variable which holds the path of the `esp-idf` directory. This allows the IDF framework to be decoupled from your project.
- The toolchain for compilation is not part of the project. The toolchain should be installed in the system command line PATH.

5.3.2 Using the Build System

idf.py

The `idf.py` command line tool provides a front-end for easily managing your project builds. It manages the following tools:

- CMake, which configures the project to be built
- A command line build tool (either Ninja build or *GNU Make*)
- esptool.py for flashing ESP32.

The *getting started guide* contains a brief introduction to how to set up `idf.py` to configure, build, and flash projects.

`idf.py` should be run in an ESP-IDF "project" directory, ie one containing a `CMakeLists.txt` file. Older style projects with a Makefile will not work with `idf.py`.

Type `idf.py --help` for a full list of commands. Here are a summary of the most useful ones:

- `idf.py menuconfig` runs the "menuconfig" tool to configure the project.
- `idf.py build` will build the project found in the current directory. This can involve multiple steps:
  - Create the build directory if needed. The sub-directory `build` is used to hold build output, although this can be changed with the `-B` option.
  - Run CMake as necessary to configure the project and generate build files for the main build tool.
  - Run the main build tool (Ninja or *GNU Make*). By default, the build tool is automatically detected but it can be explicitly set by passing the `-G` option to `idf.py`.

  Building is incremental so if no source files or configuration has changed since the last build, nothing will be done.

- `idf.py clean` will "clean" the project by deleting build output files from the build directory, forcing a "full rebuild" the next time the project is built. Cleaning doesn't delete CMake configuration output and some other files.
- `idf.py fullclean` will delete the entire "build" directory contents. This includes all CMake configuration output. The next time the project is built, CMake will configure it from scratch. Note that this option recursively deletes *all* files in the build directory, so use with care. Project configuration is not deleted.
- `idf.py reconfigure` re-runs CMake even if it doesn't seem to need re-running. This isn't necessary during normal usage, but can be useful after adding/removing files from the source tree.
- `idf.py flash` will automatically build the project if necessary, and then flash it to an ESP32. The `-p` and `-b` options can be used to set serial port name and flasher baud rate, respectively.
- `idf.py monitor` will display serial output from the ESP32. The `-p` option can be used to set the serial port name. Type `Ctrl-]` to exit the monitor. See *IDF Monitor* for more details about using the monitor.

Multiple `idf.py` commands can be combined into one. For example, `idf.py -p COM4 clean flash monitor` will clean the source tree, then build the project and flash it to the ESP32 before running the serial monitor.

Note: The environment variables `ESPPORT` and `ESPBAUD` can be used to set default values for the `-p` and `-b` options, respectively. Providing these options on the command line overrides the default.

Advanced Commands

- `idf.py app`, `idf.py bootloader`, `idf.py partition_table` can be used to build only the app, bootloader, or partition table from the project as applicable.
- There are matching commands `idf.py app-flash`, etc. to flash only that single part of the project to the ESP32.
- `idf.py -p PORT erase_flash` will use esptool.py to erase the ESP32's entire flash chip.
- `idf.py size` prints some size information about the app. `size-components` and `size-files` are similar commands which print more detailed per-component or per-source-file information, respectively.

The order of multiple `idf.py` commands on the same invocation is not important, they will automatically be executed in the correct order for everything to take effect (ie building before flashing, erasing before flashing, etc.).

Using CMake Directly

idf.py is a wrapper around CMake for convenience. However, you can also invoke CMake directly if you prefer.

When `idf.py` does something, it prints each command that it runs for easy reference. For example, the `idf.py build` command is the same as running these commands in a bash shell (or similar commands for Windows Command Prompt):

```
mkdir -p build
cd build
cmake .. -G Ninja   # or 'Unix Makefiles'
ninja
```

In the above list, the `cmake` command configures the project and generates build files for use with the final build tool. In this case the final build tool is Ninja: running `ninja` actually builds the project.

It's not necessary to run `cmake` more than once. After the first build, you only need to run `ninja` each time. `ninja` will automatically re-invoke `cmake` if the project needs reconfiguration.

If using CMake with `ninja` or `make`, there are also targets for more of the `idf.py` sub-commands - for example running `make menuconfig` or `ninja menuconfig` in the build directory will work the same as `idf.py menuconfig`.

Note: If you're already familiar with CMake, you may find the ESP-IDF CMake-based build system unusual because it wraps a lot of CMake's functionality to reduce boilerplate. See *writing pure CMake components* for some information about writing more "CMake style" components.

Flashing with ninja or make

It's possible to build and flash directly from ninja or make by running a target like:

```
ninja flash
```

Or:

```
make app-flash
```

Available targets are: `flash`, `app-flash` (app only), `bootloader-flash` (bootloader only).

When flashing this way, optionally set the `ESPPORT` and `ESPBAUD` environment variables to specify the serial port and baud rate. You can set environment variables in your operating system or IDE project. Alternatively, set them directly on the command line:

```
ESPPORT=/dev/ttyUSB0 ninja flash
```

Note: Providing environment variables at the start of the command like this is Bash shell Syntax. It will work on Linux and macOS. It won't work when using Windows Command Prompt, but it will work when using Bash-like shells on Windows.

Or:

```
make -j3 app-flash ESPPORT=COM4 ESPBAUD=2000000
```

Note: Providing variables at the end of the command line is `make` syntax, and works for `make` on all platforms.

Using CMake in an IDE

You can also use an IDE with CMake integration. The IDE will want to know the path to the project's `CMakeLists.txt` file. IDEs with CMake integration often provide their own build tools (CMake calls these "generators") to build the source files as part of the IDE.

When adding custom non-build steps like "flash" to the IDE, it is recommended to execute `idf.py` for these "special" commands.

For more detailed information about integrating ESP-IDF with CMake into an IDE, see *Build System Metadata*.

Setting the Python Interpreter

Currently, ESP-IDF only works with Python 2.7. If you have a system where the default `python` interpreter is Python 3.x, this can lead to problems.

If using `idf.py`, running `idf.py` as `python2 $IDF_PATH/tools/idf.py ...` will work around this issue (`idf.py` will tell other Python processes to use the same Python interpreter). You can set up a shell alias or another script to simplify the command.

If using CMake directly, running `cmake -D PYTHON=python2 ...` will cause CMake to override the default Python interpreter.

If using an IDE with CMake, setting the `PYTHON` value as a CMake cache override in the IDE UI will override the default Python interpreter.

To manage the Python version more generally via the command line, check out the tools pyenv or virtualenv. These let you change the default python version.

5.3.3 Example Project

An example project directory tree might look like this:

```
- myProject/
             - CMakeLists.txt
             - sdkconfig
             - components/ - component1/ - CMakeLists.txt
                                         - Kconfig
                                         - src1.c
                           - component2/ - CMakeLists.txt
                                         - Kconfig
                                         - src1.c
                                         - include/ - component2.h
             - main/       - src1.c
                           - src2.c

             - build/
```

This example "myProject" contains the following elements:

- A top-level project CMakeLists.txt file. This is the primary file which CMake uses to learn how to build the project; and may set project-wide CMake variables. It includes the file /tools/cmake/project.cmake which implements the rest of the build system. Finally, it sets the project name and defines the project.
- "sdkconfig" project configuration file. This file is created/updated when `idf.py menuconfig` runs, and holds configuration for all of the components in the project (including ESP-IDF itself). The "sdkconfig" file may or may not be added to the source control system of the project.
- Optional "components" directory contains components that are part of the project. A project does not have to contain custom components of this kind, but it can be useful for structuring reusable code or including third party components that aren't part of ESP-IDF.
- "main" directory is a special "pseudo-component" that contains source code for the project itself. "main" is a default name, the CMake variable `COMPONENT_DIRS` includes this component but you can modify this variable (or set `EXTRA_COMPONENT_DIRS` in the top-level CMakeLists.txt) to look for components in other places. If you have a lot of source files in your project, we recommend grouping most into components instead of putting them all in "main".
- "build" directory is where build output is created. This directory is created by `idf.py` if it doesn't already exist. CMake configures the project and generates interim build files in this directory. Then, after the main build

process is run, this directory will also contain interim object files and libraries as well as final binary output files. This directory is usually not added to source control or distributed with the project source code.

Component directories each contain a component `CMakeLists.txt` file. This file contains variable definitions to control the build process of the component, and its integration into the overall project. See *Component CMakeLists Files* for more details.

Each component may also include a `Kconfig` file defining the *component configuration* options that can be set via `menuconfig`. Some components may also include `Kconfig.projbuild` and `project_include.cmake` files, which are special files for *overriding parts of the project*.

5.3.4 Project CMakeLists File

Each project has a single top-level `CMakeLists.txt` file that contains build settings for the entire project. By default, the project CMakeLists can be quite minimal.

Minimal Example CMakeLists

Minimal project:

```
cmake_minimum_required(VERSION 3.5)

include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(myProject)
```

Mandatory Parts

The inclusion of these four lines, in the order shown above, is necessary for every project:

- `cmake_minimum_required(VERSION 3.5)` tells CMake what version is required to build the project. ESP-IDF is designed to work with CMake 3.5 or newer. This line must be the first line in the CMakeLists.txt file.
- `include($ENV{IDF_PATH}/tools/cmake/project.cmake)` pulls in the rest of the CMake functionality to configure the project, discover all the components, etc.
- `project(myProject)` creates the project itself, and specifies the project name. The project name is used for the final binary output files of the app - ie `myProject.elf`, `myProject.bin`. Only one project can be defined per CMakeLists file.

Optional Project Variables

These variables all have default values that can be overridden for custom behaviour. Look in /tools/cmake/project.cmake for all of the implementation details.

- `COMPONENT_DIRS`: Directories to search for components. Defaults to *${IDF_PATH}/components*, *${PROJECT_PATH}/components*, and `EXTRA_COMPONENT_DIRS`. Override this variable if you don't want to search for components in these places.
- `EXTRA_COMPONENT_DIRS`: Optional list of additional directories to search for components. Paths can be relative to the project directory, or absolute.
- `COMPONENTS`: A list of component names to build into the project. Defaults to all components found in the `COMPONENT_DIRS` directories. Use this variable to "trim down" the project for faster build times. Note that

any component which "requires" another component via COMPONENT_REQUIRES will automatically have it added to this list, so the COMPONENTS list can be very short.

- COMPONENT_REQUIRES_COMMON: A list of components that every component requires. These components are automatically added to every component's COMPONENT_PRIV_REQUIRES list and also the project's COMPONENTS list. By default, this variable is set to the minimal set of core "system" components needed for any ESP-IDF project. Usually, you would not change this variable in your project.

Any paths in these variables can be absolute paths, or set relative to the project directory.

To set these variables, use the cmake set command ie `set(VARIABLE "VALUE")`. The `set()` commands should be placed after the `cmake_minimum(...)` line but before the `include(...)` line.

5.3.5 Component CMakeLists Files

Each project contains one or more components. Components can be part of ESP-IDF, part of the project's own components directory, or added from custom component directories (*see above*).

A component is any directory in the COMPONENT_DIRS list which contains a `CMakeLists.txt` file.

Searching for Components

The list of directories in COMPONENT_DIRS is searched for the project's components. Directories in this list can either be components themselves (ie they contain a *CMakeLists.txt* file), or they can be top-level directories whose sub-directories are components.

When CMake runs to configure the project, it logs the components included in the build. This list can be useful for debugging the inclusion/exclusion of certain components.

Multiple components with the same name

When ESP-IDF is collecting all the components to compile, it will do this in the order specified by COMPONENT_DIRS; by default, this means ESP-IDF's internal components first, then the project's components, and finally any components set in EXTRA_COMPONENT_DIRS. If two or more of these directories contain component sub-directories with the same name, the component in the last place searched is used. This allows, for example, overriding ESP-IDF components with a modified version by copying that component from the ESP-IDF components directory to the project components directory and then modifying it there. If used in this way, the ESP-IDF directory itself can remain untouched.

Minimal Component CMakeLists

The minimal component `CMakeLists.txt` file is as follows:

```
set(COMPONENT_SRCS "foo.c")
set(COMPONENT_ADD_INCLUDEDIRS "include")
register_component()
```

- COMPONENT_SRCS is a (space-separated) list of source files (`*.c`, `*.cpp`, `*.cc`, `*.S`). These source files will be compiled into the component library.
- COMPONENT_ADD_INCLUDEDIRS is a (space-separated) list of directories to add to the global include search path for any component which requires this component, and also the main source files.

- `register_component()` is required to add the component (using the variables set above) to the build. A library with the name of the component will be built and linked into the final app. If this step is skipped (perhaps due to use of a CMake if function or similar), this component will not be part of the build.

Directories are usually specified relative to the `CMakeLists.txt` file itself, although they can be absolute.

See *example component CMakeLists* for more complete component `CMakeLists.txt` examples.

Preset Component Variables

The following component-specific variables are available for use inside component CMakeLists, but should not be modified:

- `COMPONENT_PATH`: The component directory. Evaluates to the absolute path of the directory containing `component.mk`. The component path cannot contain spaces. This is the same as the `CMAKE_CURRENT_SOURCE_DIR` variable.
- `COMPONENT_NAME`: Name of the component. Same as the name of the component directory.

The following variables are set at the project level, but available for use in component CMakeLists:

- `PROJECT_NAME`: Name of the project, as set in project Makefile
- `PROJECT_PATH`: Absolute path of the project directory containing the project Makefile. Same as the `CMAKE_SOURCE_DIR` variable.
- `COMPONENTS`: Names of all components that are included in this build, formatted as a semicolon-delimited CMake list.
- `CONFIG_*`: Each value in the project configuration has a corresponding variable available in make. All names begin with `CONFIG_`. *More information here*.
- `IDF_VER`: Git version of ESP-IDF (produced by `git describe`)

If you modify any of these variables inside `CMakeLists.txt` then this will not prevent other components from building but it may make your component hard to build and/or debug.

- `COMPONENT_ADD_INCLUDEDIRS`: Paths, relative to the component directory, which will be added to the include search path for all other components which require this one. If an include directory is only needed to compile this specific component, add it to `COMPONENT_PRIV_INCLUDEDIRS` instead.
- `COMPONENT_REQUIRES` is a (space-separated) list of components that are required to include this project's header files into other components. If this component has a header file in a `COMPONENT_ADD_INCLUDEDIRS` directory that includes a header from another component, that component should be listed in `COMPONENT_REQUIRES`. Requirements are recursive.

  The `COMPONENT_REQUIRES` list can be empty because some very common components (like newlib for libc, freertos for RTOS functions, etc) are always required by all components. This list is found in the project-level variable `COMPONENT_REQUIRES_COMMON`.

  If a component only requires another component's headers to compile its source files (not for including this component's headers), then these components should be listed in `COMPONENT_PRIV_REQUIRES` instead.

  See *Component Requirements* for more details.

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_PRIV_REQUIRES is a (space-separated) list of components that are required to either compile or link this component's source files. These components' header paths do not propagate to other components which require it, they are only used to compile this component's sources. See *Component Requirements* for more details.
- COMPONENT_SRCS: Paths to individual source files to compile as part of the component. This is the recommended way of adding source files to the build.
- COMPONENT_SRCDIRS: Directory paths, must be relative to the component directory, which will be searched for source files (*.cpp, *.c, *.S). Source files are globbed from the listed directories and compiled as part of the component in place of COMPONENT_SRCS, i.e. setting this will cause COMPONENT_SRCS to be ignored. This can be a convenient way of including source files to the components en masse, but is generally not recommended due to caveats attached to CMake globbing (see *File Globbing & Incremental Builds*).
- COMPONENT_SRCEXCLUDE: Paths to source files to exclude from component. Can be set in conjunction with COMPONENT_SRCDIRS if there is a directory with a large number of source files to include in the component but one or more source files which should not be. Paths can be specified relative to the component directory or absolute.

Note: If you don't set COMPONENT_SRCDIRS or COMPONENT_SRCS, your component won't compile a library but it may still add include paths for use when compiling other components.

Controlling Component Compilation

To pass compiler options when compiling source files belonging to a particular component, use the component_compile_options function:

```
component_compile_options(-Wno-unused-variable)
```

This is a wrapper around the CMake target_compile_options command.

To apply the compilation flags to a single source file, use the CMake set_source_files_properties command:

```
set_source_files_properties(mysrc.c
    PROPERTIES COMPILE_FLAGS
    -Wno-unused-variable
)
```

This can be useful if there is upstream code that emits warnings.

When using these commands, place them after the register_component() line in the component CMakeLists file.

5.3.6 Component Configuration

Each component can also have a Kconfig file, alongside CMakeLists.txt. This contains configuration settings to add to the configuration menu for this component.

These settings are found under the "Component Settings" menu when menuconfig is run.

To create a component Kconfig file, it is easiest to start with one of the Kconfig files distributed with ESP-IDF.

For an example, see *Adding conditional configuration*.

5.3.7 Preprocessor Definitions

The ESP-IDF build system adds the following C preprocessor definitions on the command line:

- `ESP_PLATFORM` — Can be used to detect that build happens within ESP-IDF.
- `IDF_VER` — Defined to a git version string. E.g. `v2.0` for a tagged release or `v1.0-275-g0efaa4f` for an arbitrary commit.

5.3.8 Component Requirements

When compiling each component, the ESP-IDF build system recursively evaluates its components.

Each component's source file is compiled with these include path directories:

- The current component's `COMPONENT_ADD_INCLUDEDIRS` and `COMPONENT_PRIV_INCLUDEDIRS`.
- The `COMPONENT_ADD_INCLUDEDIRS` set by all components in the current component's `COMPONENT_REQUIRES` and `COMPONENT_PRIV_REQUIRES` variables (ie all the current component's public and private dependencies).
- All of the `COMPONENT_REQUIRES` of those components, evaluated recursively (ie all public dependencies of this component's dependencies, recursively expanded).

When writing a component

- `COMPONENT_REQUIRES` should be set to all components whose header files are #included from the *public* header files of this component.
- `COMPONENT_PRIV_REQUIRES` should be set to all components whose header files are #included from *any source files* of this component, unless already listed in `COMPONENT_REQUIRES`. Or any component which is required to be linked in order for this component to function correctly.
- `COMPONENT_REQUIRES` and/or `COMPONENT_PRIV_REQUIRES` should be set before calling `register_component()`.
- The values of `COMPONENT_REQUIRES` and `COMPONENT_PRIV_REQUIRES` should not depend on any configuration choices (`CONFIG_xxx` macros). This is because requirements are expanded before configuration is loaded. Other component variables (like include paths or source files) can depend on configuration choices.
- Not setting either or both `REQUIRES` variables is fine. If the component has no requirements except for the "common" components needed for RTOS, libc, etc (`COMPONENT_REQUIRES_COMMON`) then both variables can be empty or unset.

When creating a project

- By default, every component is included in the build.
- If you set the `COMPONENTS` variable to a minimal list of components used directly by your project, then the build will include: - Components mentioned explicitly in `COMPONENTS`. - Those components' requirements (evaluated recursively). - The "common" components that every component depends on.
- Setting `COMPONENTS` to the minimal list of required components can significantly reduce compile times.

Requirements in the build system implementation

- Very early in the CMake configuration process, the script `expand_requirements.cmake` is run. This script does a partial evaluation of all component CMakeLists.txt files and builds a graph of component requirements (this graph may have cycles). The graph is used to generate a file `component_depends.cmake` in the build directory.
- The main CMake process then includes this file and uses it to determine the list of components to include in the build (internal `BUILD_COMPONENTS` variable).
- Configuration is then evaluated for the components included in the build.
- Each component is included in the build normally and the CMakeLists.txt file is evaluated again to add the component libraries to the build.

5.3.9 Build Process Internals

For full details about CMake and CMake commands, see the CMake v3.5 documentation.

project.cmake contents

When included from a project CMakeLists file, the `project.cmake` file defines some utility modules and global variables and then sets `IDF_PATH` if it was not set in the system environment.

It also defines an overridden custom version of the built-in CMake `project` function. This function is overridden to add all of the ESP-IDF specific project functionality.

project function

The custom `project()` function performs the following steps:

- Evaluates component dependencies and builds the `BUILD_COMPONENTS` list of components to include in the build (see *above*).
- Finds all components in the project (searching `COMPONENT_DIRS` and filtering by `COMPONENTS` if this is set).
- Loads the project configuration from the `sdkconfig` file and generates a `sdkconfig.cmake` file and a `sdkconfig.h` header. These define configuration values in CMake and C/C++, respectively. If the project configuration changes, cmake will automatically be re-run to re-generate these files and re-configure the project.
- Sets the CMAKE_TOOLCHAIN_FILE variable to the ESP-IDF toolchain file with the Xtensa ESP32 toolchain.
- Declare the actual cmake-level project by calling the CMake project function.
- Load the git version. This includes some magic which will automatically re-run CMake if a new revision is checked out in git. See *File Globbing & Incremental Builds*.
- Include `project_include.cmake` files from any components which have them.
- Add each component to the build. Each component CMakeLists file calls `register_component`, calls the CMake add_library function to add a library and then adds source files, compile options, etc.
- Add the final app executable to the build.
- Go back and add inter-component dependencies between components (ie adding the public header directories of each component to each other component).

Browse the /tools/cmake/project.cmake file and supporting functions in /tools/cmake/idf_functions.cmake for more details.

Debugging CMake

Some tips for debugging the ESP-IDF CMake-based build system:

- When CMake runs, it prints quite a lot of diagnostic information including lists of components and component paths.
- Running `cmake -DDEBUG=1` will produce more verbose diagnostic output from the IDF build system.
- Running `cmake` with the `--trace` or `--trace-expand` options will give a lot of information about control flow. See the cmake command line documentation.

Warning On Undefined Variables

By default, `idf.py` passes the `--warn-uninitialized` flag to CMake so it will print a warning if an undefined variable is referenced in the build. This can be very useful to find buggy CMake files.

If you don't want this behaviour, it can be disabled by passing `--no-warnings` to `idf.py`.

Overriding Parts of the Project

project_include.cmake

For components that have build requirements which must be evaluated before any component CMakeLists files are evaluated, you can create a file called `project_include.cmake` in the component directory. This CMake file is included when `project.cmake` is evaluating the entire project.

`project_include.cmake` files are used inside ESP-IDF, for defining project-wide build features such as `esptool.py` command line arguments and the `bootloader` "special app".

Note that `project_include.cmake` isn't necessary for the most common component uses - such as adding include directories to the project, or `LDFLAGS` to the final linking step. These values can be customised via the `CMakeLists.txt` file itself. See *Optional Project Variables* for details.

Take great care when setting variables or targets in a `project_include.cmake` file. As the values are included into the top-level project CMake pass, they can influence or break functionality across all components!

KConfig.projbuild

This is an equivalent to `project_include.cmake` for *Component Configuration* KConfig files. If you want to include configuration options at the top-level of menuconfig, rather than inside the "Component Configuration" sub-menu, then these can be defined in the KConfig.projbuild file alongside the `CMakeLists.txt` file.

Take care when adding configuration values in this file, as they will be included across the entire project configuration. Where possible, it's generally better to create a KConfig file for *Component Configuration*.

Configuration-Only Components

Special components which contain no source files, only `Kconfig.projbuild` and `KConfig`, can have a one-line `CMakeLists.txt` file which calls the function `register_config_only_component()`. This function will

include the component in the project build, but no library will be built *and* no header files will be added to any include paths.

If a CMakeLists.txt file doesn't call `register_component()` or `register_config_only_component()`, it will be excluded from the project entirely. This may sometimes be desirable, depending on the project configuration.

5.3.10 Example Component CMakeLists

Because the build environment tries to set reasonable defaults that will work most of the time, component `CMakeLists.txt` can be very small or even empty (see *Minimal Component CMakeLists*). However, overriding *component variables* is usually required for some functionality.

Here are some more advanced examples of component CMakeLists files.

Adding conditional configuration

The configuration system can be used to conditionally compile some files depending on the options selected in the project configuration.

`Kconfig`:

```
config FOO_ENABLE_BAR
    bool "Enable the BAR feature."
    help
        This enables the BAR feature of the FOO component.
```

`CMakeLists.txt`:

```
set(COMPONENT_SRCS "foo.c" "more_foo.c")

if(CONFIG_FOO_ENABLE_BAR)
    list(APPEND COMPONENT_SRCS "bar.c")
endif(CONFIG_FOO_ENABLE_BAR)
```

This example makes use of the CMake if function and list APPEND function.

This can also be used to select or stub out an implementation, as such:

`Kconfig`:

```
config ENABLE_LCD_OUTPUT
    bool "Enable LCD output."
    help
        Select this if your board has a LCD.

config ENABLE_LCD_CONSOLE
    bool "Output console text to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output debugging output to the lcd

config ENABLE_LCD_PLOT
    bool "Output temperature plots to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output temperature plots
```

`CMakeLists.txt`:

```
if(CONFIG_ENABLE_LCD_OUTPUT)
   set(COMPONENT_SRCS lcd-real.c lcd-spi.c)
else()
   set(COMPONENT_SRCS lcd-dummy.c)
endif()

# We need font if either console or plot is enabled
if(CONFIG_ENABLE_LCD_CONSOLE OR CONFIG_ENABLE_LCD_PLOT)
  list(APPEND COMPONENT_SRCS "font.c")
endif()
```

Source Code Generation

Some components will have a situation where a source file isn't supplied with the component itself but has to be generated from another file. Say our component has a header file that consists of the converted binary data of a BMP file, converted using a hypothetical tool called bmp2h. The header file is then included in as C source file called graphics_lib.c:

```
add_custom_command(OUTPUT logo.h
     COMMAND bmp2h -i ${COMPONENT_PATH}/logo.bmp -o log.h
     DEPENDS ${COMPONENT_PATH}/logo.bmp
     VERBATIM)

add_custom_target(logo DEPENDS logo.h)
add_dependencies(${COMPONENT_NAME} logo)

set_property(DIRECTORY "${COMPONENT_PATH}" APPEND PROPERTY
     ADDITIONAL_MAKE_CLEAN_FILES logo.h)
```

This answer is adapted from the CMake FAQ entry, which contains some other examples that will also work with ESP-IDF builds.

In this example, logo.h will be generated in the current directory (the build directory) while logo.bmp comes with the component and resides under the component path. Because logo.h is a generated file, it should be cleaned when the project is cleaned. For this reason it is added to the ADDITIONAL_MAKE_CLEAN_FILES property.

Note: If generating files as part of the project CMakeLists.txt file, not a component CMakeLists.txt, then use `${PROJECT_PATH}` instead of `${COMPONENT_PATH}` and `${PROJECT_NAME}.elf` instead of `${COMPONENT_NAME}`.)

If a a source file from another component included `logo.h`, then `add_dependencies` would need to be called to add a dependency between the two components, to ensure that the component source files were always compiled in the correct order.

Embedding Binary Data

Sometimes you have a file with some binary or text data that you'd like to make available to your component - but you don't want to reformat the file as C source.

You can set a variable `COMPONENT_EMBED_FILES` in your component's CMakeLists, giving space-delimited names of the files to embed:

```
set(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:

```
set(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.

To embed a file into a project, rather than a component, you can call the function `target_add_binary_data` like this:

```
target_add_binary_data(myproject.elf "main/data.bin" TEXT)
```

Place this line after the `project()` line in your project CMakeLists.txt file. Replace `myproject.elf` with your project name. The final argument can be `TEXT` to embed a null-terminated string, or `BINARY` to embed the content as-is.

For an example of using this technique, see protocols/https_request - the certificate file contents are loaded from the text .pem file at compile time.

Fully Overriding The Component Build Process

Obviously, there are cases where all these recipes are insufficient for a certain component, for example when the component is basically a wrapper around another third-party component not originally intended to be compiled under this build system. In that case, it's possible to forego the ESP-IDF build system entirely by using a CMake feature called ExternalProject. Example component CMakeLists:

```
# External build process for quirc, runs in source dir and
# produces libquirc.a
externalproject_add(quirc_build
    PREFIX ${COMPONENT_PATH}
    SOURCE_DIR ${COMPONENT_PATH}/quirc
    CONFIGURE_COMMAND ""
    BUILD_IN_SOURCE 1
    BUILD_COMMAND make CC=${CMAKE_C_COMPILER} libquirc.a
    INSTALL_COMMAND ""
    )

 # Add libquirc.a to the build process
 #
 add_library(quirc STATIC IMPORTED GLOBAL)
 add_dependencies(quirc quirc_build)

 set_target_properties(quirc PROPERTIES IMPORTED_LOCATION
      ${COMPONENT_PATH}/quirc/libquirc.a)
 set_target_properties(quirc PROPERTIES INTERFACE_INCLUDE_DIRECTORIES
```

(continues on next page)

(continued from previous page)

```
    ${COMPONENT_PATH}/quirc/lib)

set_directory_properties( PROPERTIES ADDITIONAL_MAKE_CLEAN_FILES
    "${COMPONENT_PATH}/quirc/libquirc.a")
```

(The above CMakeLists.txt can be used to create a component named `quirc` that builds the quirc project using its own Makefile.)

- `externalproject_add` defines an external build system.
  - `SOURCE_DIR`, `CONFIGURE_COMMAND`, `BUILD_COMMAND` and `INSTALL_COMMAND` should always be set. `CONFIGURE_COMMAND` can be set to an empty string if the build system has no "configure" step. `INSTALL_COMMAND` will generally be empty for ESP-IDF builds.
  - Setting `BUILD_IN_SOURCE` means the build directory is the same as the source directory. Otherwise you can set `BUILD_DIR`.
  - Consult the ExternalProject documentation for more details about `externalproject_add()`
- The second set of commands adds a library target, which points to the "imported" library file built by the external system. Some properties need to be set in order to add include directories and tell CMake where this file is.
- Finally, the generated library is added to ADDITIONAL_MAKE_CLEAN_FILES. This means `make clean` will delete this library. (Note that the other object files from the build won't be deleted.)

Note: When using an external build process with PSRAM, remember to add `-mfix-esp32-psram-cache-issue` to the C compiler arguments. See *`CONFIG_SPIRAM_CACHE_WORKAROUND`* for details of this flag.

ExternalProject dependencies, clean builds

CMake has some unusual behaviour around external project builds:

- ADDITIONAL_MAKE_CLEAN_FILES only works when "make" is used as the build system. If Ninja or an IDE build system is used, it won't delete these files when cleaning.
- However, the ExternalProject configure & build commands will *always* be re-run after a clean is run.
- Therefore, there are two alternative recommended ways to configure the external build command:
  1. Have the external `BUILD_COMMAND` run a full clean compile of all sources. The build command will be run if any of the dependencies passed to `externalproject_add` with `DEPENDS` have changed, or if this is a clean build (ie any of `idf.py clean`, `ninja clean`, or `make clean` was run.)
  2. Have the external `BUILD_COMMAND` be an incremental build command. Pass the parameter `BUILD_ALWAYS 1` to `externalproject_add`. This means the external project will be built each time a build is run, regardless of dependencies. This is only recommended if the external project has correct incremental build behaviour, and doesn't take too long to run.

The best of these approaches for building an external project will depend on the project itself, its build system, and whether you anticipate needing to frequently recompile the project.

5.3.11 Custom sdkconfig defaults

For example projects or other projects where you don't want to specify a full sdkconfig configuration, but you do want to override some key values from the ESP-IDF defaults, it is possible to create a file `sdkconfig.defaults` in the

project directory. This file will be used when creating a new config from scratch, or when any new config value hasn't yet been set in the `sdkconfig` file.

To override the name of this file, set the `SDKCONFIG_DEFAULTS` environment variable.

5.3.12 Flash arguments

There are some scenarios that we want to flash the target board without IDF. For this case we want to save the built binaries, esptool.py and esptool write_flash arguments. It's simple to write a script to save binaries and esptool.py.

After running a project build, the build directory contains binary output files (`.bin` files) for the project and also the following flashing data files:

- `flash_project_args` contains arguments to flash the entire project (app, bootloader, partition table, PHY data if this is configured).
- `flash_app_args` contains arguments to flash only the app.
- `flash_bootloader_args` contains arguments to flash only the bootloader.

You can pass any of these flasher argument files to `esptool.py` as follows:

```
python esptool.py --chip esp32 write_flash @build/flash_project_args
```

Alternatively, it is possible to manually copy the parameters from the argument file and pass them on the command line.

The build directory also contains a generated file `flasher_args.json` which contains project flash information, in JSON format. This file is used by `idf.py` and can also be used by other tools which need information about the project build.

5.3.13 Building the Bootloader

The bootloader is built by default as part of `idf.py build`, or can be built standalone via `idf.py bootloader`.

The bootloader is a special "subproject" inside /components/bootloader/subproject. It has its own project CMakeLists.txt file and builds separate .ELF and .BIN files to the main project. However it shares its configuration and build directory with the main project.

The subproject is inserted as an external project from the top-level project, by the file /components/bootloader/project_include.cmake. The main build process runs CMake for the subproject, which includes discovering components (a subset of the main components) and generating a bootloader-specific config (derived from the main `sdkconfig`).

5.3.14 Writing Pure CMake Components

The ESP-IDF build system "wraps" CMake with the concept of "components", and helper functions to automatically integrate these components into a project build.

However, underneath the concept of "components" is a full CMake build system. It is also possible to make a component which is pure CMake.

Here is an example minimal "pure CMake" component CMakeLists file for a component named `json`:

```
add_library(json STATIC
cJSON/cJSON.c
cJSON/cJSON_Utils.c)

target_include_directories(json PUBLIC cJSON)
```

- This is actually an equivalent declaration to the IDF `json` component /components/json/CMakeLists.txt.
- This file is quite simple as there are not a lot of source files. For components with a large number of files, the globbing behaviour of ESP-IDF's component logic can make the component CMakeLists style simpler.)
- Any time a component adds a library target with the component name, the ESP-IDF build system will automatically add this to the build, expose public include directories, etc. If a component wants to add a library target with a different name, dependencies will need to be added manually via CMake commands.

5.3.15 File Globbing & Incremental Builds

The preferred way to include source files in an ESP-IDF component is to list them manually in COMPONENT_SRCS:

```
set(COMPONENT_SRCS library/a.c library/b.c platform/platform.c)
```

This preference reflects the CMake best practice of manually listing source files. This could, however, be inconvenient when there are lots of source files to add to the build. The ESP-IDF build system provides an alternative way for specifying source files using `COMPONENT_SRCDIRS`:

```
set(COMPONENT_SRCDIRS library platform)
```

This uses globbing behind the scenes to find source files in the specified directories. Be aware, however, that if a new source file is added and this method is used, then CMake won't know to automatically re-run and this file won't be added to the build.

The trade-off is acceptable when you're adding the file yourself, because you can trigger a clean build or run `idf.py reconfigure` to manually re-run CMake. However, the problem gets harder when you share your project with others who may check out a new version using a source control tool like Git...

For components which are part of ESP-IDF, we use a third party Git CMake integration module (/tools/cmake/third_party/GetGitRevisionDescription.cmake) which automatically re-runs CMake any time the repository commit changes. This means if you check out a new ESP-IDF version, CMake will automatically rerun.

For project components (not part of ESP-IDF), there are a few different options:

- If keeping your project file in Git, ESP-IDF will automatically track the Git revision and re-run CMake if the revision changes.
- If some components are kept in a third git repository (not the project repository or ESP-IDF repository), you can add a call to the `git_describe` function in a component CMakeLists file in order to automatically trigger re-runs of CMake when the Git revision changes.
- If not using Git, remember to manually run `idf.py reconfigure` whenever a source file may change.
- To avoid this problem entirely, use `COMPONENT_SRCS` to list all source files in project components.

The best option will depend on your particular project and its users.

5.3.16 Build System Metadata

For integration into IDEs and other build systems, when CMake runs the build process generates a number of metadata files in the `build/` directory. To regenerate these files, run `cmake` or `idf.py reconfigure` (or any other `idf.`

py build command).

- compile_commands.json is a standard format JSON file which describes every source file which is compiled in the project. A CMake feature generates this file, and many IDEs know how to parse it.
- project_description.json contains some general information about the ESP-IDF project, configured paths, etc.
- flasher_args.json contains esptool.py arguments to flash the project's binary files. There are also flash_*_args files which can be used directly with esptool.py. See *Flash arguments*.
- CMakeCache.txt is the CMake cache file which contains other information about the CMake process, toolchain, etc.
- config/sdkconfig.json is a JSON-formatted version of the project configuration values.
- config/kconfig_menus.json is a JSON-formatted version of the menus shown in menuconfig, for use in external IDE UIs.

JSON Configuration Server

A tool called confserver.py is provided to allow IDEs to easily integrate with the configuration system logic. confserver.py is designed to run in the background and interact with a calling process by reading and writing JSON over process stdin & stdout.

You can run confserver.py from a project via idf.py confserver or ninja confserver, or a similar target triggered from a different build generator.

The config server outputs human-readable errors and warnings on stderr and JSON on stdout. On startup, it will output the full values of each configuration item in the system as a JSON dictionary, and the available ranges for values which are range constrained. The same information is contained in sdkconfig.json:

```
{"version": 1, "values": { "ITEM": "value", "ITEM_2": 1024, "ITEM_3": false }, "ranges
↪" : { "ITEM_2" : [ 0, 32768 ] } }
```

Only visible configuration items are sent. Invisible/disabled items can be parsed from the static kconfig_menus.json file which also contains the menu structure and other metadata (descriptions, types, ranges, etc.)

The Configuration Server will then wait for input from the client. The client passes a request to change one or more values, as a JSON object followed by a newline:

```
{"version": "1", "set": {"SOME_NAME": false, "OTHER_NAME": true } }
```

The Configuration Server will parse this request, update the project sdkconfig file, and return a full list of changes:

```
{"version": 1, "values": {"SOME_NAME": false, "OTHER_NAME": true , "DEPENDS_ON_SOME_
↪NAME": null}}
```

Items which are now invisible/disabled will return value null. Any item which is newly visible will return its newly visible current value.

If the range of a config item changes, due to conditional range depending on another value, then this is also sent:

```
{"version": 1, "values": {"OTHER_NAME": true }, "ranges" : { "HAS_RANGE" : [ 3, 4 ] }␣
↪}
```

If invalid data is passed, an "error" field is present on the object:

```
{"version": 1, "values": {}, "error": ["The following config symbol(s) were not␣
↪visible so were not updated: NOT_VISIBLE_ITEM"]}
```

By default, no config changes are written to the sdkconfig file. Changes are held in memory until a "save" command is sent:

```
{"version": 1, "save": null }
```

To reload the config values from a saved file, discarding any changes in memory, a "load" command can be sent:

```
{"version": 1, "load": null }
```

The value for both "load" and "save" can be a new pathname, or "null" to load/save the previous pathname.

The response to a "load" command is always the full set of config values and ranges, the same as when the server is initially started.

Any combination of "load", "set", and "save" can be sent in a single command and commands are executed in that order. Therefore it's possible to load config from a file, set some config item values and then save to a file in a single command.

Note: The configuration server does not automatically load any changes which are applied externally to the `sdkconfig` file. Send a "load" command or restart the server if the file is externally edited.

Note: The configuration server does not re-run CMake to regenerate other build files or metadata files after `sdkconfig` is updated. This will happen automatically the next time `CMake` or `idf.py` is run.

5.3.17 Migrating from ESP-IDF GNU Make System

Some aspects of the CMake-based ESP-IDF build system are very similar to the older GNU Make-based system. For example, to adapt a `component.mk` file to `CMakeLists.txt` variables like `COMPONENT_ADD_INCLUDEDIRS` and `COMPONENT_SRCDIRS` can stay the same and the syntax only needs changing to CMake syntax.

Automatic Conversion Tool

An automatic project conversion tool is available in /tools/cmake/convert_to_cmake.py. Run this command line tool with the path to a project like this:

```
$IDF_PATH/tools/cmake/convert_to_cmake.py /path/to/project_dir
```

The project directory must contain a Makefile, and GNU Make (`make`) must be installed and available on the PATH.

The tool will convert the project Makefile and any component `component.mk` files to their equivalent `CMakeLists.txt` files.

It does so by running `make` to expand the ESP-IDF build system variables which are set by the build, and then producing equivalent CMakelists files to set the same variables.

The conversion tool is not capable of dealing with complex Makefile logic or unusual targets. These will need to be converted by hand.

No Longer Available in CMake

Some features are significantly different or removed in the CMake-based system. The following variables no longer exist in the CMake-based build system:

- `COMPONENT_BUILD_DIR`: Use `CMAKE_CURRENT_BINARY_DIR` instead.
- `COMPONENT_LIBRARY`: Defaulted to `$(COMPONENT_NAME).a`, but the library name could be overriden by the component. The name of the component library can no longer be overriden by the component.
- `CC`, `LD`, `AR`, `OBJCOPY`: Full paths to each tool from the gcc xtensa cross-toolchain. Use `CMAKE_C_COMPILER`, `CMAKE_C_LINK_EXECUTABLE`, `CMAKE_OBJCOPY`, etc instead. Full list here.
- `HOSTCC`, `HOSTLD`, `HOSTAR`: Full names of each tool from the host native toolchain. These are no longer provided, external projects should detect any required host toolchain manually.
- `COMPONENT_ADD_LDFLAGS`: Used to override linker flags. Use the CMake target_link_libraries command instead.
- `COMPONENT_ADD_LINKER_DEPS`: List of files that linking should depend on. target_link_libraries will usually infer these dependencies automatically. For linker scripts, use the provided custom CMake function `target_linker_scripts`.
- `COMPONENT_SUBMODULES`: No longer used, the build system will automatically enumerate all submodules in the ESP-IDF repository.
- `COMPONENT_EXTRA_INCLUDES`: Used to be an alternative to `COMPONENT_PRIV_INCLUDEDIRS` for absolute paths. Use `COMPONENT_PRIV_INCLUDEDIRS` for all cases now (can be relative or absolute).
- `COMPONENT_OBJS`: Previously, component sources could be specified as a list of object files. Now they can be specified as an list of source files via `COMPONENT_SRCS`.
- `COMPONENT_OBJEXCLUDE`: Has been replaced with `COMPONENT_SRCEXCLUDE`. Specify source files (as absolute paths or relative to component directory), instead.
- `COMPONENT_EXTRA_CLEAN`: Set property `ADDITIONAL_MAKE_CLEAN_FILES` instead but note *CMake has some restrictions around this functionality*.
- `COMPONENT_OWNBUILDTARGET` & `COMPONENT_OWNCLEANTARGET`: Use CMake ExternalProject instead. See *Fully Overriding The Component Build Process* for full details.
- `COMPONENT_CONFIG_ONLY`: Call `register_config_only_component()` instead. See *Configuration-Only Components*.
- `CFLAGS`, `CPPFLAGS`, `CXXFLAGS`: Use equivalent CMake commands instead. See *Controlling Component Compilation*.

No Default Values

The following variables no longer have default values:

- `COMPONENT_SRCDIRS`
- `COMPONENT_ADD_INCLUDEDIRS`

No Longer Necessary

It is no longer necessary to set `COMPONENT_SRCDIRS` if setting `COMPONENT_SRCS` (in fact, in the CMake-based system `COMPONENT_SRCS` is ignored if `COMPONENT_SRCDIRS` is set).

Flashing from make

`make flash` and similar targets still work to build and flash. However, project `sdkconfig` no longer specifies serial port and baud rate. Environment variables can be used to override these. See *Flashing with ninja or make* for more details.

5.4 Error Handling

5.4.1 Overview

Identifying and handling run-time errors is important for developing robust applications. There can be multiple kinds of run-time errors:

- Recoverable errors:
  - Errors indicated by functions through return values (error codes)
  - C++ exceptions, thrown using `throw` keyword
- Unrecoverable (fatal) errors:
  - Failed assertions (using `assert` macro and equivalent methods) and `abort()` calls.
  - CPU exceptions: access to protected regions of memory, illegal instruction, etc.
  - System level checks: watchdog timeout, cache access error, stack overflow, stack smashing, heap corruption, etc.

This guide explains ESP-IDF error handling mechanisms related to recoverable errors, and provides some common error handling patterns.

For instructions on diagnosing unrecoverable errors, see *Fatal Errors*.

5.4.2 Error codes

The majority of ESP-IDF-specific functions use *`esp_err_t`* type to return error codes. *`esp_err_t`* is a signed integer type. Success (no error) is indicated with `ESP_OK` code, which is defined as zero.

Various ESP-IDF header files define possible error codes using preprocessor defines. Usually these defines start with `ESP_ERR_` prefix. Common error codes for generic failures (out of memory, timeout, invalid argument, etc.) are defined in `esp_err.h` file. Various components in ESP-IDF may define additional error codes for specific situations.

For the complete list of error codes, see *Error Code Reference*.

5.4.3 Converting error codes to error messages

For each error code defined in ESP-IDF components, *`esp_err_t`* value can be converted to an error code name using *`esp_err_to_name()`* or *`esp_err_to_name_r()`* functions. For example, passing `0x101` to *`esp_err_to_name()`* will return "ESP_ERR_NO_MEM" string. Such strings can be used in log output to make it easier to understand which error has happened.

Additionally, *`esp_err_to_name_r()`* function will attempt to interpret the error code as a standard POSIX error code, if no matching `ESP_ERR_` value is found. This is done using `strerror_r` function. POSIX error codes (such as `ENOENT`, `ENOMEM`) are defined in `errno.h` and are typically obtained from `errno` variable. In ESP-IDF this variable is thread-local: multiple FreeRTOS tasks have their own copies of `errno`. Functions which set `errno` only modify its value for the task they run in.

This feature is enabled by default, but can be disabled to reduce application binary size. See *CONFIG_ESP_ERR_TO_NAME_LOOKUP*. When this feature is disabled, *esp_err_to_name()* and *esp_err_to_name_r()* are still defined and can be called. In this case, *esp_err_to_name()* will return `UNKNOWN ERROR`, and *esp_err_to_name_r()* will return `Unknown error 0xXXXX(YYYYY)`, where `0xXXXX` and `YYYYY` are the hexadecimal and decimal representations of the error code, respectively.

5.4.4 ESP_ERROR_CHECK macro

`ESP_ERROR_CHECK()` macro serves similar purpose as `assert`, except that it checks *esp_err_t* value rather than a `bool` condition. If the argument of `ESP_ERROR_CHECK()` is not equal *ESP_OK*, then an error message is printed on the console, and `abort()` is called.

Error message will typically look like this:

```
ESP_ERROR_CHECK failed: esp_err_t 0x107 (ESP_ERR_TIMEOUT) at 0x400d1fdf

file: "/Users/user/esp/example/main/main.c" line 20
func: app_main
expression: sdmmc_card_init(host, &card)

Backtrace: 0x40086e7c:0x3ffb4ff0 0x40087328:0x3ffb5010 0x400d1fdf:0x3ffb5030␣
↪0x400d0816:0x3ffb5050
```

Note: If *IDF monitor* is used, addresses in the backtrace will be converted to file names and line numbers.

- The first line mentions the error code as a hexadecimal value, and the identifier used for this error in source code. The latter depends on *CONFIG_ESP_ERR_TO_NAME_LOOKUP* option being set. Address in the program where error has occured is printed as well.
- Subsequent lines show the location in the program where `ESP_ERROR_CHECK()` macro was called, and the expression which was passed to the macro as an argument.
- Finally, backtrace is printed. This is part of panic handler output common to all fatal errors. See *Fatal Errors* for more information about the backtrace.

5.4.5 Error handling patterns

1. Attempt to recover. Depending on the situation, this might mean to retry the call after some time, or attempt to de-initialize the driver and re-initialize it again, or fix the error condition using an out-of-band mechanism (e.g reset an external peripheral which is not responding).

   Example:

   ```
   esp_err_t err;
   do {
       err = sdio_slave_send_queue(addr, len, arg, timeout);
       // keep retrying while the sending queue is full
   } while (err == ESP_ERR_TIMEOUT);
   if (err != ESP_OK) {
       // handle other errors
   }
   ```

2. Propagate the error to the caller. In some middleware components this means that a function must exit with the same error code, making sure any resource allocations are rolled back.

Example:

```
sdmmc_card_t* card = calloc(1, sizeof(sdmmc_card_t));
if (card == NULL) {
    return ESP_ERR_NO_MEM;
}
esp_err_t err = sdmmc_card_init(host, &card);
if (err != ESP_OK) {
    // Clean up
    free(card);
    // Propagate the error to the upper layer (e.g. to notify the user).
    // Alternatively, application can define and return custom error code.
    return err;
}
```

3. Convert into unrecoverable error, for example using `ESP_ERROR_CHECK`. See *ESP_ERROR_CHECK macro* section for details.

   Terminating the application in case of an error is usually undesirable behaviour for middleware components, but is sometimes acceptable at application level.

   Many ESP-IDF examples use `ESP_ERROR_CHECK` to handle errors from various APIs. This is not the best practice for applications, and is done to make example code more concise.

   Example:

```
ESP_ERROR_CHECK(spi_bus_initialize(host, bus_config, dma_chan));
```

5.4.6 C++ Exceptions

Support for C++ Exceptions in ESP-IDF is disabled by default, but can be enabled using `CONFIG_CXX_EXCEPTIONS` option.

Enabling exception handling normally increases application binary size by a few kB. Additionally it may be necessary to reserve some amount of RAM for exception emergency pool. Memory from this pool will be used if it is not possible to allocate exception object from the heap. Amount of memory in the emergency pool can be set using `CONFIG_CXX_EXCEPTIONS_EMG_POOL_SIZE` variable.

If an exception is thrown, but there is no `catch` block, the program will be terminated by `abort` function, and backtrace will be printed. See *Fatal Errors* for more information about backtraces.

5.5 Fatal Errors

5.5.1 Overview

In certain situations, execution of the program can not be continued in a well defined way. In ESP-IDF, these situations include:

- CPU Exceptions: Illegal Instruction, Load/Store Alignment Error, Load/Store Prohibited error, Double Exception.
- System level checks and safeguards:
  - *Interrupt watchdog* timeout
  - *Task watchdog* timeout (only fatal if `CONFIG_TASK_WDT_PANIC` is set)

  - Cache access error
  - Brownout detection event
  - Stack overflow
  - Stack smashing protection check
  - Heap integrity check
- Failed assertions, via `assert`, `configASSERT` and similar macros.

This guide explains the procedure used in ESP-IDF for handling these errors, and provides suggestions on troubleshooting the errors.

5.5.2 Panic Handler

Every error cause listed in the *Overview* will be handled by *panic handler*.

Panic handler will start by printing the cause of the error to the console. For CPU exceptions, the message will be similar to:

```
Guru Meditation Error: Core 0 panic'ed (IllegalInstruction). Exception was unhandled.
```

For some of the system level checks (interrupt watchdog, cache access error), the message will be similar to:

```
Guru Meditation Error: Core 0 panic'ed (Cache disabled but cached memory region␣
↪accessed)
```

In all cases, error cause will be printed in parens. See *Guru Meditation Errors* for a list of possible error causes.

Subsequent behavior of the panic handler can be set using *CONFIG_ESP32_PANIC* configuration choice. The available options are:

- Print registers and reboot (`CONFIG_ESP32_PANIC_PRINT_REBOOT`) — default option.

  This will print register values at the point of the exception, print the backtrace, and restart the chip.
- Print registers and halt (`CONFIG_ESP32_PANIC_PRINT_HALT`)

  Similar to the above option, but halt instead of rebooting. External reset is required to restart the program.
- Silent reboot (`CONFIG_ESP32_PANIC_SILENT_REBOOT`)

  Don't print registers or backtrace, restart the chip immediately.
- Invoke GDB Stub (`CONFIG_ESP32_PANIC_GDBSTUB`)

  Start GDB server which can communicate with GDB over console UART port. See *GDB Stub* for more details.

Behavior of panic handler is affected by two other configuration options.

- If *CONFIG_ESP32_DEBUG_OCDAWARE* is enabled (which is the default), panic handler will detect whether a JTAG debugger is connected. If it is, execution will be halted and control will be passed to the debugger. In this case registers and backtrace are not dumped to the console, and GDBStub / Core Dump functions are not used.
- If *Core Dump* feature is enabled (`CONFIG_ESP32_ENABLE_COREDUMP_TO_FLASH` or `CONFIG_ESP32_ENABLE_COREDUMP_TO_UART` options), then system state (task stacks and registers) will be dumped either to Flash or UART, for later analysis.

The following diagram illustrates panic handler behavior:

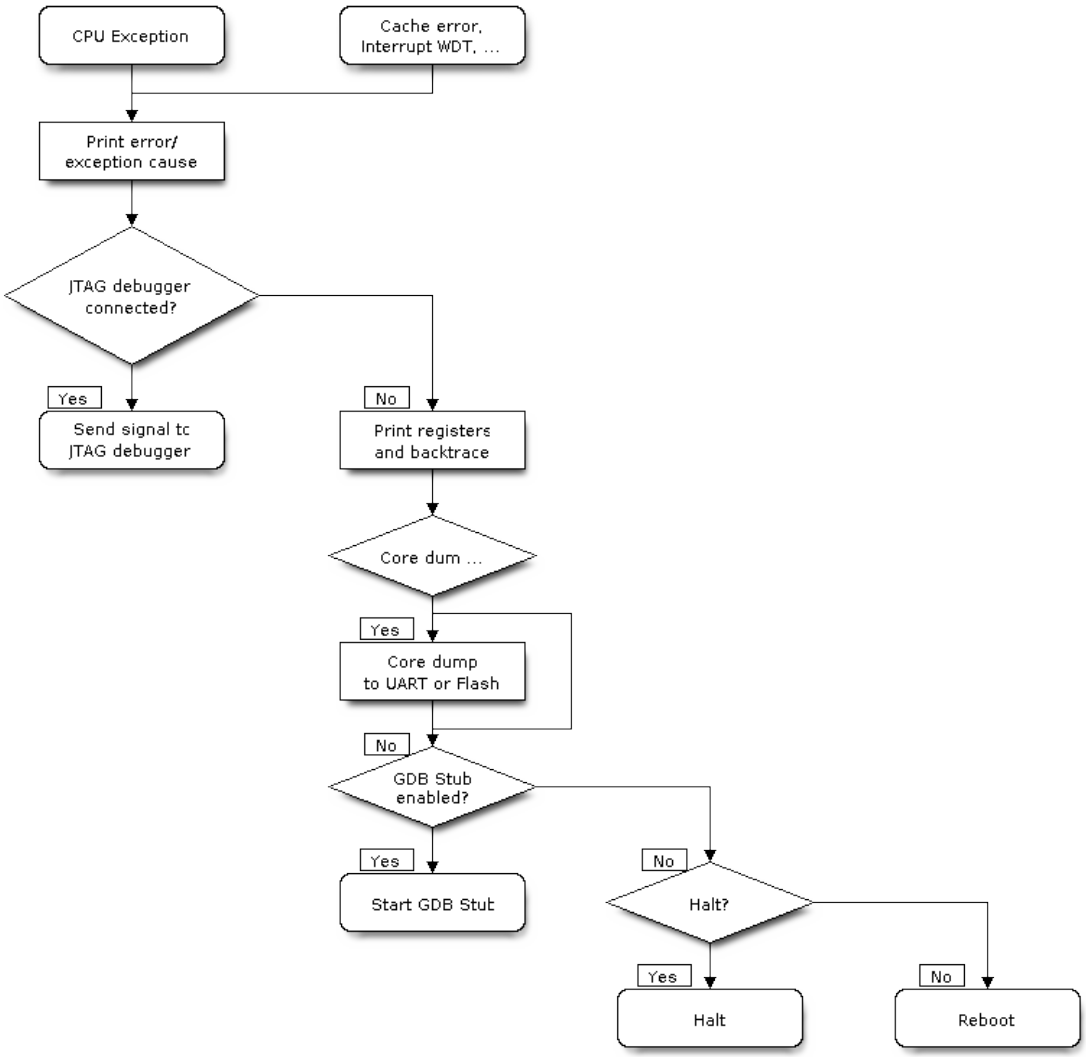


Fig. 1: Panic Handler Flowchart

5.5.3 Register Dump and Backtrace

Unless `CONFIG_ESP32_PANIC_SILENT_REBOOT` option is enabled, panic handler prints some of the CPU registers, and the backtrace, to the console:

```
Core 0 register dump:
PC      : 0x400e14ed  PS      : 0x00060030  A0      : 0x800d0805  A1      : 0x3ffb5030
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x00000001  A5      : 0x3ffb50dc
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x00000000  A9      : 0x3ffb5000
A10     : 0x00000000  A11     : 0x3ffb2bac  A12     : 0x40082d1c  A13     : 0x06ff1ff8
A14     : 0x3ffb7078  A15     : 0x00000000  SAR     : 0x00000014  EXCCAUSE: 0x0000001d
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  : 0xffffffff

Backtrace: 0x400e14ed:0x3ffb5030 0x400d0802:0x3ffb5050
```

Register values printed are the register values in the exception frame, i.e. values at the moment when CPU exception or other fatal error has occured.

Register dump is not printed if the panic handler was executed as a result of an `abort()` call.

In some cases, such as interrupt watchdog timeout, panic handler may print additional CPU registers (EPC1-EPC4) and the registers/backtrace of the code running on the other CPU.

Backtrace line contains PC:SP pairs, where PC is the Program Counter and SP is Stack Pointer, for each stack frame of the current task. If a fatal error happens inside an ISR, the backtrace may include PC:SP pairs both from the task which was interrupted, and from the ISR.

If *IDF Monitor* is used, Program Counter values will be converted to code locations (function name, file name, and line number), and the output will be annotated with additional lines:

```
Core 0 register dump:
PC      : 0x400e14ed  PS      : 0x00060030  A0      : 0x800d0805  A1      : 0x3ffb5030
0x400e14ed: app_main at /Users/user/esp/example/main/main.cpp:36

A2      : 0x00000000  A3      : 0x00000001  A4      : 0x00000001  A5      : 0x3ffb50dc
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x00000000  A9      : 0x3ffb5000
A10     : 0x00000000  A11     : 0x3ffb2bac  A12     : 0x40082d1c  A13     : 0x06ff1ff8
0x40082d1c: _calloc_r at /Users/user/esp/esp-idf/components/newlib/syscalls.c:51

A14     : 0x3ffb7078  A15     : 0x00000000  SAR     : 0x00000014  EXCCAUSE: 0x0000001d
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  : 0xffffffff

Backtrace: 0x400e14ed:0x3ffb5030 0x400d0802:0x3ffb5050
0x400e14ed: app_main at /Users/user/esp/example/main/main.cpp:36

0x400d0802: main_task at /Users/user/esp/esp-idf/components/esp32/cpu_start.c:470
```

To find the location where a fatal error has happened, look at the lines which follow the "Backtrace" line. Fatal error location is the top line, and subsequent lines show the call stack.

5.5.4 GDB Stub

If `CONFIG_ESP32_PANIC_GDBSTUB` option is enabled, panic handler will not reset the chip when fatal error happens. Instead, it will start GDB remote protocol server, commonly referred to as GDB Stub. When this happens, GDB instance running on the host computer can be instructed to connect to the ESP32 UART port.

If *IDF Monitor* is used, GDB is started automatically when GDB Stub prompt is detected on the UART. The output would look like this:

```
Entering gdb stub now.
$T0b#e6GNU gdb (crosstool-NG crosstool-ng-1.22.0-80-gff1f415) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_apple-darwin16.3.0 --target=xtensa-
↪esp32-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /Users/user/esp/example/build/example.elf...done.
Remote debugging using /dev/cu.usbserial-31301
0x400e1b41 in app_main ()
    at /Users/user/esp/example/main/main.cpp:36
36      *((int*) 0) = 0;
(gdb)
```

GDB prompt can be used to inspect CPU registers, local and static variables, and arbitrary locations in memory. It is not possible to set breakpoints, change PC, or continue execution. To reset the program, exit GDB and perform external reset: Ctrl-T Ctrl-R in IDF Monitor, or using external reset button on the development board.

5.5.5 Guru Meditation Errors

This section explains the meaning of different error causes, printed in parens after `Guru Meditation Error: Core panic'ed` message.

Note: See Wikipedia article for historical origins of "Guru Meditation".

IllegalInstruction

This CPU exception indicates that the instruction which was executed was not a valid instruction. Most common reasons for this error include:

- FreeRTOS task function has returned. In FreeRTOS, if task function needs to terminate, it should call `vTaskDelete()` function and delete itself, instead of returning.
- Failure to load next instruction from SPI flash. This usually happens if:
  - Application has reconfigured SPI flash pins as some other function (GPIO, UART, etc.). Consult Hardware Design Guidelines and the Datasheet for the chip or module for details about SPI flash pins.
  - Some external device was accidentally connected to SPI flash pins, and has interfered with communication between ESP32 and SPI flash.

InstrFetchProhibited

This CPU exception indicates that CPU could not load an instruction because the the address of the instruction did not belong to a valid region in instruction RAM or ROM.

Usually this means an attempt to call a function pointer, which does not point to valid code. `PC` (Program Counter) register can be used as an indicator: it will be zero or will contain garbage value (not `0x4xxxxxxx`).

LoadProhibited, StoreProhibited

This CPU exception happens when application attempts to read from or write to an invalid memory location. The address which was written/read is found in `EXCVADDR` register in the register dump. If this address is zero, it usually means that application attempted to dereference a NULL pointer. If this address is close to zero, it usually means that application attempted to access member of a structure, but the pointer to the structure was NULL. If this address is something else (garbage value, not in `0x3fxxxxxx` - `0x6xxxxxxx` range), it likely means that the pointer used to access the data was either not initialized or was corrupted.

IntegerDivideByZero

Application has attempted to do integer division by zero.

LoadStoreAlignment

Application has attempted to read or write memory location, and address alignment did not match load/store size. For example, 32-bit load can only be done from 4-byte aligned address, and 16-bit load can only be done from a 2-byte aligned address.

LoadStoreError

Application has attempted to do a 8- or 16- bit load/store from a memory region which only supports 32-bit loads/stores. For example, dereferencing a `char*` pointer which points into intruction memory will result in such an error.

Unhandled debug exception

This will usually be followed by a message like:

```
Debug exception reason: Stack canary watchpoint triggered (task_name)
```

This error indicates that application has written past the end of the stack of `task_name` task. Note that not every stack overflow is guaranteed to trigger this error. It is possible that the task writes to stack beyond the stack canary location, in which case the watchpoint will not be triggered.

Interrupt wdt timeout on CPU0 / CPU1

Indicates that interrupt watchdog timeout has occured. See *Watchdogs* for more information.

Cache disabled but cached memory region accessed

In some situations ESP-IDF will temporarily disable access to external SPI Flash and SPI RAM via caches. For example, this happens with spi_flash APIs are used to read/write/erase/mmap regions of SPI Flash. In these situations, tasks are suspended, and interrupt handlers not registered with `ESP_INTR_FLAG_IRAM` are disabled. Make sure that any interrupt handlers registered with this flag have all the code and data in IRAM/DRAM. Refer to the *SPI flash API documentation* for more details.

5.5.6 Other Fatal Errors

Brownout

ESP32 has a built-in brownout detector, which is enabled by default. Brownout detector can trigger system reset if supply voltage goes below safe level. Brownout detector can be configured using *CONFIG_BROWNOUT_DET* and *CONFIG_BROWNOUT_DET_LVL_SEL* options. When brownout detector triggers, the following message is printed:

```
Brownout detector was triggered
```

Chip is reset after the message is printed.

Note that if supply voltage is dropping at a fast rate, only part of the message may be seen on the console.

Corrupt Heap

ESP-IDF heap implementation contains a number of run-time checks of heap structure. Additional checks ("Heap Poisoning") can be enabled in menuconfig. If one of the checks fails, message similar to the following will be printed:

```
CORRUPT HEAP: Bad tail at 0x3ffe270a. Expected 0xbaad5678 got 0xbaac5678
assertion "head != NULL" failed: file "/Users/user/esp/esp-idf/components/heap/multi_
↪heap_poisoning.c", line 201, function: multi_heap_free
abort() was called at PC 0x400dca43 on core 0
```

Consult *Heap Memory Debugging* documentation for further information.

Stack Smashing

Stack smashing protection (based on GCC `-fstack-protector*` flags) can be enabled in ESP-IDF using *CONFIG_STACK_CHECK_MODE* option. If stack smashing is detected, message similar to the following will be printed:

```
Stack smashing protect failure!

abort() was called at PC 0x400d2138 on core 0

Backtrace: 0x4008e6c0:0x3ffc1780 0x4008e8b7:0x3ffc17a0 0x400d2138:0x3ffc17c0␣
↪0x400e79d5:0x3ffc17e0 0x400e79a7:0x3ffc1840 0x400e79df:0x3ffc18a0␣
↪0x400e2235:0x3ffc18c0 0x400e1916:0x3ffc18f0 0x400e19cd:0x3ffc1910␣
↪0x400e1a11:0x3ffc1930 0x400e1bb2:0x3ffc1950 0x400d2c44:0x3ffc1a80
0
```

The backtrace should point to the function where stack smashing has occured. Check the function code for unbounded access to local arrays.

5.6 Deep Sleep Wake Stubs

ESP32 supports running a "deep sleep wake stub" when coming out of deep sleep. This function runs immediately as soon as the chip wakes up - before any normal initialisation, bootloader, or ESP-IDF code has run. After the wake stub runs, the SoC can go back to sleep or continue to start ESP-IDF normally.

Deep sleep wake stub code is loaded into "RTC Fast Memory" and any data which it uses must also be loaded into RTC memory. RTC memory regions hold their contents during deep sleep.

5.6.1 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 unintiailised and have random contents. The wake stub can use other RAM for temporary storage, but the contents will be overwritten when the SoC goes back to sleep or starts ESP-IDF.
- RTC memory must include any read-only data (.rodata) used by the stub.
- Data in RTC memory is initialised whenever the SoC restarts, except when waking from deep sleep. When waking from deep sleep, the values which were present before going to sleep are kept.
- Wake stub code is a part of the main esp-idf app. During normal running of esp-idf, functions can call the wake stub functions or access RTC memory. It is as if these were regular parts of the app.

5.6.2 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.

5.6.3 Loading Code Into RTC Memory

Wake stub code must be resident in RTC Fast Memory. This can be done in one of two ways.

The first way is to use the `RTC_IRAM_ATTR` attribute to place a function into RTC memory:

```
void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    // Add additional functionality here
}
```

The second way is to place the function into any source file whose name starts with `rtc_wake_stub`. Files names `rtc_wake_stub*` have their contents automatically put into RTC memory by the linker.

The first way is simpler for very short and simple code, or for source files where you want to mix "normal" and "RTC" code. The second way is simpler when you want to write longer pieces of code for RTC memory.

5.6.4 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, respectivley) 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.

5.7 ESP32 Core Dump

5.7.1 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 meaningfull

5.7.2 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.

5.7.3 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 requrements for partition name. It can be choosen 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 partirion size should be at least 20 + max tasks number x (12 + TCB size + max task stack size) bytes.

The example of generic command to analyze core dump from flash is: *espcoredump.py -p </path/to/serial/port> info_corefile </path/to/program/elf/file>* or *espcoredump.py -p </path/to/serial/port> dbg_corefile </path/to/program/elf/file>*

5.7.4 Print core dump to UART

When this option is selected base64-encoded core dumps are printed on UART upon system panic. In this case user should save core dump text body to some file manually and then run the following command: *espcoredump.py info_corefile -t b64 -c </path/to/saved/base64/text> </path/to/program/elf/file>* or *espcoredump.py dbg_corefile -t b64 -c </path/to/saved/base64/text> </path/to/program/elf/file>*

Base64-encoded body of core dump will be between the following header and footer:

```
================= CORE DUMP START =================
<body of base64-encoded core dump, save it to file on disk>
================= CORE DUMP END ===================
```

The *CORE DUMP START* and *CORE DUMP END* lines must not be included in core dump text file.

5.7.5 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".

5.8 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 *Limitations of Flash Encryption*) and understand the implications of enabling flash encryption.

Note: Flash encryption is only supported when using the default GNU Make build system. The CMake build system preview in ESP-IDF v3.x does not support flash encryption.

5.8.1 Background

- The contents of the flash are encrypted using AES-256. The flash encryption key is stored in efuse internal to the chip, and is (by default) protected from software access.
- 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 since NVS library is not directly compatible with flash encryption. Refer to *NVS Encryption* for more details.
- 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.
- By default, the Efuse Block 1 Coding Scheme is "None" and a 256 bit key is stored in this block. On some ESP32s, the Coding Scheme is set to 3/4 Encoding (CODING_SCHEME efuse has value 1) and a 192 bit key must be stored in this block. See ESP32 Technical Reference Manual section 20.3.1.3 *System Parameter coding_scheme* for more details. The algorithm operates on a 256 bit key in all cases, 192 bit keys are extended by repeating some bits (*details*). The coding scheme is shown in the `Features` line when `esptool.py` connects to the chip, or in the `espefuse.py summary` output.
- 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*.

5.8.2 Storing Encrypted Data

Aside from encrypting the firmware binary, the app may need to store some sensitive data in an encrypted form. For example, in a filesystem or NVS data partition.

The recommended way to do this is to use *NVS Encryption* .

Alternatively, it is possible to use the *Wear Levelling feature* with an encrypted partition, if the "encrypted" flag is set on the partition. This allows, for example, a FATFS partition to be stored encrypted in flash.

The following are **not suitable** and will store data where an attacker with physical access can read it out:

- Custom efuse fields (these can be write protected against modification but not read protected if the app needs to read them)
- SPIFFS (SPIFFS is optimized for the read and write behavior of NOR flash, so it's not possible to encrypt this filesystem)

5.8.3 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.

5.8.4 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 from the perspective of Flash Encryption. It is up to the library to provide encryption feature if required. Refer to *NVS Encryption* for more details.

Writing Encrypted Flash

Where possible, we recommend using the partition write function `esp_partition_write`. When using this function, data will only be encrypted when writing to encrypted partitions. Data will be written to other partitions unencrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

The `esp_spi_flash_write` function will write data when the write_encrypted parameter is set to true. Otherwise, data will be written unencrypted.

The ROM function `esp_rom_spiflash_write_encrypted` will write encrypted data to flash, the ROM function `SPIWrite` will write unencrypted to flash. (these function are not supported in esp-idf apps).

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.)

5.8.5 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 --keylen 256 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.

5.8.6 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 continously, 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.

5.8.7 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 runningon the device, use flash encryption in combination with *Secure Boot*.

5.8.8 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.

5.8.9 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:

```
espefuse.py --port PORT burn_efuse DISABLE_DL_DECRYPT
espefuse.py --port PORT write_protect_efuse DISABLE_DL_ENCRYPT
```

(Note that all 3 of these efuses are disabled via one write protect bit, so write protecting one will write protect all of them. For this reason, it's necessary to set any bits before write-protecting.)

IMPORTANT: Write protecting these efuses to keep them unset is not currently very useful, as `esptool.py` does not support 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 it the value is zero. If this efuse is set to zero, no bits in the flash encryption key are tweaked and the flash encryption algorithm is equivalent to AES ECB mode.

5.8.10 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.
- The main flash encryption key is stored in efuse (BLOCK1) and by default is protected from further writes or software readout.
- AES-256 key size is 256 bits (32 bytes), read from efuse block 1. The hardware AES engine uses the key in reversed byte order to the order stored in the efuse block. - If `CODING_SCHEME` efuse is set to 0 (default "None" Coding Scheme) then the efuse key block is 256 bits and the key is stored as-is (in reversed byte order). - If `CODING_SCHEME` efuse is set to 1 (3/4 Encoding) then the efuse key block is 192 bits (in reversed byte order), so overall entropy is reduced. The hardware flash encryption still operates on a 256-bit key, after being read (and un-reversed), the key is extended by as `key = key[0:255] + key[64:127]`.
- AES algorithm is used inverted in flash encryption, so the flash encryption "encrypt" operation is AES decrypt and the "decrypt" operation is AES encrypt. This is for performance reasons and does not alter the effectiveness of the algorithm.
- Each 32 byte block (two adjacent 16 byte AES blocks) is encrypted with a unique key. The key is derived from the main flash encryption key in efuse, XORed with the offset of this block in the flash (a "key tweak").
- The specific tweak depends on the setting of `FLASH_CRYPT_CONFIG` efuse. This is a 4 bit efuse, where each bit enables XORing of a particular range of the key bits:
  - Bit 1, bits 0-66 of the key are XORed.
  - Bit 2, bits 67-131 of the key are XORed.
  - Bit 3, bits 132-194 of the key are XORed.
  - Bit 4, bits 195-256 of the key are XORed.

  It is recommended that `FLASH_CRYPT_CONFIG` is always left to set the default value *0xF*, so that all key bits are XORed with the block offset. See *Setting FLASH_CRYPT_CONFIG* for details.
- The high 19 bits of the block offset (bit 5 to bit 23) are XORed with the main flash encryption key. This range is chosen for two reasons: the maximum flash size is 16MB (24 bits), and each block is 32 bytes so the least significant 5 bits are always zero.
- There is a particular mapping from each of the 19 block offset bits to the 256 bits of the flash encryption key, to determine which bit is XORed with which. See the variable `_FLASH_ENCRYPTION_TWEAK_PATTERN` in the `espsecure.py` source code for the complete mapping.
- To see the full flash encryption algorithm implemented in Python, refer to the *_flash_encryption_operation()* function in the `espsecure.py` source code.

5.9 ESP-IDF FreeRTOS SMP Changes

5.9.1 Overview

The vanilla FreeRTOS is designed to run on a single core. However the ESP32 is dual core containing a Protocol CPU (known as **CPU 0** or **PRO_CPU**) and an Application CPU (known as **CPU 1** or **APP_CPU**). The two cores are identical in practice and share the same memory. This allows the two cores to run tasks interchangeably between them.

The ESP-IDF FreeRTOS is a modified version of vanilla FreeRTOS which supports symmetric multiprocessing (SMP). ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0. This guide outlines the major differences between vanilla FreeRTOS and ESP-IDF FreeRTOS. The API reference for vanilla FreeRTOS can be found via http://www.freertos.org/a00106.html

For information regarding features that are exclusive to ESP-IDF FreeRTOS, see *ESP-IDF FreeRTOS Additions*.

Backported Features: Although ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0, a number of FreeRTOS v9.0.0 features have been backported to ESP-IDF.

Tasks and Task Creation: Use `xTaskCreatePinnedToCore()` or `xTaskCreateStaticPinnedToCore()` to create tasks in ESP-IDF FreeRTOS. The last parameter of the two functions is `xCoreID`. This parameter specifies which core the task is pinned to. Acceptable values are `0` for **PRO_CPU**, `1` for **APP_CPU**, or `tskNO_AFFINITY` which allows the task to run on both.

Round Robin Scheduling: The ESP-IDF FreeRTOS scheduler will skip tasks when implementing Round-Robin scheduling between multiple tasks in the Ready state that are of the same priority. To avoid this behavior, ensure that those tasks either enter a blocked state, or are distributed across a wider range of priorities.

Scheduler Suspension: Suspending the scheduler in ESP-IDF FreeRTOS will only affect the scheduler on the the calling core. In other words, calling `vTaskSuspendAll()` on **PRO_CPU** will not prevent **APP_CPU** from scheduling, and vice versa. Use critical sections or semaphores instead for simultaneous access protection.

Tick Interrupt Synchronicity: Tick interrupts of **PRO_CPU** and **APP_CPU** are not synchronized. Do not expect to use `vTaskDelay()` or `vTaskDelayUntil()` as an accurate method of synchronizing task execution between the two cores. Use a counting semaphore instead as their context switches are not tied to tick interrupts due to preemption.

Critical Sections & Disabling Interrupts: In ESP-IDF FreeRTOS, critical sections are implemented using mutexes. Entering critical sections involve taking a mutex, then disabling the scheduler and interrupts of the calling core. However the other core is left unaffected. If the other core attemps to take same mutex, it will spin until the calling core has released the mutex by exiting the critical section.

Floating Point Aritmetic: The ESP32 supports hardware acceleration of single precision floating point arithmetic (`float`). However the use of hardware acceleration leads to some behavioral restrictions in ESP-IDF FreeRTOS. Therefore, tasks that utilize `float` will automatically be pinned to a core if not done so already. Furthermore, `float` cannot be used in interrupt service routines.

Task Deletion: Task deletion behavior has been backported from FreeRTOS v9.0.0 and modified to be SMP compatible. Task memory will be freed immediately when `vTaskDelete()` is called to delete a task that is not currently running and not pinned to the other core. Otherwise, freeing of task memory will still be delegated to the Idle Task.

Thread Local Storage Pointers & Deletion Callbacks: ESP-IDF FreeRTOS has backported the Thread Local Storage Pointers (TLSP) feature. However the extra feature of Deletion Callbacks has been added. Deletion callbacks are called automatically during task deletion and are used to free memory pointed to by TLSP. Call `vTaskSetThreadLocalStoragePointerAndDelCallback()` to set TLSP and Deletion Callbacks.

Configuring ESP-IDF FreeRTOS: Several aspects of ESP-IDF FreeRTOS can be configured using `make meunconfig` such as running ESP-IDF in Unicore Mode, or configuring the number of Thread Local Storage Pointers each task will have.

5.9.2 Backported Features

The following features have been backported from FreeRTOS v9.0.0 to ESP-IDF.

Static Alocation

This feature has been backported from FreeRTOS v9.0.0 to ESP-IDF. The `CONFIG_SUPPORT_STATIC_ALLOCATION` option must be enabled in *menuconfig* in order for static allocation functions to be available. Once enabled, the following functions can be called...

- `xTaskCreateStatic()` (see *Backporting Notes* below)
- `xQueueCreateStatic`
- `xSemaphoreCreateBinaryStatic`
- `xSemaphoreCreateCountingStatic`
- `xSemaphoreCreateMutexStatic`
- `xSemaphoreCreateRecursiveMutexStatic`
- `xTimerCreateStatic()` (see *Backporting Notes* below)
- `xEventGroupCreateStatic()`

Other Features

- `vTaskSetThreadLocalStoragePointer()` (see *Backporting Notes* below)
- `pvTaskGetThreadLocalStoragePointer()` (see *Backporting Notes* below)
- `vTimerSetTimerID()`
- `xTimerGetPeriod()`
- `xTimerGetExpiryTime()`
- `pcQueueGetName()`
- `uxSemaphoreGetCount`

Backporting Notes

1) `xTaskCreateStatic()` has been made SMP compatible in a similar fashion to `xTaskCreate()` (see *Tasks and Task Creation*). Therefore `xTaskCreateStaticPinnedToCore()` can also be called.

2) Although vanilla FreeRTOS allows the Timer feature's daemon task to be statically allocated, the daemon task is always dynamically allocated in ESP-IDF. Therefore `vApplicationGetTimerTaskMemory` **does not** need to be defined when using statically allocated timers in ESP-IDF FreeRTOS.

3) The Thread Local Storage Pointer feature has been modified in ESP-IDF FreeRTOS to include Deletion Callbacks (see *Thread Local Storage Pointers & Deletion Callbacks*). Therefore the function `vTaskSetThreadLocalStoragePointerAndDelCallback()` can also be called.

5.9.3 Tasks and Task Creation

Tasks in ESP-IDF FreeRTOS are designed to run on a particular core, therefore two new task creation functions have been added to ESP-IDF FreeRTOS by appending `PinnedToCore` to the names of the task creation functions in vanilla FreeRTOS. The vanilla FreeRTOS functions of *`xTaskCreate()`* and *`xTaskCreateStatic()`* have led to the addition of *`xTaskCreatePinnedToCore()`* and *`xTaskCreateStaticPinnedToCore()`* in ESP-IDF FreeRTOS (see *Backported Features*).

For more details see freertos/task.c

The ESP-IDF FreeRTOS task creation functions are nearly identical to their vanilla counterparts with the exception of the extra parameter known as `xCoreID`. This parameter specifies the core on which the task should run on and can be one of the following values.

- `0` pins the task to **PRO_CPU**
- `1` pins the task to **APP_CPU**
- `tskNO_AFFINITY` allows the task to be run on both CPUs

For example `xTaskCreatePinnedToCore(tsk_callback, "APP_CPU Task", 1000, NULL, 10, NULL, 1)` creates a task of priority 10 that is pinned to **APP_CPU** with a stack size of 1000 bytes. It should be noted that the `uxStackDepth` parameter in vanilla FreeRTOS specifies a task's stack depth in terms of the number of words, whereas ESP-IDF FreeRTOS specifies the stack depth in terms of bytes.

Note that the vanilla FreeRTOS functions *`xTaskCreate()`* and *`xTaskCreateStatic()`* have been defined in ESP-IDF FreeRTOS as inline functions which call *`xTaskCreatePinnedToCore()`* and *`xTaskCreateStaticPinnedToCore()`* respectively with `tskNO_AFFINITY` as the `xCoreID` value.

Each Task Control Block (TCB) in ESP-IDF stores the `xCoreID` as a member. Hence when each core calls the scheduler to select a task to run, the `xCoreID` member will allow the scheduler to determine if a given task is permitted to run on the core that called it.

5.9.4 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.

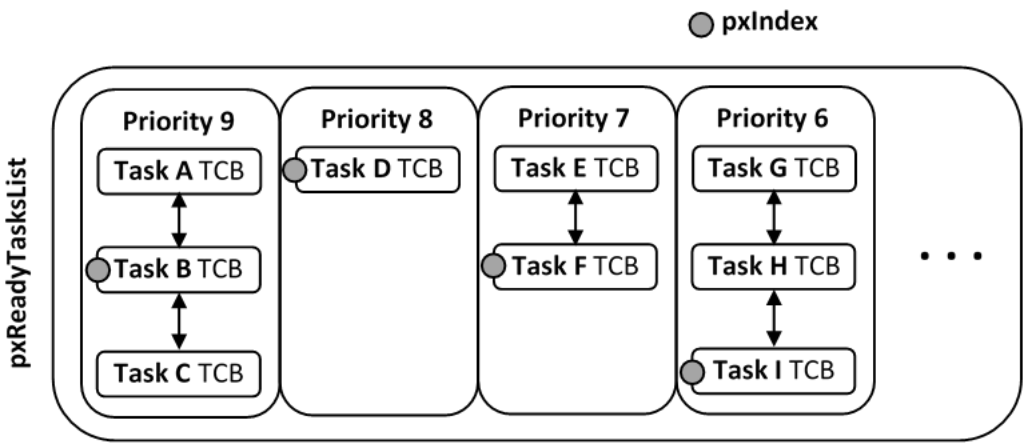


Fig. 2: Illustration of FreeRTOS Ready Task List Data Structure

Each linked list also contains a `pxIndex` which points to the last TCB returned when the list was queried. This index allows the `vTaskSwitchContext()` to start traversing the list at the TCB immediately after `pxIndex` hence implementing Round Robin Scheduling between tasks of the same priority.

In ESP-IDF FreeRTOS, the Ready Tasks List is shared between cores hence `pxReadyTasksList` will contain tasks pinned to different cores. When a core calls the scheduler, it is able to look at the `xCoreID` member of each TCB in the list to determine if a task is allowed to run on calling the core. The ESP-IDF FreeRTOS `pxReadyTasksList` is illustrated below.

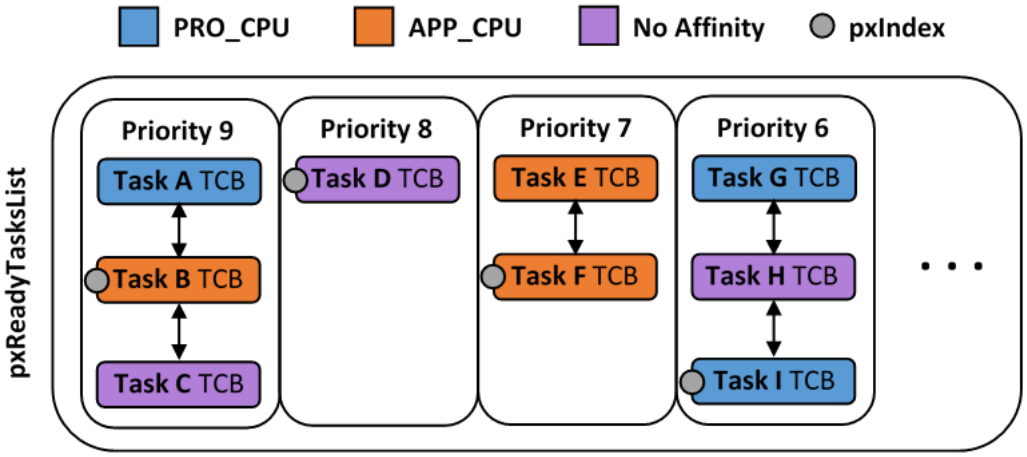


Fig. 3: Illustration of FreeRTOS Ready Task List Data Structure in ESP-IDF

Therefore when **PRO_CPU** calls the scheduler, it will only consider the tasks in blue or purple. Whereas when **APP_CPU** calls the scheduler, it will only consider the tasks in orange or purple.

Although each TCB has an `xCoreID` in ESP-IDF FreeRTOS, the linked list of each priority only has a single `pxIndex`. Therefore when the scheduler is called from a particular core and traverses the linked list, it will skip all TCBs pinned to the other core and point the pxIndex at the selected task. If the other core then calls the scheduler, it will traverse the linked list starting at the TCB immediately after `pxIndex`. Therefore, TCBs skipped on the previous scheduler call from the other core would not be considered on the current scheduler call. This issue is demonstrated in the following illustration.

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.

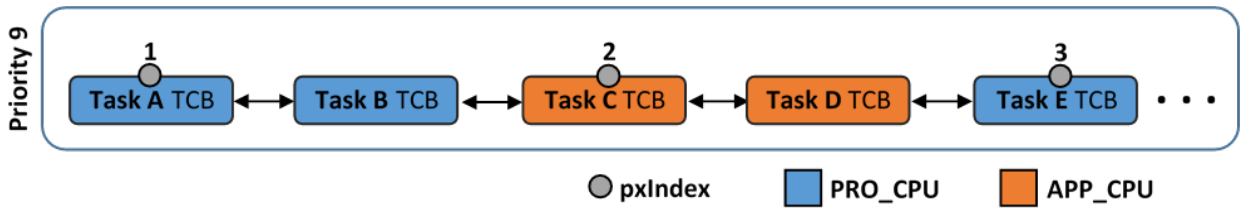


Fig. 4: Illustration of pxIndex behavior in ESP-IDF FreeRTOS

1) **PRO_CPU** calls the scheduler and selects Task A to run, hence moves `pxIndex` to point to Task A

2) **APP_CPU** calls the scheduler and starts traversing from the task after `pxIndex` which is Task B. However Task B is not selected to run as it is not pinned to **APP_CPU** hence it is skipped and Task C is selected instead. `pxIndex` now points to Task C

3) **PRO_CPU** calls the scheduler and starts traversing from Task D. It skips Task D and selects Task E to run and points `pxIndex` to Task E. Notice that Task B isn't traversed because it was skipped the last time **APP_CPU** called the scheduler to traverse the list.

4) The same situation with Task D will occur if **APP_CPU** calls the scheduler again as `pxIndex` now points to Task E

One solution to the issue of task skipping is to ensure that every task will enter a blocked state so that they are removed from the Ready Task List. Another solution is to distribute tasks across multiple priorities such that a given priority will not be assigned multiple tasks that are pinned to different cores.

Scheduler Suspension

In vanilla FreeRTOS, suspending the scheduler via *`vTaskSuspendAll()`* will prevent calls of `vTaskSwitchContext` from context switching until the scheduler has been resumed with *`xTaskResumeAll()`*. However servicing ISRs are still permitted. Therefore any changes in task states as a result from the current running task or 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.

5.9.5 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.

5.9.6 Floating Point Aritmetic

The ESP32 supports hardware acceleration of single precision floating point arithmetic (`float`) via Floating Point Units (FPU, also known as coprocessors) attached to each core. The use of the FPUs imposes some behavioral restrictions on ESP-IDF FreeRTOS.

ESP-IDF FreeRTOS implements Lazy Context Switching for FPUs. In other words, the state of a core's FPU registers are not immediately saved when a context switch occurs. Therefore, tasks that utilize `float` must be pinned to a particular core upon creation. If not, ESP-IDF FreeRTOS will automatically pin the task in question to whichever core the task was running on upon the task's first use of `float`. Likewise due to Lazy Context Switching, interrupt service routines must also not use `float`.

ESP32 does not support hardware acceleration for double precision floating point arithmetic (`double`). Instead `double` is implemented via software hence the behavioral restrictions with regards to `float` do not apply to `double`. Note that due to the lack of hardware acceleration, `double` operations may consume significantly larger amount of CPU time in comparison to `float`.

5.9.7 Task Deletion

FreeRTOS task deletion prior to v9.0.0 delegated the freeing of task memory entirely to the Idle Task. Currently, the freeing of task memory will occur immediately (within `vTaskDelete()`) if the task being deleted is not currently running or is not pinned to the other core (with respect to the core `vTaskDelete()` is called on). TLSP deletion callbacks will also run immediately if the same conditions are met.

However, calling `vTaskDelete()` to delete a task that is either currently running or pinned to the other core will still result in the freeing of memory being delegated to the Idle Task.

5.9.8 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 to *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.

`CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS` in menuconfig can be used to configure the number TLSP and Deletion Callbacks a TCB will have.

For more details see *FreeRTOS API reference*.

5.9.9 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*

`CONFIG_FREERTOS_UNICORE` will run ESP-IDF FreeRTOS only on **PRO_CPU**. Note that this is **not equivalent to running vanilla FreeRTOS**. Behaviors of multiple components in ESP-IDF will be modified such as esp32/cpu_start.c. For more details regarding the effects of running ESP-IDF FreeRTOS on a single core, search for occurences of `CONFIG_FREERTOS_UNICORE` in the ESP-IDF components.

`CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS` will define the number of Thread Local Storage Pointers each task will have in ESP-IDF FreeRTOS.

CONFIG_SUPPORT_STATIC_ALLOCATION will enable the backported functionality of *xTaskCreateStaticPinnedToCore()* in ESP-IDF FreeRTOS

CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION will trigger a halt in particular functions in ESP-IDF FreeRTOS which have not been fully tested in an SMP context.

5.10 Thread Local Storage

5.10.1 Overview

Thread-local storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread. ESP-IDF provides three ways to make use of such variables:

- *FreeRTOS Native API*: ESP-IDF FreeRTOS native API.
- *Pthread API*: ESP-IDF's pthread API.
- *C11 Standard*: C11 standard introduces special keyword to declare variables as thread local.

5.10.2 FreeRTOS Native API

The ESP-IDF FreeRTOS provides the following API to manage thread local variables:

- *vTaskSetThreadLocalStoragePointer()*
- *pvTaskGetThreadLocalStoragePointer()*
- *vTaskSetThreadLocalStoragePointerAndDelCallback()*

In this case maximum number of variables that can be allocated is limited by `configNUM_THREAD_LOCAL_STORAGE_POINTERS` macro. Variables are kept in the task control block (TCB) and accessed by their index. Note that index 0 is reserved for ESP-IDF internal uses. Using that API user can allocate thread local variables of an arbitrary size and assign them to any number of tasks. Different tasks can have different sets of TLS variables. If size of the variable is more then 4 bytes then user is responsible for allocating/deallocating memory for it. Variable's deallocation is initiated by FreeRTOS when task is deleted, but user must provide function (callback) to do proper cleanup.

5.10.3 Pthread API

The ESP-IDF provides the following pthread API to manage thtread local variables:

- `pthread_key_create()`
- `pthread_key_delete()`
- `pthread_getspecific()`
- `pthread_setspecific()`

This API has all benefits of the one above, but eliminates some its limits. The number of variables is limited only by size of available memory on the heap. Due to the dynamic nature this API introduces additional performance overhead compared to the native one.

5.10.4 C11 Standard

The ESP-IDF FreeRTOS supports thread local variables according to C11 standard (ones specified with `__thread` keyword). For details on this GCC feature please see https://gcc.gnu.org/onlinedocs/gcc-5.5.0/gcc/Thread-Local.html#Thread-Local. Storage for that kind of variables is allocated on the task's stack. Note that area for all such variables in the program will be allocated on the stack of every task in the system even if that task does not use such variables at all. For example ESP-IDF system tasks (like `ipc`, `timer` tasks etc.) will also have that extra stack space allocated. So this feature should be used with care. There is a tradeoff: C11 thread local variables are quite handy to use in programming and can be accessed using just a few Xtensa instructions, but this benefit goes with the cost of additional stack usage for all tasks in the system. Due to static nature of variables allocation all tasks in the system have the same sets of C11 thread local variables.

5.11 High-Level Interrupts

The Xtensa architecture has support for 32 interrupts, divided over 8 levels, plus an assortment of exceptions. On the ESP32, the interrupt mux allows most interrupt sources to be routed to these interrupts using the *interrupt allocator*. Normally, interrupts will be written in C, but ESP-IDF allows high-level interrupts to be written in assembly as well, allowing for very low interrupt latencies.

5.11.1 Interrupt Levels

Level	Symbol	Remark
1	N/A	Exception and level 0 interrupts. Handled by ESP-IDF
2-3	N/A	Medium level interrupts. Handled by ESP-IDF
4	xt_highint4	Normally used by ESP-IDF debug logic
5	xt_highint5	Free to use
NMI	xt_nmi	Free to use
dbg	xt_debugexception	Debug exception. Called on e.g. a BREAK instruction.

Using these symbols is done by creating an assembly file (suffix .S) and defining the named symbols, like this:

```
    .section .iram1,"ax"
    .global     xt_highint5
    .type       xt_highint5,@function
    .align      4
xt_highint5:
    ... your code here
    rsr     a0, EXCSAVE_5
    rfi     5
```

For a real-life example, see the components/esp32/panic_highint_hdl.S file; the panic handler iunterrupt is implemented there.

5.11.2 Notes

- Do not call C code from a high-level interrupt; because these interupts still run in critical sections, this can cause crashes. (The panic handler interrupt does call normal C code, but this is OK because there is no intention of returning to the normal code flow afterwards.)

- Make sure your assembly code gets linked in. If the interrupt handler symbol is the only symbol the rest of the code uses from this file, the linker will take the default ISR instead and not link the assembly file into the final project. To get around this, in the assembly file, define a symbol, like this:

```
    .global ld_include_my_isr_file
ld_include_my_isr_file:
```

(The symbol is called `ld_include_my_isr_file` here but can have any arbitrary name not defined anywhere else.) Then, in the component.mk, add this file as an unresolved symbol to the ld command line arguments:

```
COMPONENT_ADD_LDFLAGS := -u ld_include_my_isr_file
```

This should cause the linker to always include a file defining `ld_include_my_isr_file`, causing the ISR to always be linked in.

- High-level interrupts can be routed and handled using esp_intr_alloc and associated functions. The handler and handler arguments to esp_intr_alloc must be NULL, however.
- In theory, medium priority interrupts could also be handled in this way. For now, ESP-IDF does not support this.

5.12 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 from *Eclipse* as well as from *Command Line*.

Building OpenOCD from Sources Procedure to build OpenOCD from sources for *Windows*, *Linux* and *MacOS* operating systems.

Tips and Quirks This section provides collection of tips and quirks related JTAG debugging of ESP32 with OpenOCD and GDB.

5.12.1 Introduction

The ESP32 has two powerful Xtensa cores, allowing for a great deal of variety of program architectures. The FreeRTOS OS that comes with ESP-IDF is capable of multi-core preemptive multithreading, allowing for an intuitive way of writing software.

The downside of the ease of programming is that debugging without the right tools is harder: figuring out a bug that is caused by two threads, running even simultaneously on two different CPU cores, can take a long time when all you

have are printf statements. A better and in many cases quicker way to debug such problems is by using a debugger, connected to the processors over a debug port.

Espressif has ported OpenOCD to support the ESP32 processor and the multicore FreeRTOS, which will be the foundation of most ESP32 apps, and has written some tools to help with features OpenOCD does not support natively.

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB under Linux, Windows and MacOS. Except for OS specific installation procedures, the s/w user interface and use procedures are the same across all supported operating systems.

Note: Screenshots presented in this document have been made for Eclipse Neon 3 running on Ubuntu 16.04 LTE. There may be some small differences in what a particular user interface looks like, depending on whether you are using Windows, MacOS or Linux and / or a different release of Eclipse.

5.12.2 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.

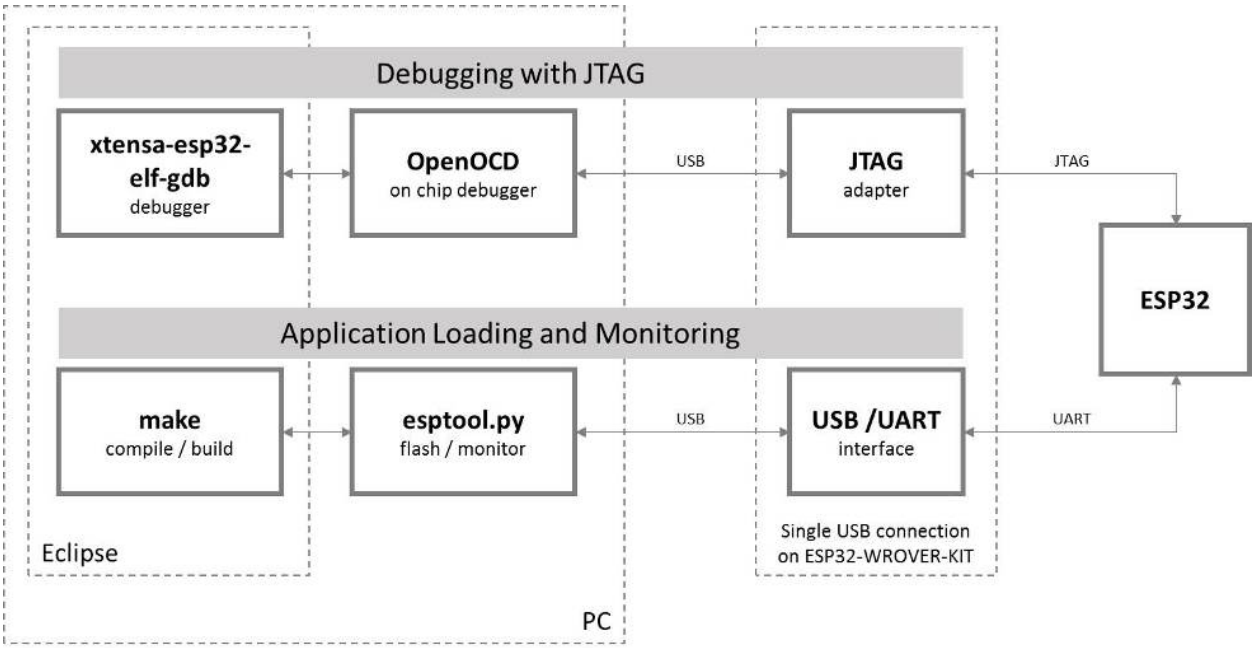


Fig. 5: JTAG debugging - overview diagram

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.

5.12.3 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.

5.12.4 Setup of OpenOCD

This step covers installation of OpenOCD binaries. If you like to build OpenOCS 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.

Set up OpenOCD for Windows

Set up OpenOCD

OpenOCD for Windows is available for download from Github:

https://github.com/espressif/openocd-esp32/releases

Download latest release archive with *win32* in its name, for example *openocd-esp32-macos-0.10.0-win32-20180418.zip*.

Extract the downloaded file in `~/esp/` directory.

```
cd ~/esp unzip /c/Users/<user>/Downloads/openocd-esp32-win32-<version>.zip
```

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 *Set up 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*.

Set up OpenOCD for Linux

Set up OpenOCD

OpenOCD for 64-bit Linux is available for download from Github:

https://github.com/espressif/openocd-esp32/releases

Download latest release archive with *linux64* in its name, for example *openocd-esp32-linux64-0.10.0-esp32-20180418.tar.gz*.

Extract the downloaded file in `~/esp/` directory:

```
cd ~/esp
tar -xzf ~/Downloads/openocd-esp32-linux64-<version>.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 *Set up 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*.

Set up OpenOCD for MacOS

Install libusb

Use Homebrew or Macports to install *libusb* package.

Set up OpenOCD

OpenOCD for MacOS is available for download from Github:

https://github.com/espressif/openocd-esp32/releases

Download latest release archive with *macos* in its name, for example *openocd-esp32-macos-0.10.0-esp32-20180418.tar.gz*.

Extract the downloaded file in `~/esp/` directory:

```
cd ~/esp
tar -xzf ~/Downloads/openocd-esp32-macos-<version>.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 *Set up 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.

Windows	Linux	Mac OS

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*.

5.12.5 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 (v6xxxxx)". To do so, select "Dual RS232-HS (Interface 0) and reinstall attached driver to the "WinUSB (v6xxxxx)", see picture above.

Note: Do not change the second device "Dual RS232-HS (Interface 1)". It is routed to ESP32's serial port (UART) used for upload of application to ESP32's flash.

Now 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.

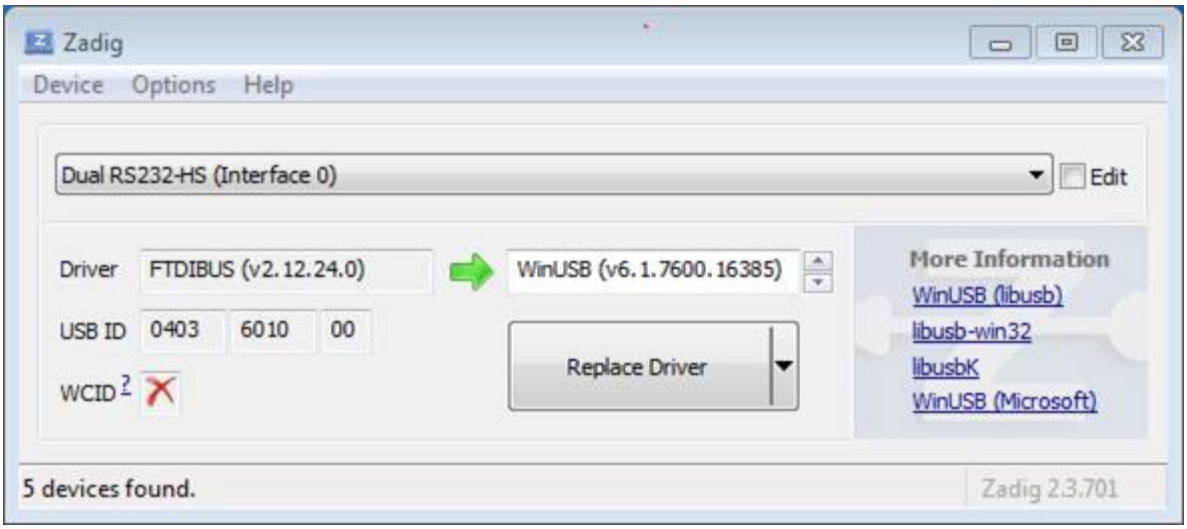


Fig. 6: Configuration of JTAG USB driver in Zadig tool

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 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.

5.12.6 Launching Debugger

The toolchain for ESP32 features GNU Debugger, in short GDB. It is available with other toolchain programs under filename `xtensa-esp32-elf-gdb`. GDB can be called and operated directly from command line in a terminal. Another option is to call it from within IDE (like Eclipse, Visual Studio Code, etc.) and operate indirectly with help of GUI instead of typing commands in a terminal.

Both options of using debugger are discussed under links below.

- *Eclipse*
- *Command Line*

It is recommended to first check if debugger works from *Command Line* and then move to using *Eclipse*.

5.12.7 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.

5.12.8 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
```

5.12.9 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*

5.12.10 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 `flushregs` (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.

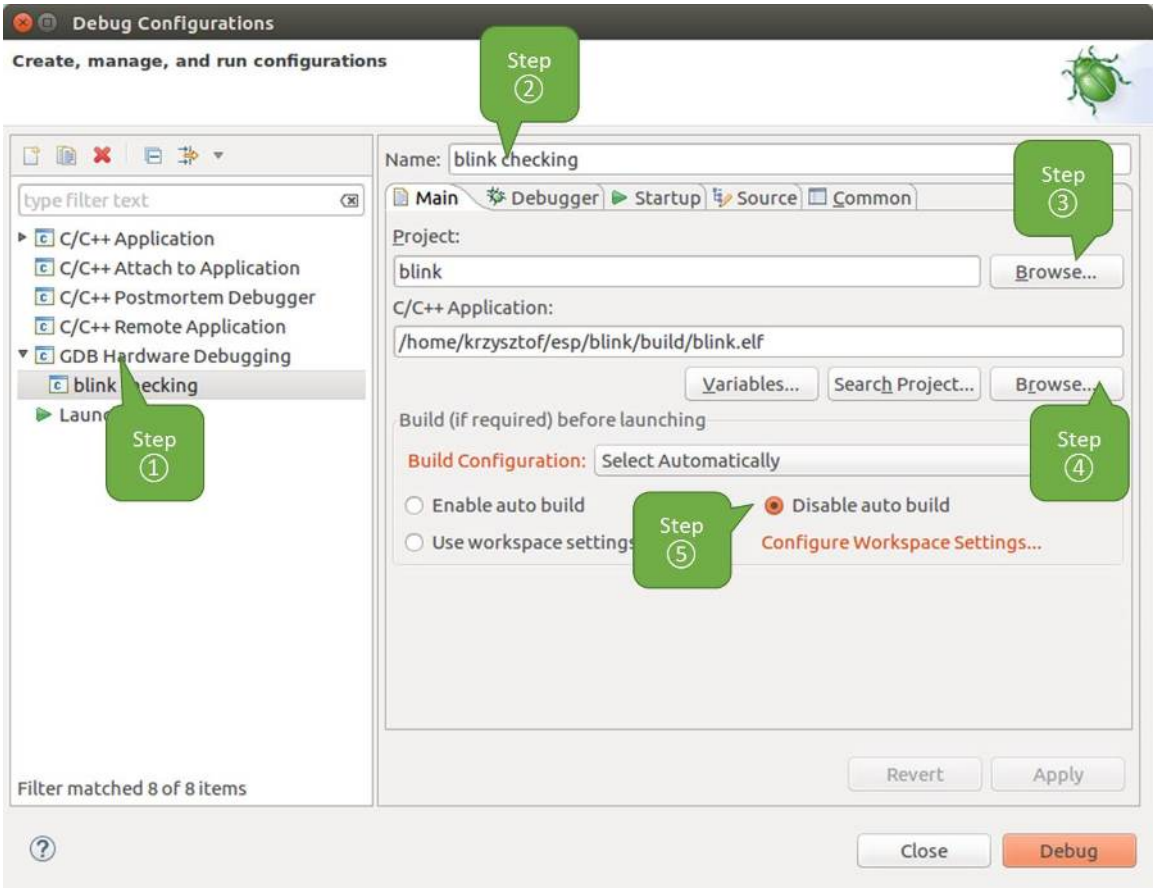


Fig. 7: Configuration of GDB Hardware Debugging - Main tab

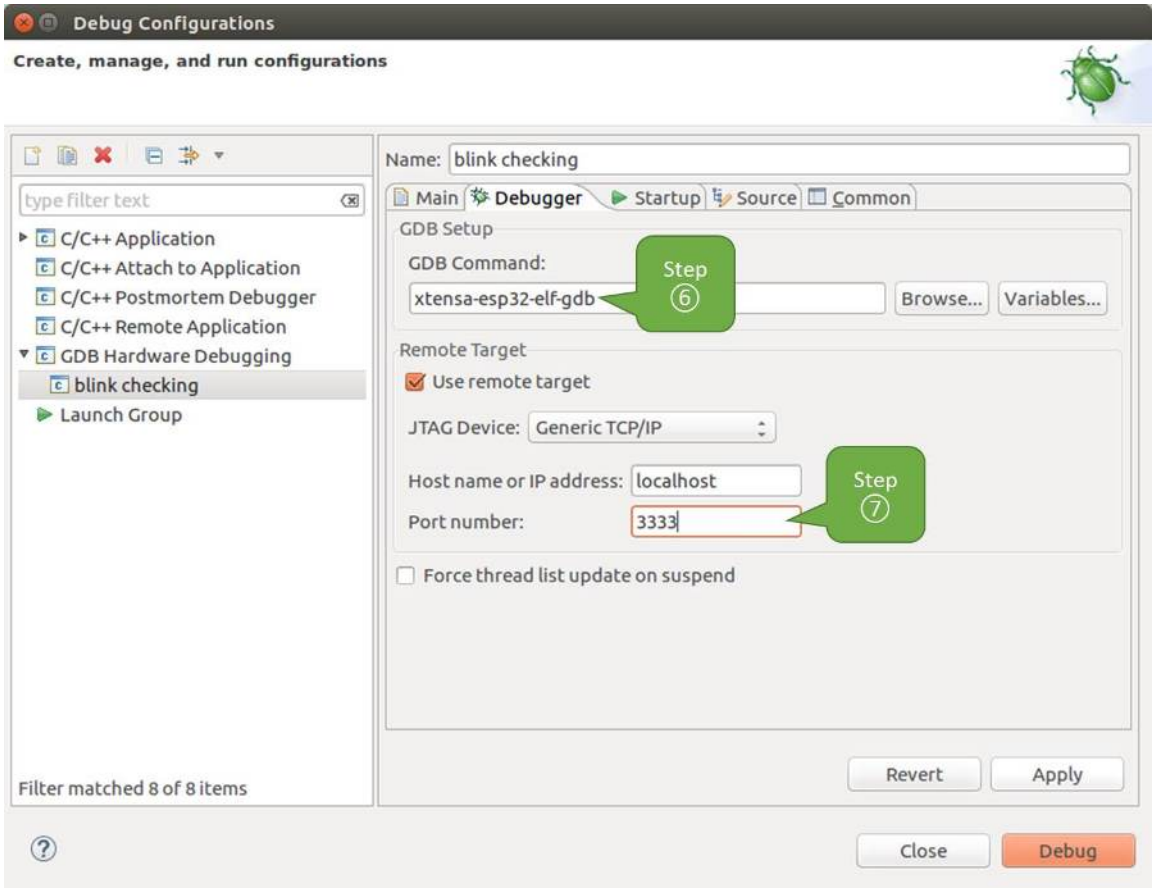


Fig. 8: Configuration of GDB Hardware Debugging - Debugger tab

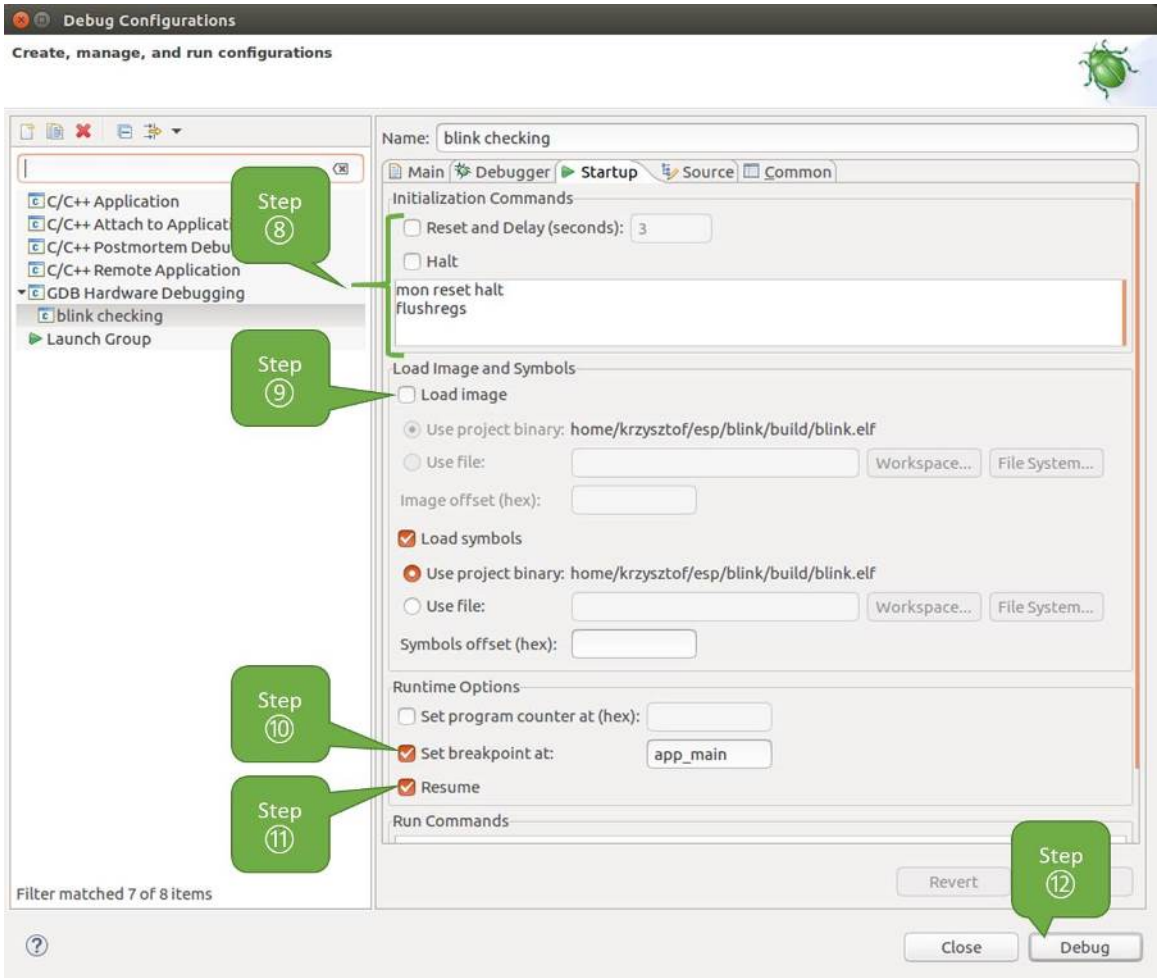


Fig. 9: Configuration of GDB Hardware Debugging - Startup tab

12. If you previously completed *Configuring ESP32 Target* steps described above, so the target is running and ready to talk to debugger, go right to debugging by pressing "Debug" button. Otherwise press "Apply" to save changes, go back to *Configuring ESP32 Target* and return here to start debugging.

Once all 1 - 12 configuration steps are satisfied, the new Eclipse perspective called "Debug" will open as shown on example picture below.

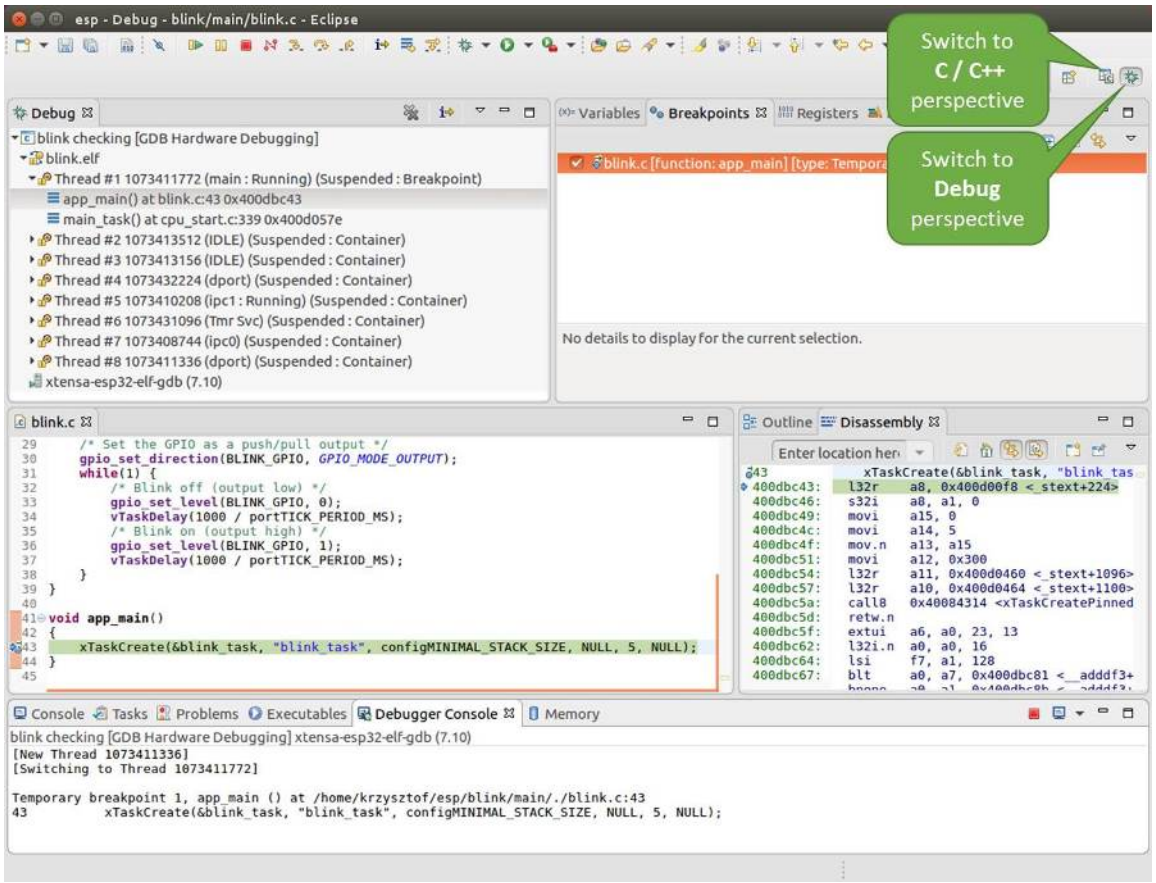


Fig. 10: Debug Perspective in Eclipse

If you are not quite sure how to use GDB, check *Eclipse* example debugging session in section *Debugging Examples*.

Command Line

1. To be able start debugging session, the target should be up and running. If not done already, complete steps described under *Configuring ESP32 Target*.
2. Open a new terminal session and go to directory that contains project for debugging, e.g.

```
cd ~/esp/blink
```

3. When launching a debugger, you will need to provide couple of configuration parameters and commands. Instead of entering them one by one in command line, create a configuration file and name it `gdbinit`:

```
target remote :3333
mon reset halt
```

(continues on next page)

(continued from previous page)

```
flushregs
thb app_main
c
```

Save this file in current directory.

For more details what's inside `gdbinit` file, see *What is the meaning of debugger's startup commands?*

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 was reset (pwrstat=0x5F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x5000004B (active)    APP_CPU: PC=0x00000000
esp32: target state: halted
esp32: Core was reset (pwrstat=0x1F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x40000400 (active)    APP_CPU: PC=0x40000400
esp32: target state: halted
Hardware assisted breakpoint 1 at 0x400db717: file /home/user-name/esp/blink/main/
↪./blink.c, line 43.
0x0:    0x00000000
Target halted. PRO_CPU: PC=0x400DB717 (active)    APP_CPU: PC=0x400D10D8
[New Thread 1073428656]
[New Thread 1073413708]
[New Thread 1073431316]
[New Thread 1073410672]
[New Thread 1073408876]
[New Thread 1073432196]
[New Thread 1073411552]
```

(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.

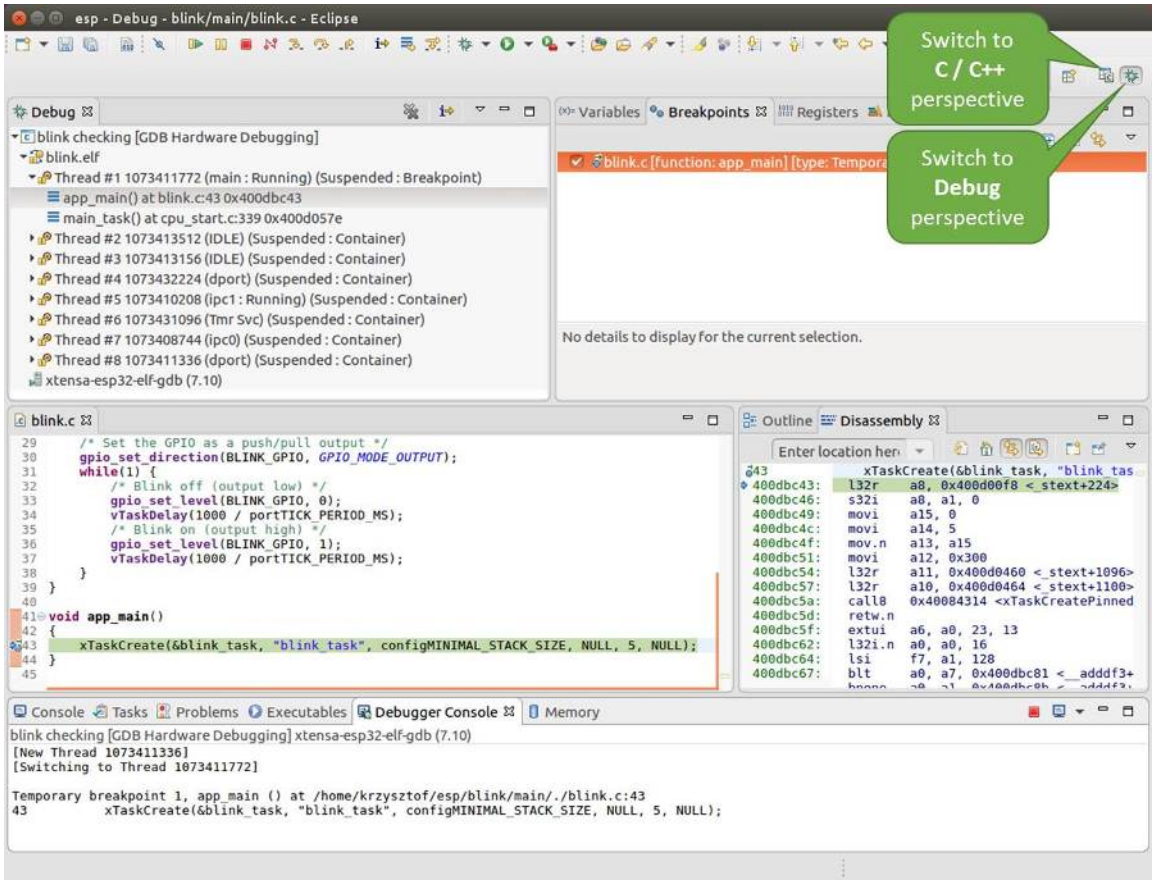


Fig. 11: Debug Perspective in Eclipse

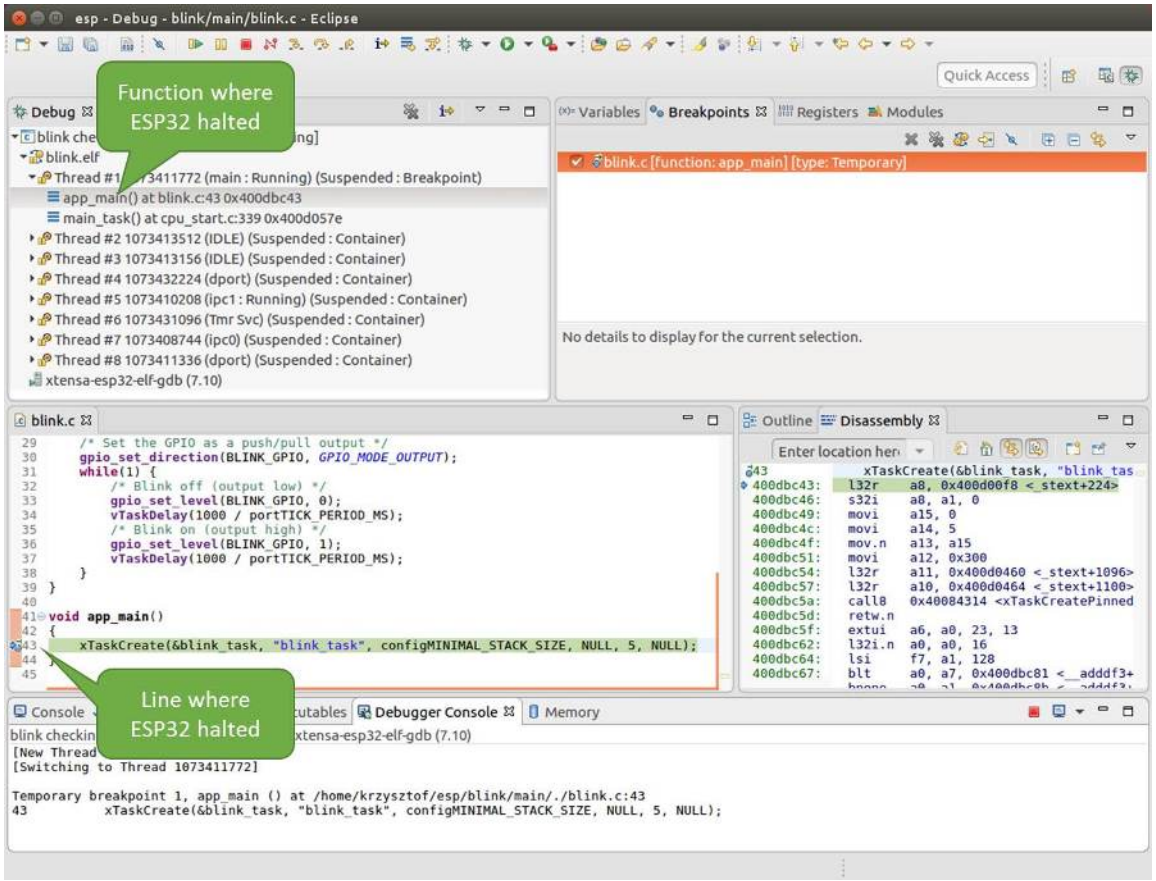


Fig. 12: Target halted during debugging

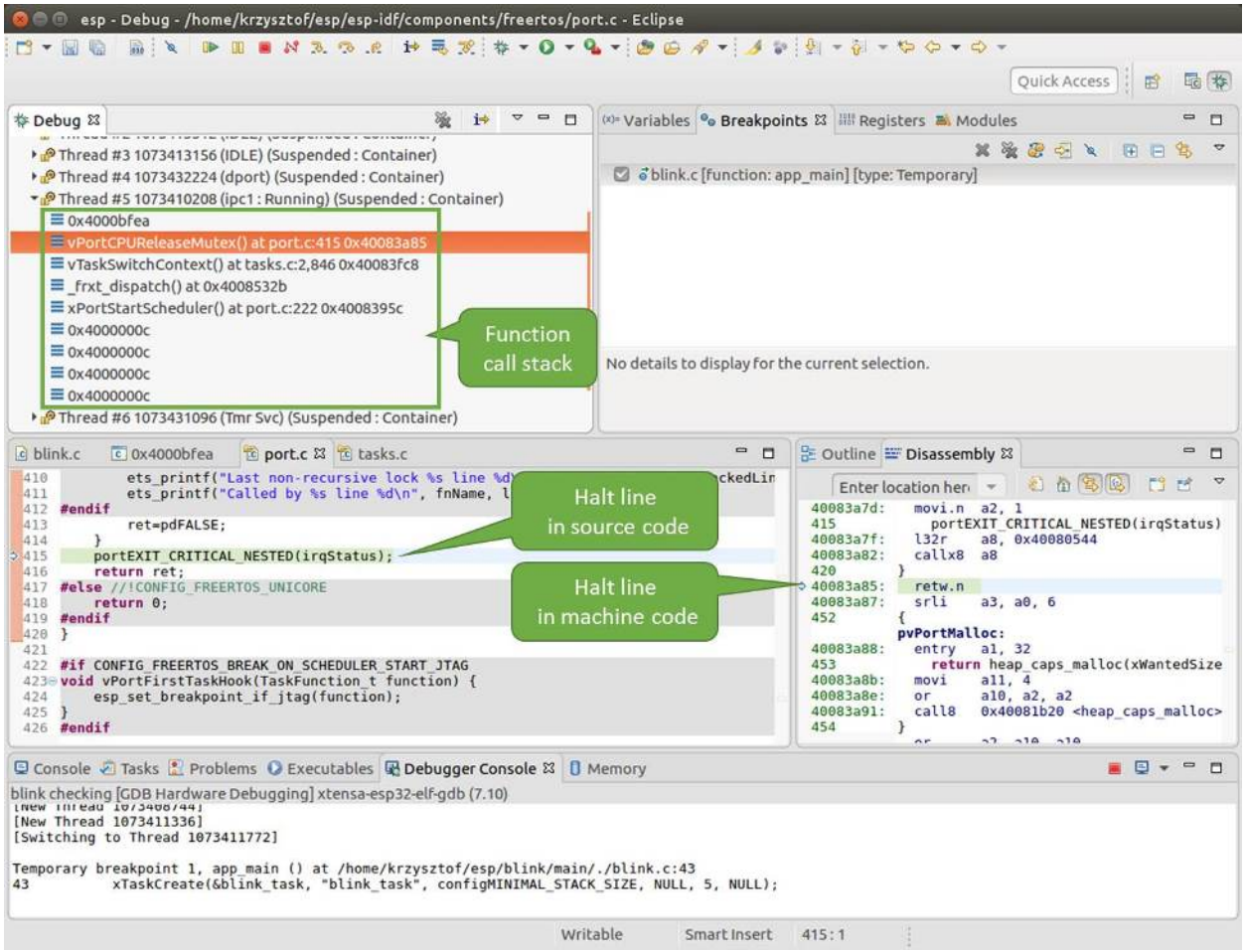


Fig. 13: Navigate through the call stack

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 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. To do so, hold the "Control" on the keyboard and double clink 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.

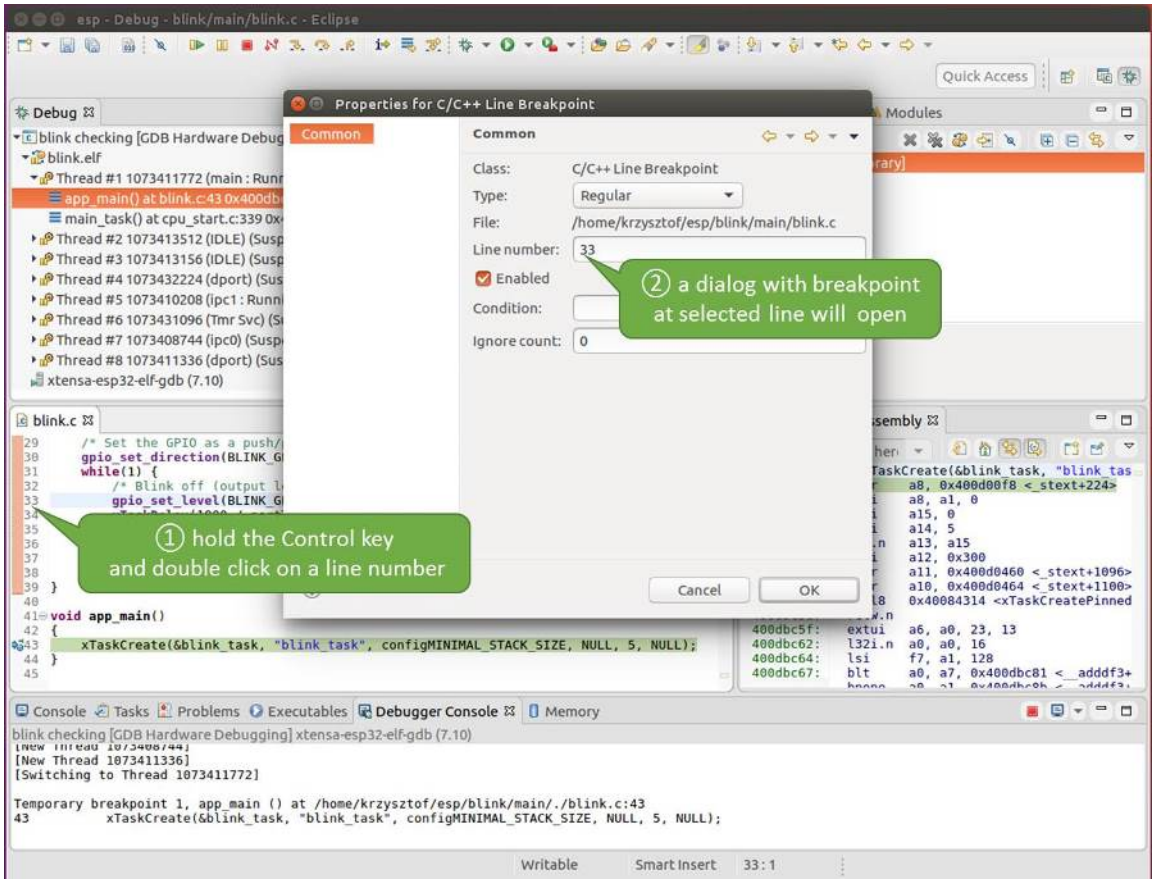


Fig. 14: Setting a breakpoint

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

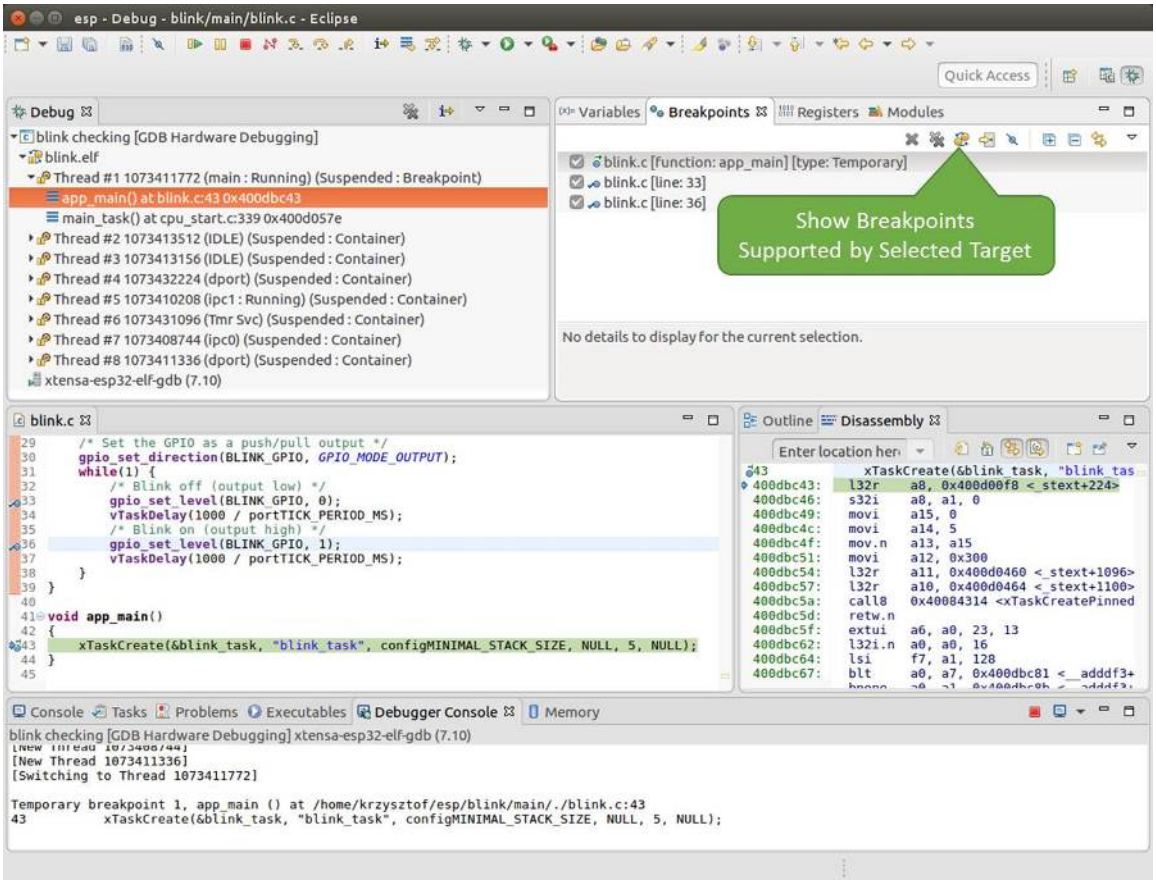


Fig. 15: Three breakpoints are set / maximum two are allowed

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.

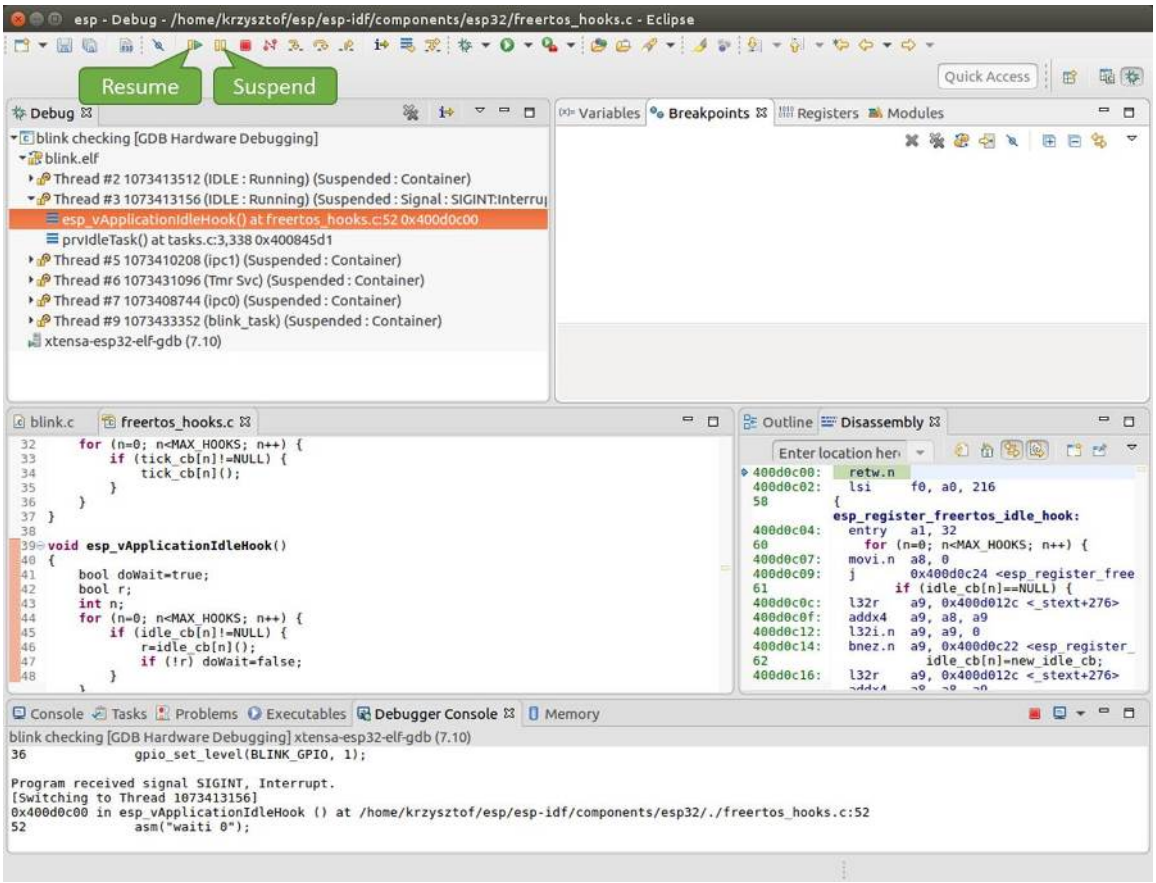


Fig. 16: Target halted manually

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 subroutines 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.

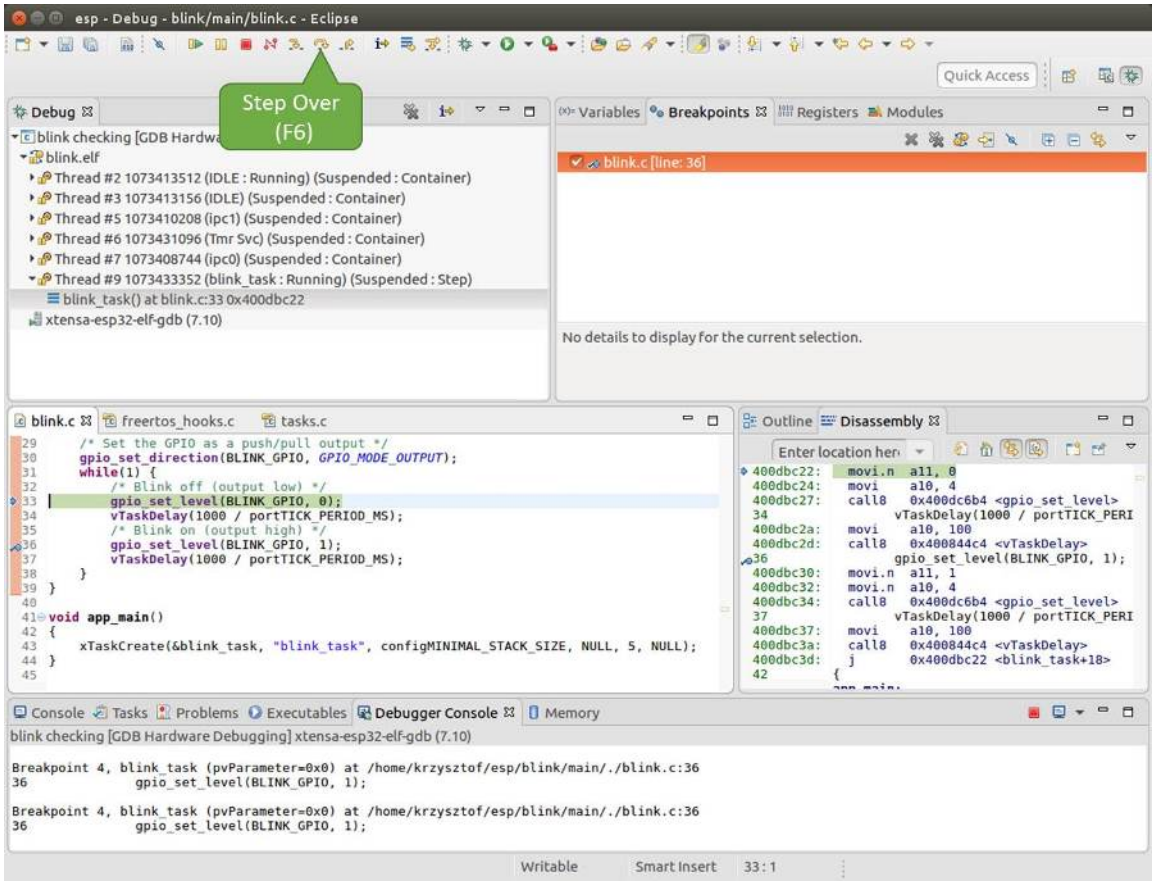


Fig. 17: Stepping through the code with "Step Over (F6)"

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

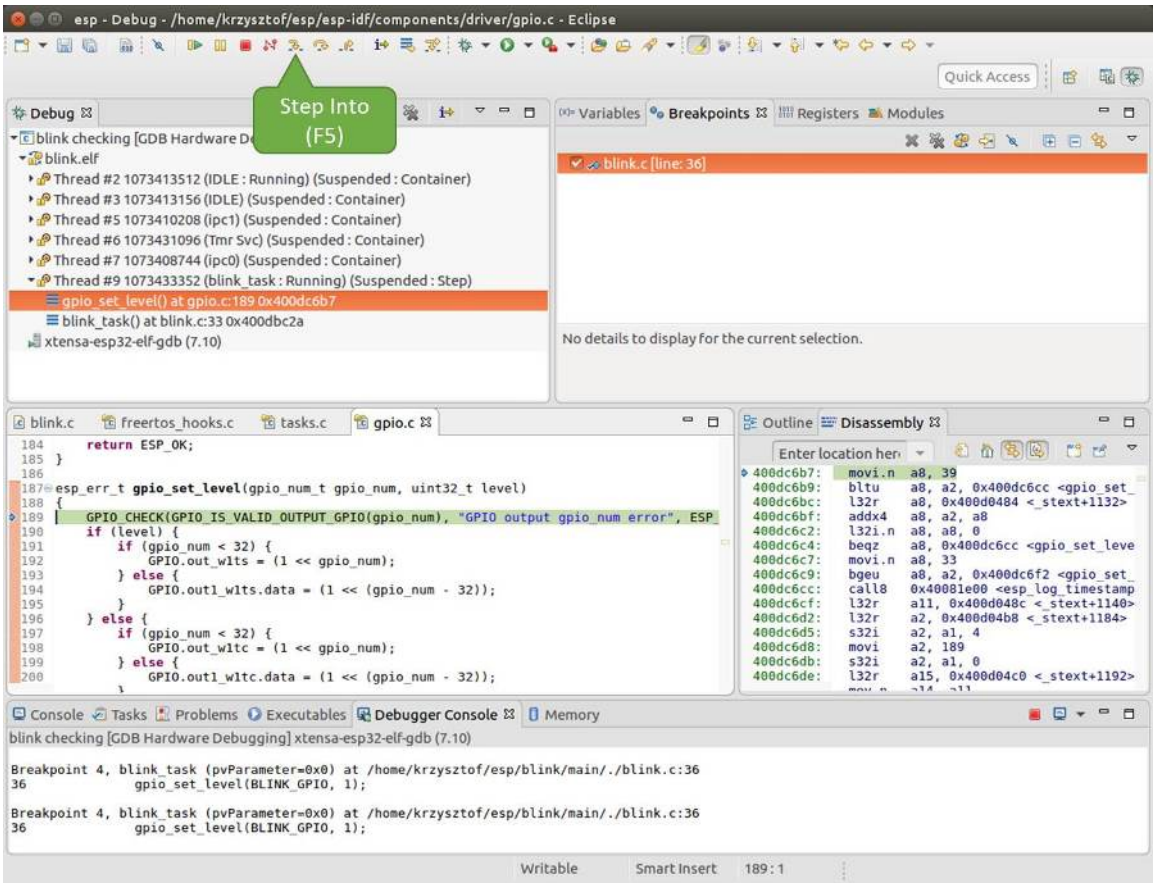


Fig. 18: Stepping through the code with "Step Into (F5)"

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.

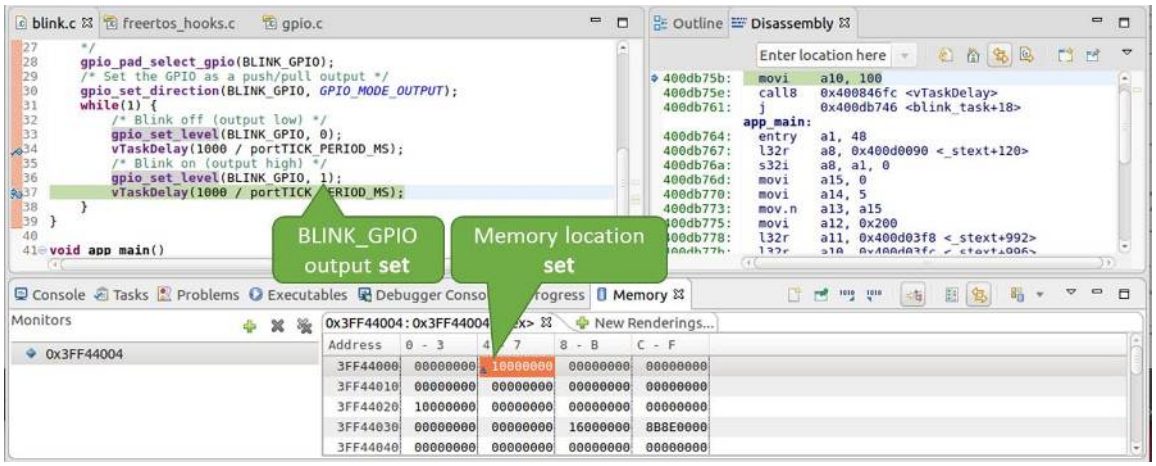


Fig. 19: Observing memory location 0x3FF44004 changing one bit to ON"

You should see one bit being flipped over at memory location `0x3FF44004` (and LED changing the state) each time F8 is pressed.

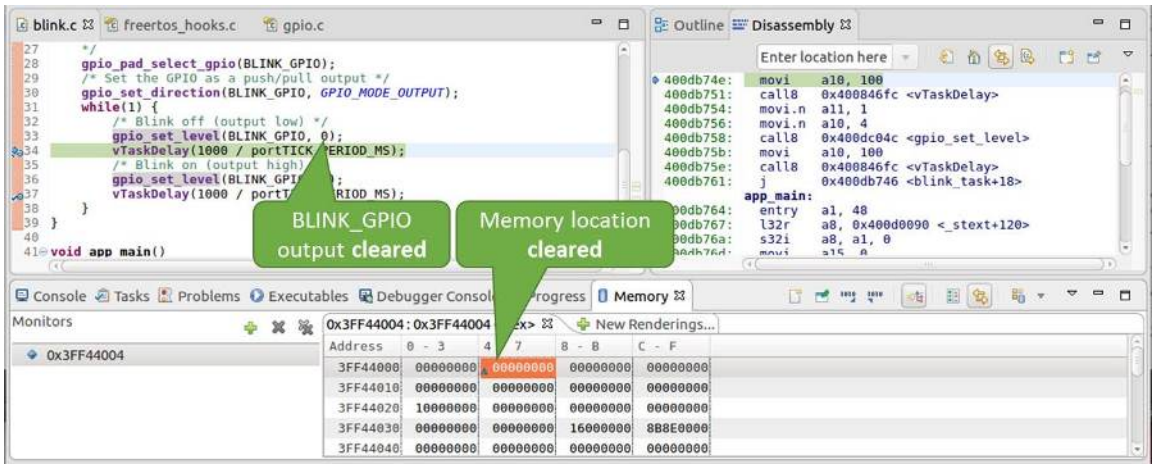


Fig. 20: Observing memory location 0x3FF44004 changing one bit to ON"

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.

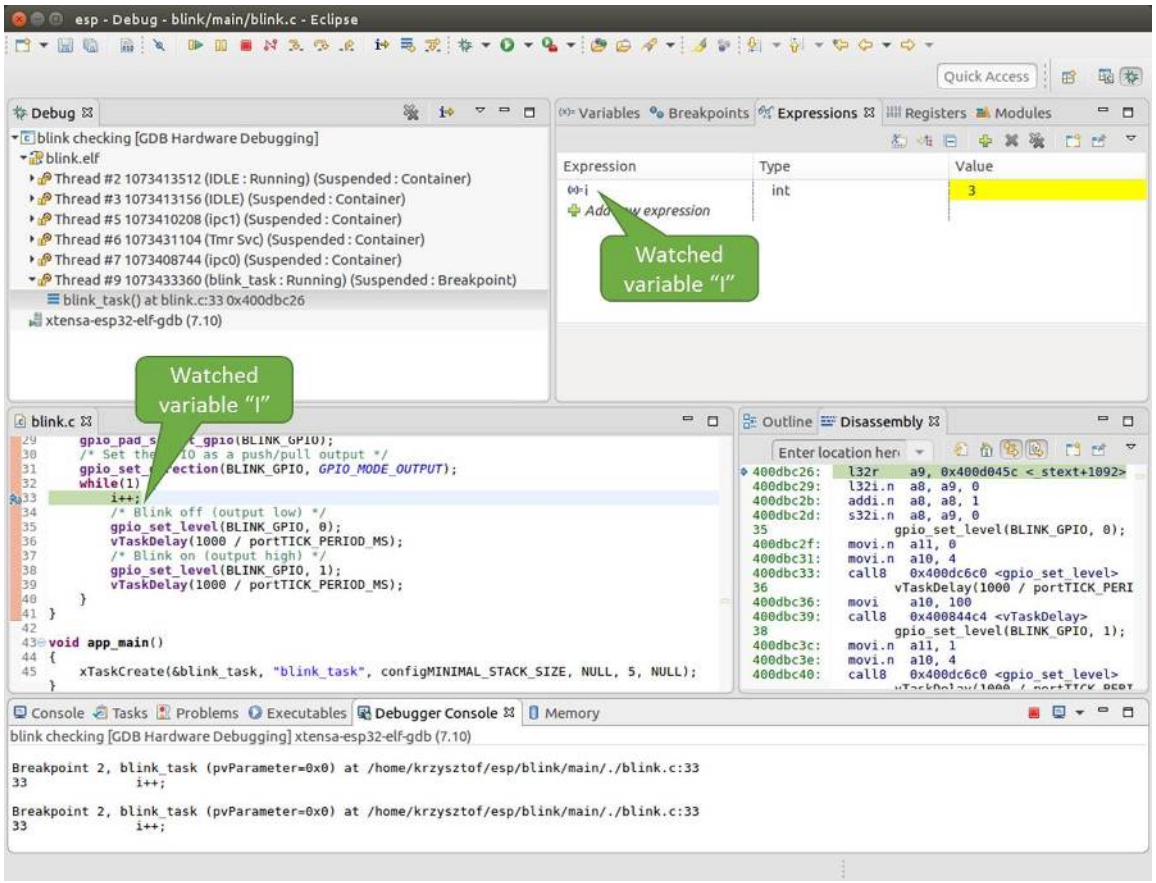


Fig. 21: Watching program variable "i"

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.

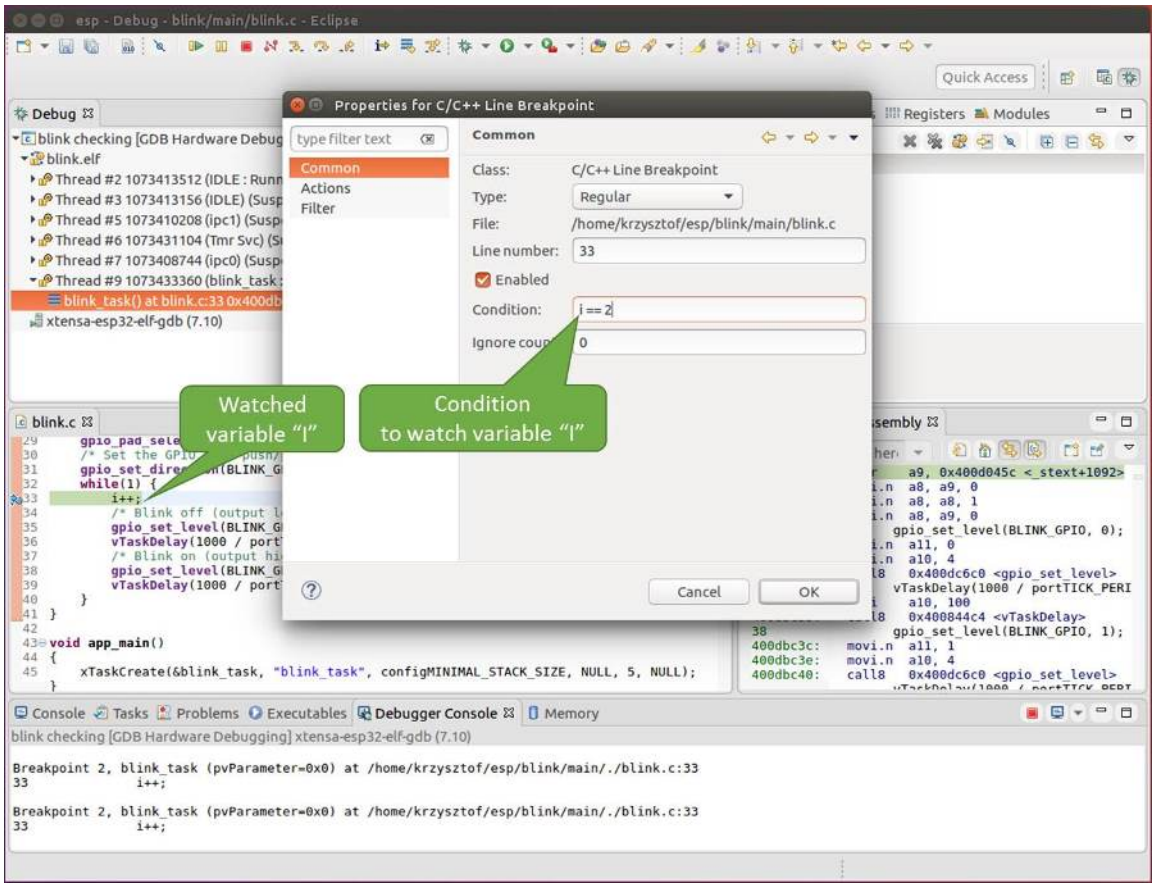


Fig. 22: Setting a conditional breakpoint

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 `l` and this will reveal the piece of code that called `app_main()` (in line 339):

```
(gdb) l
334             ;
335         }
336     #endif
337         //Enable allocation in region where the startup stacks were located.
338         heap_caps_enable_nonos_stack_heaps();
339         app_main();
340         vTaskDelete(NULL);
341     }
342
(gdb)
```

By listing some lines before, you will see the function name `main_task` we have been looking for:

```
(gdb) l 326, 341
326     static void main_task(void* args)
327     {
328         // Now that the application is about to start, disable boot watchdogs
329         REG_CLR_BIT(TIMG_WDTCONFIG0_REG(0), TIMG_WDT_FLASHBOOT_MOD_EN_S);
330         REG_CLR_BIT(RTC_CNTL_WDTCONFIG0_REG, RTC_CNTL_WDT_FLASHBOOT_MOD_EN);
331     #if !CONFIG_FREERTOS_UNICORE
332         // Wait for FreeRTOS initialization to finish on APP CPU, before␣
↪replacing its startup stack
333         while (port_xSchedulerRunning[1] == 0) {
334             ;
335         }
336     #endif
337         //Enable allocation in region where the startup stacks were located.
338         heap_caps_enable_nonos_stack_heaps();
339         app_main();
340         vTaskDelete(NULL);
341     }
(gdb)
```

To see the other code, enter `i threads`. This will show the list of threads running on target:

```
(gdb) i threads
  Id   Target Id         Frame
  8    Thread 1073411336 (dport) 0x400d0848 in dport_access_init_core (arg=<optimized␣
↪out>)
    at /home/user-name/esp/esp-idf/components/esp32/./dport_access.c:170
  7    Thread 1073408744 (ipc0) xQueueGenericReceive (xQueue=0x3ffae694, pvBuffer=0x0,
↪ xTicksToWait=1644638200,
```

(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 (ipc1 : Running) 0x4000bfea in ?? ()
  4    Thread 1073432224 (dport) dport_access_init_core (arg=0x0)
    at /home/user-name/esp/esp-idf/components/esp32/./dport_access.c:150
  3    Thread 1073413156 (IDLE) prvIdleTask (pvParameters=0x0)
    at /home/user-name/esp/esp-idf/components/freertos/./tasks.c:3282
  2    Thread 1073413512 (IDLE) prvIdleTask (pvParameters=0x0)
    at /home/user-name/esp/esp-idf/components/freertos/./tasks.c:3282
* 1    Thread 1073411772 (main : Running) app_main () at /home/user-name/esp/blink/
↪main/./blink.c:43
(gdb)
```

The thread list shows the last function calls per each thread together with the name of C source file if available.

You can navigate to specific thread by entering `thread N`, where `N` is the thread Id. To see how it works go to thread thread 5:

```
(gdb) thread 5
[Switching to thread 5 (Thread 1073410208)]
#0  0x4000bfea in ?? ()
(gdb)
```

Then check the backtrace:

```
(gdb) bt
#0  0x4000bfea in ?? ()
#1  0x40083a85 in vPortCPUReleaseMutex (mux=<optimized out>) at /home/user-name/esp/
↪esp-idf/components/freertos/./port.c:415
#2  0x40083fc8 in vTaskSwitchContext () at /home/user-name/esp/esp-idf/components/
↪freertos/./tasks.c:2846
#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.

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 your are blinking LED connected to GPIO4, then you should see fourth bit being flipped each time the LED changes the state:

```
0x3ff44004: 0x00000000
...
0x3ff44004: 0x00000010
```

Now, when the LED is off, that corresponds to `0x3ff44004: 0x00000000` being displayed, try using `set` command to set this bit by writting `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 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 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 for should provide are very basis and intended to let you quickly get started with JTAG debugging. Check help what are the other commands at you 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 *CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK* uses the 2nd watchpoint and will not provide expected results, if you also try to use it within OpenOCD / GDB. See menuconfig's help for detailed description.

What else should I know about breakpoints?

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:

- *CONFIG_ESP32_DEBUG_OCDAWARE* is enabled by default. If a panic or unhandled exception is thrown and a JTAG debugger is connected (ie openocd is running), ESP-IDF will break into the debugger.
- *CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK* (disabled by default) sets watchpoint index 1 (the second of two) at the end of any task stack. This is the most accurate way to debug task stack overflows. Click the link for more details.

Please see the *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 MDTI 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
- `flushregs` — monitor (`mon`) command can not inform GDB that the target state has changed. GDB will assume that whatever stack the target had before `mon reset halt` will still be valid. In fact, after reset the target state will change, and executing `flushregs` is a way to force GDB to get new state from the target.
- `thb app_main` — insert a temporary hardware breakpoint at `app_main`, put here another function name if required
- `c` — resume the program. It will then stop at breakpoint inserted at `app_main`.

Configuration of OpenOCD for specific target

OpenOCD needs to be told what JTAG adapter **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.

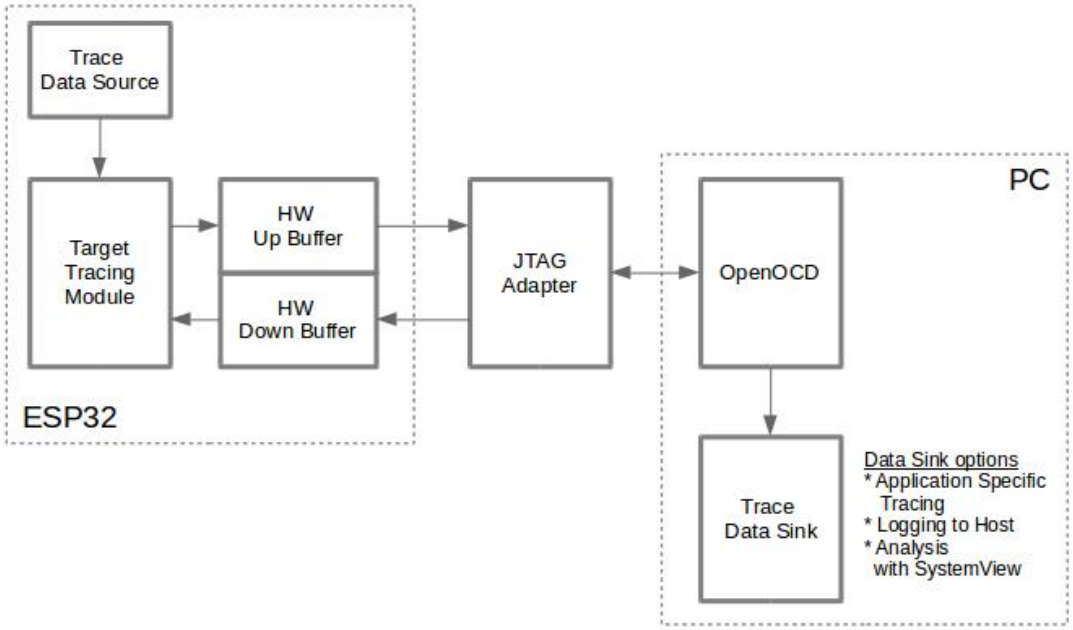


Fig. 23: Tracing Components when Working Over JTAG

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 controled via timeout values passed by users to corresponding API routines. So when application tries to write data to trace buffer using finite value of the maximum waiting time it is possible situation that this data will be dropped. Especially this is true for tracing from time critical code (ISRs, OS scheduler code etc.) when infinite timeouts can lead to system malfunction. In order to avoid loss of such critical data developers can enable additional data buffering via menuconfig option `CONFIG_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* (`CONFIG_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* (`CONFIG_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_apptrace_read(ESP_APPTRACE_DEST_TRAX, buf, &sz, 0/*do not␣
↪wait*/);
if (res != ESP_OK) {
    ESP_LOGE(TAG, "Failed to read data from host!");
    return res;
}
if (sz > 0) {
    /* we have data, process them */
    ...
}
```

`esp_apptrace_read()` function uses memcpy to copy host data to user buffer. In some cases it can be more optimal to use `esp_apptrace_down_buffer_get()` and `esp_apptrace_down_buffer_put()` functions. They allow developers to occupy chunk of read buffer and process it in-place. The following piece of code shows how to do this.

```
#include "esp_app_trace.h"
...
char down_buf[32];
uint32_t *number;
size_t sz = 32;

/* config down buffer */
esp_apptrace_down_buffer_config(down_buf, sizeof(down_buf));
char *ptr = (char *)esp_apptrace_down_buffer_get(ESP_APPTRACE_DEST_TRAX, &sz, 100/
↪*tmo in us*/);
if (ptr == NULL) {
    ESP_LOGE("Failed to get buffer!");
    return ESP_FAIL;
}
if (sz > 4) {
    number = (uint32_t *)ptr;
    printf("Here is the number %d", *number);
} else {
    printf("No data");
}
esp_err_t res = esp_apptrace_down_buffer_put(ESP_APPTRACE_DEST_TRAX, ptr, 100/
↪*tmo in us*/);
if (res != ESP_OK) {
    /* in case of error host tracing tool (e.g. OpenOCD) will report incomplete␣
↪user buffer */
    ESP_LOGE("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 then 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 apptrace start file://trace.log 1 2048 5 0 0
   ```

   The tracing data will be retrieved and saved in non-blocking mode. This process will stop automatically after 2048 bytes are collected, or if no data are available for more than 5 seconds.

   Note: Tracing data is buffered before it is made available to OpenOCD. If you see "Data timeout!" message, then 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_apptrace_flush()` to flush the data on specific intervals.

2. Retrieve tracing data indefinitely in non-blocking mode.

   ```
   esp32 apptrace 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 apptrace stop` command on OpenOCD telnet prompt, or by pressing Ctrl+C in OpenOCD window.

3. Retrieve tracing data and save them indefinitely.

   ```
   esp32 apptrace start file://trace.log 0 -1 -1 0 0
   ```

   OpenOCD telnet command line prompt will not be available until tracing is stopped. To stop tracing press Ctrl+C in 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 apptrace start file://trace.log 0 2048 -1 1 0
   ```

   To configure tracing immediately after reset use the openocd `reset halt` command.

Logging to Host

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* (*CONFIG_SYSVIEW_ENABLE*) menuconfig option. There are several other options enabled under the same menu:

1. *ESP32 timer to use as SystemView timestamp source* (*CONFIG_SYSVIEW_TS_SOURCE*) selects the source of timestamps for SystemView events. In single core mode timestamps are generated using ESP32 internal cycle counter running at maximum 240 Mhz (~4 ns granularity). In dual-core mode external timer working at 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 then 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 "pro-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 its 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.

5.13 Bootloader

Bootloader performs the following functions:

1. Minimal initial configuration of internal modules;
2. Select the application partition to boot, based on the partition table and ota_data (if any);
3. Load this image to RAM (IRAM & DRAM) and transfer management to it.

Bootloader is located at the address *0x1000* in the flash.

5.13.1 FACTORY reset

The user can write a basic working firmware and load it into the factory partition. Next, update the firmware via OTA (over the air). The updated firmware will be loaded into an OTA app partition slot and the OTA data partition is updated to boot from this partition. If you want to be able to roll back to the factory firmware and clear the settings, then you need to set *`CONFIG_BOOTLOADER_FACTORY_RESET`*. The factory reset mechanism allows to reset the device to factory settings:

- Clear one or more data partitions.

- Boot from "factory" partition.

CONFIG_BOOTLOADER_DATA_FACTORY_RESET allows customers to select which data partitions will be erased when the factory reset is executed. Can specify the names of partitions through comma-delimited with optional spaces for readability. (Like this: "nvs, phy_init, nvs_custom, ...”). Make sure that the name specified in the partition table and here are the same. Partitions of type "app" cannot be specified here.

CONFIG_BOOTLOADER_OTA_DATA_ERASE - the device will boot from "factory" partition after a factory reset. The OTA data partition will be cleared.

CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET- number of the GPIO input for factory reset uses to trigger a factory reset, this GPIO must be pulled low on reset to trigger this.

CONFIG_BOOTLOADER_HOLD_TIME_GPIO- this is hold time of GPIO for reset/test mode (by default 5 seconds). The GPIO must be held low continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.

Partition table.:

```
# Name,   Type, SubType, Offset,   Size, Flags
# Note: if you change the phy_init or app partition offset, make sure to change the␣
↪offset in Kconfig.projbuild
nvs,      data, nvs,     0x9000,   0x4000
otadata,  data, ota,     0xd000,   0x2000
phy_init, data, phy,     0xf000,   0x1000
factory,  0,    0,       0x10000,  1M
test,     0,    test,    ,         512K
ota_0,    0,    ota_0,   ,         512K
ota_1,    0,    ota_1,   ,         512K
```

5.13.2 Boot from TEST firmware

The user can write a special firmware for testing in production, and run it as needed. The partition table also needs a dedicated partition for this testing firmware (See *partition table*). To trigger a test app you need to set *CONFIG_BOOTLOADER_APP_TEST*.

CONFIG_BOOTLOADER_NUM_PIN_APP_TEST - GPIO number to boot TEST partition. The selected GPIO will be configured as an input with internal pull-up enabled. To trigger a test app, this GPIO must be pulled low on reset. After the GPIO input is deactivated and the device reboots, the normally configured application will boot (factory or any OTA slot).

CONFIG_BOOTLOADER_HOLD_TIME_GPIO - this is hold time of GPIO for reset/test mode (by default 5 seconds). The GPIO must be held low continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.

5.13.3 Customer bootloader

The current bootloader implementation allows the customer to override it. To do this, you must copy the folder */esp-idf/components/bootloader* and then edit */your_project/components/bootloader/subproject/main/bootloader_main.c*. In the bootloader space, you can not use the drivers and functions from other components. If necessary, then the required functionality should be placed in the folder bootloader (note that this will increase its size). It is necessary to monitor its size because there can be overlays in memory with a partition table leading to damage. At the moment the bootloader is limited to the partition table from the address *0x8000*.

5.14 Partition Tables

5.14.1 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). An MD5 checksum is appended after the table data. If the partition table is signed due to *secure boot*, the signature is appended after the partition table.

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

The simplest way to use the partition table is to *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.

5.14.2 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.

5.14.3 Creating Custom Tables

If you choose "Custom partition table CSV" in menuconfig then you can also enter the name of a CSV file (in the project directory) to use for your partition table. The CSV file can describe any number of definitions for the table you need.

The CSV format is the same format as printed in the summaries shown above. However, not all fields are required in the CSV. For example, here is the "input" CSV for the OTA partition table:

```
# Name,   Type, SubType,  Offset,   Size,  Flags
nvs,      data, nvs,      0x9000,  0x4000
otadata,  data, ota,      0xd000,  0x2000
phy_init, data, phy,      0xf000,  0x1000
factory,  app,  factory,  0x10000,  1M
ota_0,    app,  ota_0,    ,         1M
ota_1,    app,  ota_1,    ,         1M
nvs_key,  data, nvs_keys, ,        0x1000
```

- Whitespace between fields is ignored, and so is any line starting with # (comments).
- Each non-comment line in the CSV file is a partition definition.
- 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 will be used as the fallback boot partition if no other valid app partition is found. It is also possible to configure the bootloader to read a GPIO input during each boot, and boot this partition if the GPIO is held low, see *Boot from TEST firmware*.

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 `CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION` option. You will also need to flash your devices with phy init data as the esp-idf build system does not do this automatically.
- nvs (2) is for the *Non-Volatile Storage (NVS) API*.
  - NVS is used to store per-device PHY calibration data (different to initialisation data).
  - NVS is used to store WiFi data if the *esp_wifi_set_storage(WIFI_STORAGE_FLASH)* initialisation function is used.
  - The NVS API can also be used for other application data.
  - It is strongly recommended that you include an NVS partition of at least 0x3000 bytes in your project.
  - If using NVS API to store a lot of data, increase the NVS partition size from the default 0x6000 bytes.
- keys (4) is for the NVS key partition. See *Non-Volatile Storage (NVS) API* for more details. - It is used to store NVS encryption keys when *NVS Encryption* feature is enabled. - The size of this partition should be 4096 bytes (minimum partition size).

Other data subtypes are reserved for future esp-idf uses.

Offset & Size

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).

5.14.4 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.)

MD5 checksum

The binary format of the partition table contains an MD5 checksum computed based on the partition table. This checksum is used for checking the integrity of the partition table during the boot.

The MD5 checksum generation can be disabled by the `--disable-md5sum` option of `gen_esp32part.py` or by the *`CONFIG_PARTITION_TABLE_MD5`* option. This is useful for example when one uses a legacy bootloader which cannot process MD5 checksums and the boot fails with the error message `invalid magic number 0xebeb`.

5.14.5 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.

5.15 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 throughly and understand the implications of enabling secure boot.

Note: Secure boot is only supported when using the default GNU Make build system. The CMake build system preview in ESP-IDF v3.x does not support secure boot.

5.15.1 Background

- Most data is stored in flash. Flash access does not need to be protected from physical access in order for secure boot to function, because critical data is stored (non-software-accessible) in Efuses internal to the chip.
- Efuses are used to store the secure bootloader key (in efuse BLOCK2), and also a single Efuse bit (ABS_DONE_0) is burned (written to 1) to permanently enable secure boot on the chip. For more details about efuse, see Chapter 11 "eFuse Controller" in the Technical Reference Manual.
- To understand the secure boot process, first familiarise yourself with the standard *ESP-IDF boot process*.
- Both stages of the boot process (initial software bootloader load, and subsequent partition & app loading) are verified by the secure boot process, in a "chain of trust" relationship.

5.15.2 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.

5.15.3 Keys

The following keys are used by the secure boot process:

- "secure bootloader key" is a 256-bit AES key that is stored in Efuse block 2. The bootloader can generate this key itself from the internal hardware random number generator, the user does not need to supply it (it is optionally possible to supply this key, see *Re-Flashable Software Bootloader*). The Efuse holding this key is read & write protected (preventing software access) before secure boot is enabled.
  - By default, the Efuse Block 2 Coding Scheme is "None" and a 256 bit key is stored in this block. On some ESP32s, the Coding Scheme is set to 3/4 Encoding (CODING_SCHEME efuse has value 1) and a 192 bit key must be stored in this block. See ESP32 Technical Reference Manual section 20.3.1.3 *System Parameter coding_scheme* for more details. The algorithm operates on a 256 bit key in all cases, 192 bit keys are extended by repeating some bits (*details*).
- "secure boot signing key" is a standard ECDSA public/private key pair (see *Image Signing Algorithm*) in PEM format.
  - The public key from this key pair (for signature verification but not signature creation) is compiled into the software bootloader and used to verify the second stage of booting (partition table, app image) before booting continues. The public key can be freely distributed, it does not need to be kept secret.
  - The private key from this key pair *must be securely kept private*, as anyone who has this key can authenticate to any bootloader that is configured with secure boot and the matching public key.

5.15.4 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 `espsecure.py generate_signing_key`.

   IMPORTANT A signing key generated this way will use the best random number source available to the OS and its Python installation (*/dev/urandom* on OSX/Linux and *CryptGenRandom()* on Windows). If this random number source is weak, then the private key will be weak.

   IMPORTANT For production environments, we recommend generating the keypair using openssl or another industry standard encryption program. See *Generating Secure Boot Signing Key* for more details.
5. Run `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).

5.15.5 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 binary key file that is used for the secure bootloader key. As you have the key file, you can generate new bootloader images and secure boot digests for them.

In the esp-idf build process, this 256-bit key file is derived from the app signing key generated during the generate_signing_key step above. The private key's SHA-256 digest is used as the secure bootloader key (as-is for Coding Scheme None, or truncate to 192 bytes for 3/4 Encoding). This is a convenience so you only need to generate/protect a single private key.

NOTE: Although it's possible, we strongly recommend not generating one secure boot key and flashing it to every device in a production environment. The "One-Time Flash" option is recommended for production environments.

To enable a reflashable bootloader:

1. In the `make menuconfig` step, select "Bootloader Config" -> *`CONFIG_SECURE_BOOT_ENABLED`* -> *`CONFIG_SECURE_BOOTLOADER_MODE`* -> Reflashable.

2. If necessary, set the *`CONFIG_SECURE_BOOTLOADER_KEY_ENCODING`* based on the coding scheme used by the device. The coding scheme is shown in the `Features` line when `esptool.py` connects to the chip, or in the `espefuse.py summary` output.

2. Follow the steps shown above to choose a signing key file, and generate the key file.

3. Run `make bootloader`. A binary key file will be created, derived from the private key that is used for signing. Two sets of flashing steps will be printed - the first set of steps includes an `espefuse.py burn_key` command which is used to write the bootloader key to efuse. (Flashing this key is a one-time-only process.) The second set of steps can be used to reflash the bootloader with a pre-calculated digest (generated during the build process).

4. Resume from *Step 6 of the one-time flashing process*, to flash the bootloader and enable secure boot. Watch the console log output closely to ensure there were no errors in the secure boot configuration.

5.15.6 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.

5.15.7 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
```

5.15.8 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 interpeter, and disabling the UART bootloader encrypted flash access.
- Use secure boot in combination with *flash encryption* to prevent local readout of the flash contents.

5.15.9 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. Read the AES key from efuse block 2, in reversed byte order. If Coding Scheme is set to 3/4 Encoding, extend the 192 bit key to 256 bits using the same algorithm described in *Flash Encryption Algorithm*.
2. Prefix the image with a 128 byte randomly generated IV.
3. If the image length is not modulo 128, pad the image to a 128 byte boundary with 0xFF. (^)
4. For each 16 byte plaintext block of the input image: - Reverse the byte order of the plaintext input block (^) - Apply AES256 in ECB mode to the plaintext block. - Reverse the byte order of the ciphertext output block. (^) - Append to the overall ciphertext output.
5. Byte-swap each 4 byte word of the ciphertext (^)
6. Calculate SHA-512 of the ciphertext.

Output digest is 192 bytes of data: The 128 byte IV, followed by the 64 byte SHA-512 digest.

Image Signing Algorithm

Deterministic ECDSA as specified by RFC 6979.

- Curve is NIST256p (openssl calls this curve "prime256v1", it is also sometimes called secp256r1).
- Hash function is SHA256.
- Key format used for storage is PEM.
  - In the bootloader, the public key (for signature verification) is flashed as 64 raw bytes.
- Image signature is 68 bytes - a 4 byte version word (currently zero), followed by a 64 bytes of signature data. These 68 bytes are appended to an app image or partition table data.

Manual Commands

Secure boot is integrated into the esp-idf build system, so `make` will automatically sign an app image if secure boot is enabled. `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
```

5.15.10 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.

5.15.11 Signed App Verification Without Hardware Secure Boot

The integrity of apps can be checked even without enabling the hardware secure boot option. This option uses the same app signature scheme as hardware secure boot, but unlike hardware secure boot it does not prevent the bootloader from being physically updated. This means that the device can be secured against remote network access, but not physical access. Compared to using hardware Secure Boot this option is much simpler to implement. See *How To Enable Signed App Verification* for step by step instructions.

An app can be verified on update and, optionally, be verified on boot.

- Verification on update: When enabled, the signature is automatically checked whenever the esp_ota_ops.h APIs are used for OTA updates. If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option still adds significant security against network-based attackers by preventing spoofing of OTA updates.
- Verification on boot: When enabled, the bootloader will be compiled with code to verify that an app is signed before booting it. If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option doesn't add significant security by itself so most users will want to leave it disabled.

How To Enable Signed App Verification

1. Run `make menuconfig` -> Security features -> Enable "Require signed app images"
2. "Bootloader verifies app signatures" can be enabled, which verifies app on boot.

3. By default, "Sign binaries during build" will be enabled on selecting "Require signed app images" option, which will sign binary files as a part of build process. The file named in "Secure boot private signing key" will be used to sign the image.
4. If you disable "Sign binaries during build" option then you'll have to enter path of a public key file used to verify signed images in "Secure boot public signature verification key". In this case, private signing key should be generated by following instructions in *Generating Secure Boot Signing Key*; public verification key and signed image should be generated by following instructions in *Remote Signing of Images*.

5.16 ULP coprocessor programming

5.16.1 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 regiter 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 instuction, 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
         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
```

(continues on next page)

(continued from previous page)

```
        .text                       //Text section definition
        MOVE      R1, 1             // R1 = 1
        MOVE      R2, Addr1         // R2 = Addr1 / 4 (address of label is␣
↪converted into words)
        LD        R1, R2, offs      // R1 = MEM[R2 +  0]
                                    // R1 will be 123
```

JUMP – Jump to an absolute address

Syntax **JUMP** *Rdst*

JUMP *ImmAddr*

JUMP *Rdst, Condition*

JUMP *ImmAddr, Condition*

Operands

- *Rdst* – Register R[0..3] containing address to jump to (expressed in 32-bit words)
- *ImmAddr* – 13 bits address (expressed in bytes), aligned to 4 bytes
- ***Condition*:**
  - EQ – jump if last ALU operation result was zero
  - OV – jump if last ALU has set overflow flag

Cycles 2 cycles to execute, 2 cycles to fetch next instruction

Description The instruction makes jump to the specified address. Jump can be either unconditional or based on an ALU flag.

Examples:

```
1:        JUMP       R1            // Jump to address in R1 (address in R1 is in 32-
↪bit words)

2:        JUMP       0x120, EQ     // Jump to address 0x120 (in bytes) if ALU result␣
↪is zero

3:        JUMP       label         // Jump to label
          ...
  label:  nop                      // Definition of label

4:        .global    label         // Declaration of global label

          MOVE       R1, label     // R1 = label (value loaded into R1 is in words)
          JUMP       R1            // Jump to label
          ...
  label:  nop                      // Definition of label
```

JUMPR – Jump to a relative offset (condition based on R0)

Syntax **JUMPR** *Step, Threshold, Condition*

Operands

- *Step* – relative shift from current position, in bytes
- *Threshold* – threshold value for branch condition
- ***Condition***:
  - *GE* (greater or equal) – jump if value in R0 >= 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 <= threshold
  - *GT* (greater than) – jump if value in stage_cnt > threshold
  - *GE* (greater or equal) — jump if value in stage_cnt >= threshold

Cycles Conditions *LE*, *LT*, *GE*: 2 cycles to execute, 2 cycles to fetch next instruction

Conditions *EQ*, *GT* are implemented in the assembler using two **JUMPS** instructions:

```
// JUMPS target, threshold, EQ is implemented as:

         JUMPS next, threshold, LT
         JUMPS target, threshold, LE
next:

// JUMPS target, threshold, GT is implemented as:

         JUMPS next, threshold, LE
         JUMPS target, threshold, GE
next:
```

Therefore the execution time will depend on the branches taken: either 2 cycles to execute + 2 cycles to fetch, or 4 cycles to execute + 4 cycles to fetch.

Description The instruction makes a jump to a relative address if condition is true. Condition is the result of comparison of count register value and threshold value.

Examples:

```
1:pos:    JUMPS     16, 20, EQ     // Jump to (position + 16 bytes) if stage_cnt == 20

2:        // Up counting loop using stage count register
          STAGE_RST                  // set stage_cnt to 0
  label:  STAGE_INC  1               // stage_cnt++
          NOP                        // do something
          JUMPS       label, 16, LT  // jump to label if stage_cnt < 16
```

STAGE_RST – Reset stage count register

Syntax **STAGE_RST**

Operands No operands

Description The instruction sets the stage count register to 0

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Examples:

```
1:       STAGE_RST      // Reset stage count register
```

STAGE_INC – Increment stage count register

Syntax **STAGE_INC** *Value*

Operands

- *Value* – 8 bits value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction increments stage count register by given value.

Examples:

```
1:        STAGE_INC      10          // stage_cnt += 10

2:        // Up counting loop example:
          STAGE_RST                  // set stage_cnt to 0
  label:  STAGE_INC  1               // stage_cnt++
          NOP                        // do something
          JUMPS      label, 16, LT   // jump to label if stage_cnt < 16
```

STAGE_DEC – Decrement stage count register

Syntax **STAGE_DEC** *Value*

Operands

- *Value* – 8 bits value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction decrements stage count register by given value.

Examples:

```
1:        STAGE_DEC      10        // stage_cnt -= 10;

2:        // Down counting loop 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
```

(continues on next page)

(continued from previous page)

```
        // 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 + *Wait_Delay* + 3 * TSENS_CLK to execute, 4 cycles to fetch next instruction

Description The instruction performs measurement using TSENS and stores the result into a general purpose register.

Examples:

```
1:        TSENS     R1, 1000     // Measure temperature sensor for 1000 cycles,
                                 // and store result to R1
```

ADC – do measurement with ADC

Syntax

- **ADC** *Rdst, Sar_sel, Mux*
- **ADC** *Rdst, Sar_sel, Mux, 0* — deprecated form

Operands

- *Rdst* – Destination Register R[0..3], result will be stored to this register
- *Sar_sel* – Select ADC: 0 = SARADC1, 1 = SARADC2
- *Mux* - selected PAD, SARADC Pad[Mux+1] is enabled

Cycles `23 + max(1, SAR_AMP_WAIT1) + max(1, SAR_AMP_WAIT2) + max(1, SAR_AMP_WAIT3) + SARx_SAMPLE_CYCLE + 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 = REG[Addr][High:Low]`.

This instruction can access registers in RTC_CNTL, RTC_IO, SENS, and RTC_I2C peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

```
addr_ulp = (addr_dport - 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, SENS, and RTC_I2C peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

```
addr_ulp = (addr_dport - DR_REG_RTCCNTL_BASE) / 4
```

Examples:

```
1:        REG_WR      0x120, 7, 0, 0x10   // set 8 bits: REG[0x120][7:0] = 0x10
```

Convenience macros for peripheral registers access

ULP source files are passed through C preprocessor before the assembler. This allows certain macros to be used to facilitate access to peripheral registers.

Some existing macros are defined in `soc/soc_ulp.h` header file. These macros allow access to the fields of peripheral registers by their names. Peripheral registers names which can be used with these macros are the ones defined in `soc/rtc_cntl_reg.h`, `soc/rtc_io_reg.h`, `soc/sens_reg.h`, and `soc/rtc_i2c_reg.h`.

READ_RTC_REG(rtc_reg, low_bit, bit_width) Read up to 16 bits from rtc_reg[low_bit + bit_width - 1 : low_bit] into R0. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Read 16 lower bits of RTC_CNTL_TIME0_REG into R0 */
READ_RTC_REG(RTC_CNTL_TIME0_REG, 0, 16)
```

READ_RTC_FIELD(rtc_reg, field) Read from a field in rtc_reg into R0, up to 16 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/sens_reg.h"

/* Read 8-bit SENS_TSENS_OUT field of SENS_SAR_SLAVE_ADDR3_REG into R0 */
READ_RTC_FIELD(SENS_SAR_SLAVE_ADDR3_REG, SENS_TSENS_OUT)
```

WRITE_RTC_REG(rtc_reg, low_bit, bit_width, value) Write immediate value into rtc_reg[low_bit + bit_width - 1 : low_bit], bit_width <= 8. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_io_reg.h"

/* Set BIT(2) of RTC_GPIO_OUT_DATA_W1TS field in RTC_GPIO_OUT_W1TS_REG */
WRITE_RTC_REG(RTC_GPIO_OUT_W1TS_REG, RTC_GPIO_OUT_DATA_W1TS_S + 2, 1, 1)
```

WRITE_RTC_FIELD(rtc_reg, field, value) Write immediate value into a field in rtc_reg, up to 8 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Set RTC_CNTL_ULP_CP_SLP_TIMER_EN field of RTC_CNTL_STATE0_REG to 0 */
WRITE_RTC_FIELD(RTC_CNTL_STATE0_REG, RTC_CNTL_ULP_CP_SLP_TIMER_EN, 0)
```

5.16.2 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 numer.

Here is an example of using labels and branches:

```
const ulp_insn_t program[] = {
    I_MOVI(R0, 34),          // R0 <- 34
    M_LABEL(1),              // label_1
    I_MOVI(R1, 32),          // R1 <- 32
    I_LD(R1, R1, 0),         // R1 <- RTC_SLOW_MEM[R1]
    I_MOVI(R2, 33),          // R2 <- 33
    I_LD(R2, R2, 0),         // R2 <- RTC_SLOW_MEM[R2]
```

(continues on next page)

(continued from previous page)

```
    I_SUBR(R3, R1, R2),      // R3 <- R1 - R2
    I_ST(R3, R0, 0),         // R3 -> RTC_SLOW_MEM[R0 + 0]
    I_ADDI(R0, R0, 1),       // R0++
    M_BL(1, 64),             // if (R0 < 64) goto label_1
    I_HALT(),
};
RTC_SLOW_MEM[32] = 42;
RTC_SLOW_MEM[33] = 18;
size_t load_addr = 0;
size_t size = sizeof(program)/sizeof(ulp_insn_t);
ulp_process_macros_and_load(load_addr, program, &size);
ulp_run(load_addr);
```

Functions

esp_err_t **ulp_process_macros_and_load**(uint32_t *load_addr*, **const** ulp_insn_t **program*, size_t **psize*)

Resolve all macro references in a program and load it into RTC memory.

Return

- ESP_OK on success
- ESP_ERR_NO_MEM if auxiliary temporary structure can not be allocated
- one of ESP_ERR_ULP_xxx if program is not valid or can not be loaded

Parameters

- `load_addr`: address where the program should be loaded, expressed in 32-bit words
- `program`: ulp_insn_t array with the program
- `psize`: size of the program, expressed in 32-bit words

esp_err_t **ulp_run**(uint32_t *entry_point*)

Run the program loaded into RTC memory.

Return ESP_OK on success

Parameters

- `entry_point`: entry point, expressed in 32-bit words

Error codes

ESP_ERR_ULP_BASE

Offset for ULP-related error codes

ESP_ERR_ULP_SIZE_TOO_BIG

Program doesn't fit into RTC memory reserved for the ULP

ESP_ERR_ULP_INVALID_LOAD_ADDR

Load address is outside of RTC memory reserved for the ULP

ESP_ERR_ULP_DUPLICATE_LABEL

More than one label with the same number was defined

ESP_ERR_ULP_UNDEFINED_LABEL
Branch instructions references an undefined label

ESP_ERR_ULP_BRANCH_OUT_OF_RANGE
Branch target is out of range of B instruction (try replacing with BX)

ULP coprocessor registers

ULP co-processor has 4 16-bit general purpose registers. All registers have same functionality, with one exception. R0 register is used by some of the compare-and-branch instructions as a source register.

These definitions can be used for all instructions which require a register.

R0
general purpose register 0

R1
general purpose register 1

R2
general purpose register 2

R3
general purpose register 3

ULP coprocessor instruction defines

I_DELAY(cycles_)
Delay (nop) for a given number of cycles

I_HALT()
Halt the coprocessor.

This instruction halts the coprocessor, but keeps ULP timer active. As such, ULP program will be restarted again by timer. To stop the program and prevent the timer from restarting the program, use I_END(0) instruction.

I_END()
Stop ULP program timer.

This is a convenience macro which disables the ULP program timer. Once this instruction is used, ULP program will not be restarted anymore until ulp_run function is called.

ULP program will continue running after this instruction. To stop the currently running program, use I_HALT().

I_ST(reg_val, reg_addr, offset_)
Store value from register reg_val into RTC memory.

The value is written to an offset calculated by adding value of reg_addr register and offset_ field (this offset is expressed in 32-bit words). 32 bits written to RTC memory are built as follows:

- bits [31:21] hold the PC of current instruction, expressed in 32-bit words
- bits [20:16] = 5'b1
- bits [15:0] are assigned the contents of reg_val

RTC_SLOW_MEM[addr + offset_] = { 5'b0, insn_PC[10:0], val[15:0] }

I_LD(reg_dest, reg_addr, offset_)
Load value from RTC memory into reg_dest register.

Loads 16 LSBs from RTC memory word given by the sum of value in reg_addr and value of offset_.

I_WR_REG (reg, low_bit, high_bit, val)
Write literal value to a peripheral register

reg[high_bit : low_bit] = val This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

I_RD_REG (reg, low_bit, high_bit)
Read from peripheral register into R0

R0 = reg[high_bit : low_bit] This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

I_BL (pc_offset, imm_value)
Branch relative if R0 less than immediate value.

pc_offset is expressed in words, and can be from -127 to 127 imm_value is a 16-bit value to compare R0 against

I_BGE (pc_offset, imm_value)
Branch relative if R0 greater or equal than immediate value.

pc_offset is expressed in words, and can be from -127 to 127 imm_value is a 16-bit value to compare R0 against

I_BXR (reg_pc)
Unconditional branch to absolute PC, address in register.

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXI (imm_pc)
Unconditional branch to absolute PC, immediate address.

Address imm_pc is expressed in 32-bit words.

I_BXZR (reg_pc)
Branch to absolute PC if ALU result is zero, address in register.

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXZI (imm_pc)
Branch to absolute PC if ALU result is zero, immediate address.

Address imm_pc is expressed in 32-bit words.

I_BXFR (reg_pc)
Branch to absolute PC if ALU overflow, address in register

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXFI (imm_pc)
Branch to absolute PC if ALU overflow, immediate address

Address imm_pc is expressed in 32-bit words.

I_ADDR (reg_dest, reg_src1, reg_src2)
Addition: dest = src1 + src2

I_SUBR (reg_dest, reg_src1, reg_src2)
Subtraction: dest = src1 - src2

I_ANDR (reg_dest, reg_src1, reg_src2)
Logical AND: dest = src1 & src2

I_ORR (reg_dest, reg_src1, reg_src2)
Logical OR: dest = src1 | src2

I_MOVR (reg_dest, reg_src)
Copy: dest = src

I_LSHR (reg_dest, reg_src, reg_shift)
Logical shift left: dest = src << shift

I_RSHR (reg_dest, reg_src, reg_shift)
Logical shift right: dest = src >> shift

I_ADDI (reg_dest, reg_src, imm_)
Add register and an immediate value: dest = 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.

5.16.3 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.

5.16.4 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.

5.16.5 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:

```
                        .global measurement_count
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);
```

5.16.6 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 */
```

5.16.7 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.

5.17 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.

5.17.1 Add normal test cases

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.

5.17.2 Add multiple devices test cases

The normal test cases will be executed on one DUT (Device Under Test). Components need to communicate with each other (like GPIO, SPI . . .) can't be tested with normal test cases. Multiple devices test cases support writing and running test with multiple DUTs.

Here's an example of multiple devices test case:

```
void gpio_master_test()
{
    gpio_config_t slave_config = {
            .pin_bit_mask = 1 << MASTER_GPIO_PIN,
            .mode = GPIO_MODE_INPUT,
    };
    gpio_config(&slave_config);
    unity_wait_for_signal("output high level");
    TEST_ASSERT(gpio_get_level(MASTER_GPIO_PIN) == 1);
}

void gpio_slave_test()
{
    gpio_config_t master_config = {
            .pin_bit_mask = 1 << SLAVE_GPIO_PIN,
            .mode = GPIO_MODE_OUTPUT,
    };
    gpio_config(&master_config);
    gpio_set_level(SLAVE_GPIO_PIN, 1);
    unity_send_signal("output high level");
}

TEST_CASE_MULTIPLE_DEVICES("gpio multiple devices test example", "[driver]", gpio_
↪master_test, gpio_slave_test);
```

The macro `TEST_CASE_MULTIPLE_DEVICES` is used to declare multiple devices test cases. First argument is test case name, second argument is test case description. From the third argument, upto 5 test functions can be defined, each function will be the entry point of tests running on each DUT.

Running test cases from different DUTs could require synchronizing between DUTs. We provide `unity_wait_for_signal` and `unity_send_signal` to support synchronizing with UART. As the secnario in the above example, slave should get GPIO level after master set level. DUT UART console will prompt and requires user interaction:

DUT1 (master) console:

```
Waiting for signal: [output high level]!
Please press "Enter" key to once any board send this signal.
```

DUT2 (slave) console:

```
Send signal: [output high level]!
```

Once the signal is set from DUT2, you need to press "Enter" on DUT1, then DUT1 unblocks from `unity_wait_for_signal` and starts to change GPIO level.

5.17.3 Add multiple stages test cases

The normal test cases are expected to finish without reset (or only need to check if reset happens). Sometimes we want to run some specific test after certain kinds of reset. For example, we want to test if reset reason is correct after

wakeup from deep sleep. We need to create deep sleep reset first and then check the reset reason. To support this, we can define multiple stages test case, to group a set of test functions together:

```
static void trigger_deepsleep(void)
{
    esp_sleep_enable_timer_wakeup(2000);
    esp_deep_sleep_start();
}

void check_deepsleep_reset_reason()
{
    RESET_REASON reason = rtc_get_reset_reason(0);
    TEST_ASSERT(reason == DEEPSLEEP_RESET);
}

TEST_CASE_MULTIPLE_STAGES("reset reason check for deepsleep", "[esp32]", trigger_
↪deepsleep, check_deepsleep_reset_reason);
```

Multiple stages test cases present a group of test functions to users. It need user interactions (select case and select different stages) to run the case.

5.17.4 Building unit test app

Follow the setup instructions in the top-level esp-idf README. Make sure that IDF_PATH environment variable is set to point to the path of esp-idf top-level directory.

Change into tools/unit-test-app directory to configure and build it:

- *make menuconfig* - configure unit test app.
- *make TESTS_ALL=1* - build unit test app with tests for each component having tests in the `test` subdirectory.
- *make TEST_COMPONENTS='xxx'* - build unit test app with tests for specific components.

When the build finishes, it will print instructions for flashing the chip. You can simply run `make flash` to flash all build output.

You can also run `make flash TESTS_ALL=1` or `make TEST_COMPONENTS='xxx'` to build and flash. Everything needed will be rebuilt automatically before flashing.

Use menuconfig to set the serial port for flashing.

5.17.5 Running unit tests

After flashing reset the ESP32 and it will boot the unit test app.

When unit test app is idle, press "Enter" will make it print test menu with all available tests:

```
Here's the test menu, pick your combo:
(1)     "esp_ota_begin() verifies arguments" [ota]
(2)     "esp_ota_get_next_update_partition logic" [ota]
(3)     "Verify bootloader image in flash" [bootloader_support]
(4)     "Verify unit test app image" [bootloader_support]
(5)     "can use new and delete" [cxx]
(6)     "can call virtual functions" [cxx]
(7)     "can use static initializers for non-POD types" [cxx]
(8)     "can use std::vector" [cxx]
(9)     "static initialization guards work as expected" [cxx]
```

(continues on next page)

(continued from previous page)

```
(10)    "global initializers run in the correct order" [cxx]
(11)    "before scheduler has started, static initializers work correctly" [cxx]
(12)    "adc2 work with wifi" [adc]
(13)    "gpio master/slave test example" [ignore][misc][test_env=UT_T2_1][multi_
↪device]
        (1)     "gpio_master_test"
        (2)     "gpio_slave_test"
(14)    "SPI Master clockdiv calculation routines" [spi]
(15)    "SPI Master test" [spi][ignore]
(16)    "SPI Master test, interaction of multiple devs" [spi][ignore]
(17)    "SPI Master no response when switch from host1 (HSPI) to host2 (VSPI)" [spi]
(18)    "SPI Master DMA test, TX and RX in different regions" [spi]
(19)    "SPI Master DMA test: length, start, not aligned" [spi]
(20)    "reset reason check for deepsleep" [esp32][test_env=UT_T2_1][multi_stage]
        (1)     "trigger_deepsleep"
        (2)     "check_deepsleep_reset_reason"
```

Normal case will print the case name and description. Master slave cases will also print the sub-menu (the registered test function names).

Test cases can be run by inputting one of the following:

- Test case name in quotation marks to run a single test case
- Test case index to run a single test case
- Module name in square brackets to run all test cases for a specific module
- An asterisk to run all test cases

`[multi_device]` and `[multi_stage]` tags tell the test runner whether a test case is a multiple devices or multiple stages test case. These tags are automatically added by `` `TEST_CASE_MULTIPLE_STAGES `` and `TEST_CASE_MULTIPLE_DEVICES` macros.

After you select a multiple devices test case, it will print sub menu:

```
Running gpio master/slave test example...
gpio master/slave test example
        (1)     "gpio_master_test"
        (2)     "gpio_slave_test"
```

You need to input number to select the test running on the DUT.

Similar to multiple devices test cases, multiple stages test cases will also print sub menu:

```
Running reset reason check for deepsleep...
reset reason check for deepsleep
        (1)     "trigger_deepsleep"
        (2)     "check_deepsleep_reset_reason"
```

First time you execute this case, input `1` to run first stage (trigger deepsleep). After DUT is rebooted and able to run test cases, select this case again and input `2` to run the second stage. The case only passes if the last stage passes and all previous stages trigger reset.

5.18 Console

ESP-IDF provides `console` component, which includes building blocks needed to develop an interactive console over serial port. This component includes following facilities:

- Line editing, provided by linenoise library. This includes handling of backspace and arrow keys, scrolling through command history, command auto-completion, and argument hints.
- Splitting of command line into arguments.
- Argument parsing, provided by argtable3 library. This library includes APIs useful for parsing GNU style command line arguments.
- Functions for registration and dispatching of commands.

These facilities can be used together or independently. For example, it is possible to use line editing and command registration features, but use `getopt` or custom code for argument parsing, instead of argtable3. Likewise, it is possible to use simpler means of command input (such as `fgets`) together with the rest of the means for command splitting and argument parsing.

5.18.1 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.

5.18.2 Splitting of command line into arguments

`console` component provides `esp_console_split_argv` function to split command line string into arguments. The function returns the number of arguments found (`argc`) and fills an array of pointers which can be passed as `argv` argument to any function which accepts arguments in `argc, argv` format.

The command line is split into arguments according to the following rules:

- Arguments are separated by spaces
- If spaces within arguments are required, they can be escaped using `\` (backslash) character.
- Other escape sequences which are recognized are `\\` (which produces literal backslash) and `\"`, which produces a double quote.
- Arguments can be quoted using double quotes. Quotes may appear only in the beginning and at the end of the argument. Quotes within the argument must be escaped as mentioned above. Quotes surrounding the argument are stripped by `esp_console_split_argv` function.

Examples:

- `abc def 1 20 .3` [`abc`, `def`, `1`, `20`, `.3`]

- `abc "123 456" def` [`abc`, `123 456`, `def`]
- `` `a\ b\\c\" `` [`a b\c"`]

5.18.3 Argument parsing

For argument parsing, `console` component includes argtable3 library. Please see tutorial for an introduction to argtable3. Github repository also includes examples.

5.18.4 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.

5.18.5 Example

Example application illustrating usage of the `console` component is available in `examples/system/console` directory. This example shows how to initialize UART and VFS functions, set up linenoise library, read and handle commands from UART, and store command history in Flash. See README.md in the example directory for more details.

5.19 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 preceeding line number, commands entered are executed immediately; lines with a prefixed line number are stored as part of a program.

5.19.1 Full list of supported statements and functions

System

- BYE - *exits Basic, reboots ESP32, retries booting from flash*
- END - *stops execution from the program, also "STOP"*
- MEM - *displays memory usage statistics*
- NEW - *clears the current program*
- RUN - *executes the current program*

IO, Documentation

- PEEK(address) - *get a 32-bit value from a memory address*
- POKE - *write a 32-bit value to memory*
- USR(addr, arg1, ..) - *Execute a machine language function*
- PRINT expression - *print out the expression, also "?"*
- PHEX expression - *print expression as a hex number*
- REM stuff - *remark/comment, also "'"*

Expressions, Math

- A=V, LET A=V - *assign value to a variable*
- +, -, *, / - *Math*
- <,<=,=,<>,!=,>=,> - *Comparisons*
- ABS(expression) - *returns the absolute value of the expression*
- RSEED(v) - *sets the random seed to v*
- RND(m) - *returns a random number from 0 to m*
- A=1234 - * Assign a decimal value*
- A=&h1A2 - * Assign a hex value*
- A=&b1001 - *Assign a binary value*

Control

- IF expression statement - *perform statement if expression is true*
- FOR variable = start TO end - *start for block*
- FOR variable = start TO end STEP value - *start for block with step*
- NEXT - *end of for block*

- GOTO linenumber - *continue execution at this line number*
- GOSUB linenumber - *call a subroutine at this line number*
- RETURN - *return from a subroutine*
- DELAY - *Delay a given number of milliseconds*

Pin IO

- IODIR - *Set a GPIO-pin as an output (1) or input (0)*
- IOSET - *Set a GPIO-pin, configured as output, to high (1) or low (0)*
- IOGET - *Get the value of a GPIO-pin*

5.19.2 Example programs

Here are a few example commands and programs to get you started…

Read UART_DATE register of uart0

```
> PHEX PEEK(&h3FF40078)
15122500
```

Set GPIO2 using memory writes to GPIO_OUT_REG

Note: you can do this easier with the IOSET command

```
> POKE &h3FF44004,PEEK(&h3FF44004) OR &b100
```

Get value of GPIO0

```
> IODIR 0,0
> PRINT IOGET(0)
0
```

Blink LED

Hook up an LED between GPIO2 and ground. When running the program, the LED should blink 10 times.

```
10 IODIR 2,1
20 FOR A=1 TO 10
30 IOSET 2,1
40 DELAY 250
50 IOSET 2,0
60 DELAY 250
70 NEXT A
RUN
```

5.19.3 Credits

The ROM console is based on "TinyBasicPlus" by Mike Field and Scott Lawrence, which is based on "68000 Tiny-Basic" by Gordon Brandly

5.20 Wi-Fi Driver

5.20.1 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.

5.20.2 ESP32 Wi-Fi Feature List

- Support Station-only mode, SoftAP-only mode, Station/SoftAP-coexistence mode
- Support IEEE-802.11B, IEEE-802.11G, IEEE802.11N and APIs to configure the protocol mode
- Support WPA/WPA2/WPA2-Enterprise and WPS
- Support AMPDU, HT40, QoS and other key features
- Support Modem-sleep
- Support an Espressif-specific protocol which, in turn, supports up to **1 km** of data traffic
- Up to 20 MBit/sec TCP throughput and 30 MBit/sec UDP throughput over the air
- Support Sniffer
- Support set fast_crypto algorithm and normal algorithm switch which used in wifi connect
- Support both fast scan and all channel scan feature
- Support multiple antennas
- Support channel state information

5.20.3 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*>

5.20.4 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_OK means that the API returns successfully
- Recoverable errors, such as ESP_ERR_NO_MEM, etc.
- Non-recoverable, non-critical errors
- Non-recoverable, critical errors

Whether the error is critical or not depends on the API and the application scenario, and it is defined by the API user.

The primary principle to write a robust application with Wi-Fi API is to always check the error code and write the error-handling code. Generally, the error-handling code can be used:

- for recoverable errors, in which case you can write a recoverable-error code. For example, when esp_wifi_start returns ESP_ERR_NO_MEM, the recoverable-error code vTaskDelay can be called, in order to get a 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.

5.20.5 ESP32 Wi-Fi Programming Model

The ESP32 Wi-Fi programming model is depicted as follows:

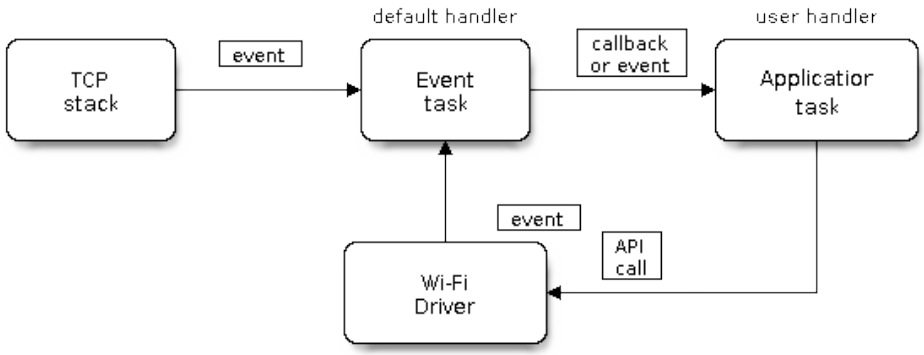


Fig. 24: Wi-Fi Programming Model

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.

5.20.6 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_deauth_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.

5.20.7 ESP32 Wi-Fi Station General Scenario

Below is a "big scenario" which describes some small scenarios in Station mode:

1. Wi-Fi/LwIP Init Phase

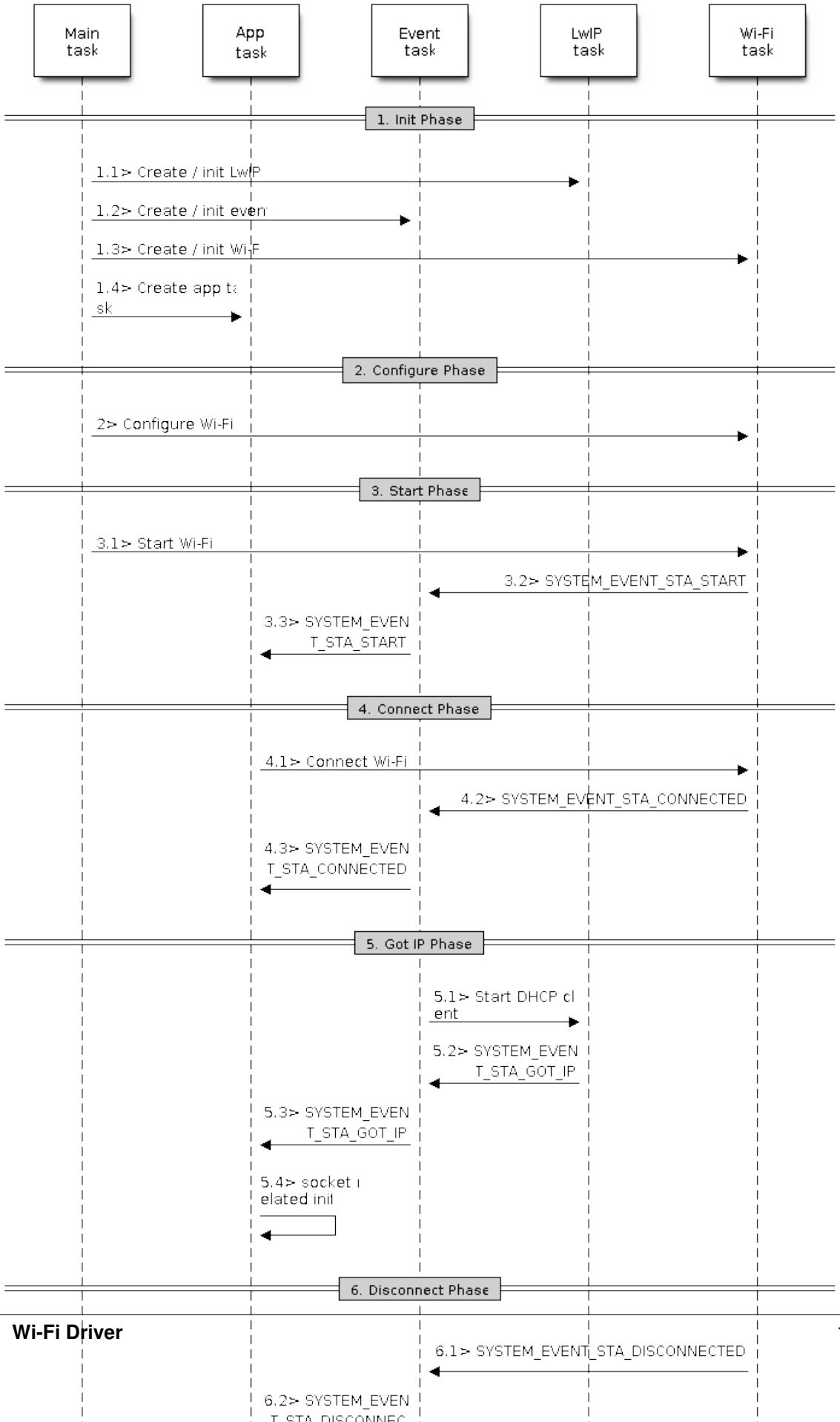
- s1.1: The main task calls tcpip_adapter_init() to create an LwIP core task and initialize LwIP-related work.
- s1.2: The main task calls esp_event_loop_init() to create a system Event task and initialize an application event's callback function. In the scenario above, the application event's callback 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


Main task
App task
Event task
LwIP task
Wi-Fi task
1. Init Phase
1.1> Create / init LwIP
1.2> Create / init even
1.3> Create / init Wi-F
1.4> Create app ta sk
2. Configure Phase
2> Configure Wi-Fi
3. Start Phase
3.1> Start Wi-Fi
3.2> SYSTEM_EVENT_STA_START
3.3> SYSTEM_EVEN T_STA_START
4. Connect Phase
4.1> Connect Wi-Fi
4.2> SYSTEM_EVENT_STA_CONNECTED
4.3> SYSTEM_EVEN T_STA_CONNECTED
5. Got IP Phase
5.1> Start DHCP cl ent
5.2> SYSTEM_EVEN T_STA_GOT_IP
5.3> SYSTEM_EVEN T_STA_GOT_IP
5.4> socket r elated inil
6. Disconnect Phase
6.1> SYSTEM_EVENT_STA_DISCONNECTED
6.2> SYSTEM_EVEN

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.

5.20.8 ESP32 Wi-Fi soft-AP General Scenario

Below is a "big scenario" which describes some small scenarios in Soft-AP mode:

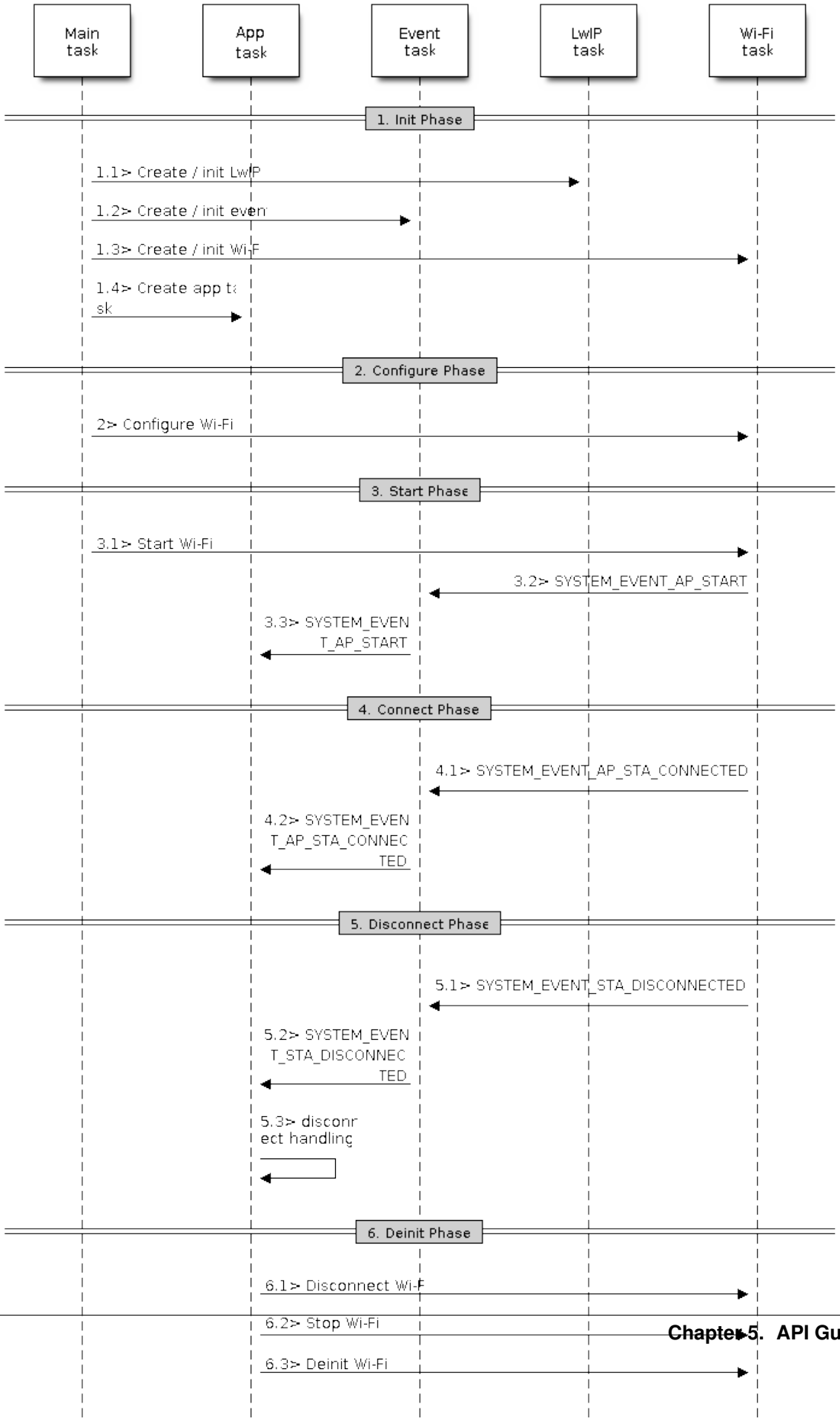
5.20.9 ESP32 Wi-Fi Scan

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 scan_type field of wifi_scan_config_t.
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 channel field of wifi_scan_config_t is set to 0, it is an all-channel scan.
Specific Channel Scan	It scans specific channels only. If the channel field of wifi_scan_config_t 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:


Main task
App task
Event task
LwIP task
Wi-Fi task
1. Init Phase
1.1> Create / init LwIP
1.2> Create / init even
1.3> Create / init Wi-F
1.4> Create app ta sk
2. Configure Phase
2> Configure Wi-Fi
3. Start Phase
3.1> Start Wi-Fi
3.2> SYSTEM_EVENT_AP_START
3.3> SYSTEM_EVEN T_AP_START
4. Connect Phase
4.1> SYSTEM_EVENT_AP_STA_CONNECTED
4.2> SYSTEM_EVEN T_AP_STA_CONNEC TED
5. Disconnect Phase
5.1> SYSTEM_EVENT_STA_DISCONNECTED
5.2> SYSTEM_EVEN T_STA_DISCONNEC TED
5.3> disconn ect handling
6. Deinit Phase
6.1> Disconnect Wi-F
6.2> Stop Wi-Fi
6.3> Deinit Wi-Fi

- All-Channel Background Active Scan
- All-Channel Background Passive Scan
- All-Channel Foreground Active Scan
- All-Channel Foreground Passive Scan
- Specific-Channel Background Active Scan
- Specific-Channel Background Passive Scan
- Specific-Channel Foreground Active Scan
- Specific-Channel Foreground Passive Scan

Scan Configuration

The scan type and other scan attributes are configured by esp_wifi_scan_start. The table below provides a detailed description of wifi_scan_config_t.

Field	Description
ssid	If the SSID is not NULL, it is only the AP with the same SSID that can be scanned.
bssid	If the BSSID is not NULL, it is only the AP with the same BSSID that can be scanned.
channel	If "channel" is 0, there will be an all-channel scan; otherwise, there will be a specific-channel scan.
show_hidden	If "show_hidden" is 0, the scan ignores the AP with a hidden SSID; otherwise, the scan considers the hidden AP a normal one.
scan_type	If "scan_type" is WIFI_SCAN_TYPE_ACTIVE, the scan is "active"; otherwise, it is a "passive" one.
scan_time	This field is used to control how long the scan dwells on each channel. For passive scans, scan_time.passive designates the dwell time for each channel. For active scans, dwell times for each channel are listed in the table below. Here, min is short for scan time.active.min and max is short for scan_time.active.max. • min=0, max=0: scan dwells on each channel for 120 ms. • min>0, max=0: scan dwells on each channel for 120 ms. • min=0, max>0: scan dwells on each channel for `max` ms. • min>0, max>0: the minimum time the scan dwells on each channel is `min` ms. If no AP is found during this time frame, the scan switches to the next channel. Otherwise, the scan dwells on the channel for `max` ms. If you want to improve the performance of the the scan, you can try to modify these two parameters.

Scan All APs In All Channels(foreground)

Scenario:

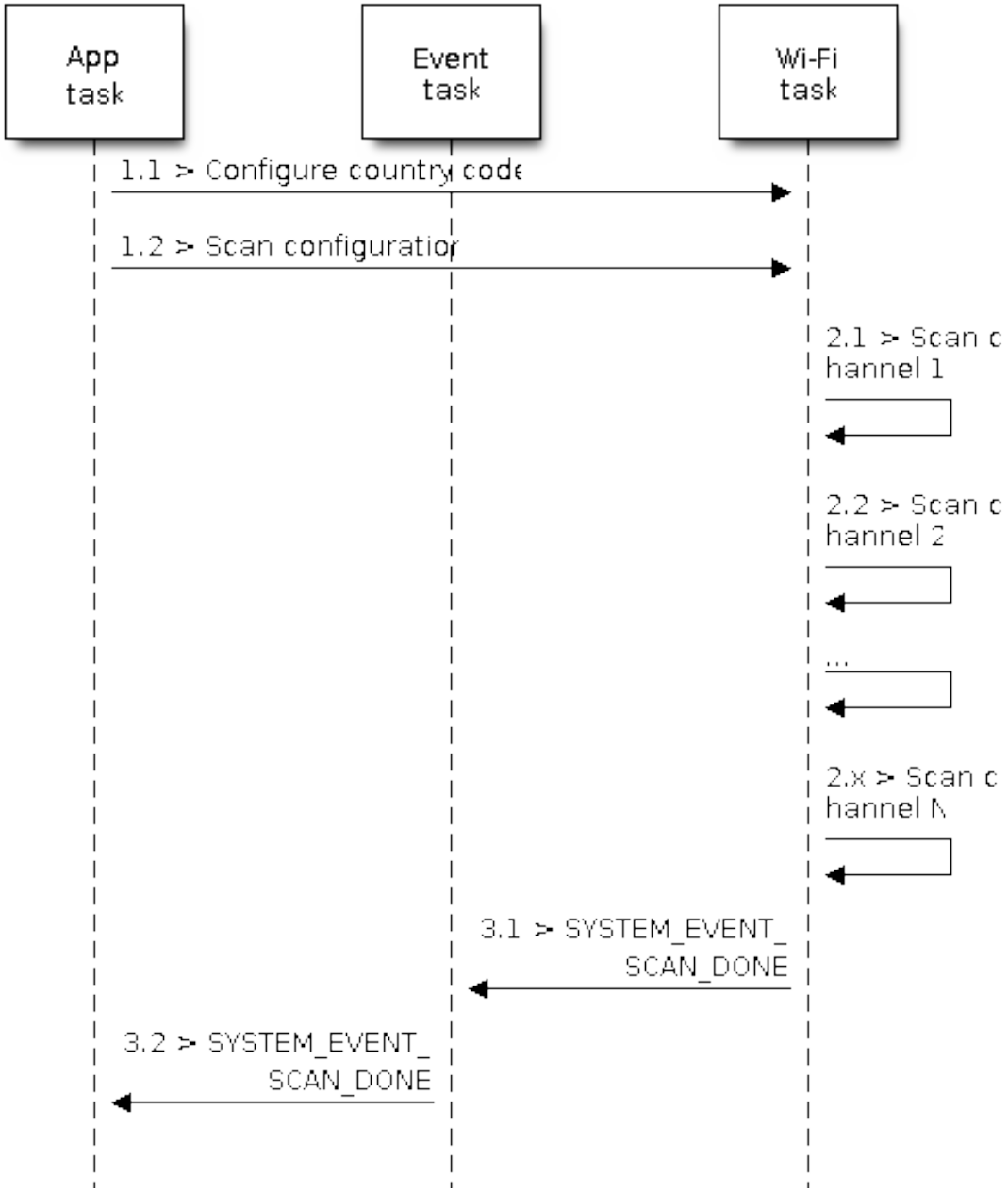


Fig. 27: Foreground Scan of all Wi-Fi Channels

The scenario above describes an all-channel, foreground scan. The foreground scan can only occur in Station mode where the station does not connect to any AP. Whether it is a foreground or background scan is totally determined by the Wi-Fi driver, and cannot be configured by the application.

Detailed scenario description:

Scan Configuration Phase

- s1.1: Call esp_wifi_set_country() to set the country code. For China/Japan, the channel value ranges from 1 to 14; for the USA, it ranges from 1 to 11; and for Europe, it ranges from 1 to 13. The default country is China.
- s1.2: Call esp_wifi_scan_start() to configure the scan. To do so, you can refer to <*Scan Configuration*>. Since this is an all-channel scan, just set the SSID/BSSID/channel to 0.

Wi-Fi Driver's Internal Scan Phase

- s2.1: The Wi-Fi driver switches to channel 1, in case the scan type is WIFI_SCAN_TYPE_ACTIVE, and broadcasts a probe request. Otherwise, the Wi-Fi will wait for a beacon from the APs. The Wi-Fi driver will stay in channel 1 for some time. The dwell time is configured in min/max time, with default value being 120 ms.
- s2.2: The Wi-Fi driver switches to channel 2 and performs the same operation as in step 2.1.
- s2.3: The Wi-Fi driver scans the last channel N, where N is determined by the country code which is configured in step 1.1.

Scan-Done Event Handling Phase

- s3.1: When all channels are scanned, <*SYSTEM_EVENT_SCAN_DONE*> will arise.
- s3.2: The application's event callback function notifies the application task that <*SYSTEM_EVENT_SCAN_DONE*> is received. esp_wifi_scan_get_ap_num() is called to get the number of APs that have been found in this scan. Then, it allocates enough entries and calls esp_wifi_scan_get_ap_records() to get the AP records. Please note that the AP records in the Wi-Fi driver will be freed, once esp_wifi_scan_get_ap_records() is called. Do not call esp_wifi_scan_get_ap_records() twice for a single scan-done event. If esp_wifi_scan_get_ap_records() is not called when the scan-done event occurs, the AP records allocated by the Wi-Fi driver will not be freed. So, make sure you call esp_wifi_scan_get_ap_records(), yet only once.

Scan All APs on All Channels(background)

Scenario:

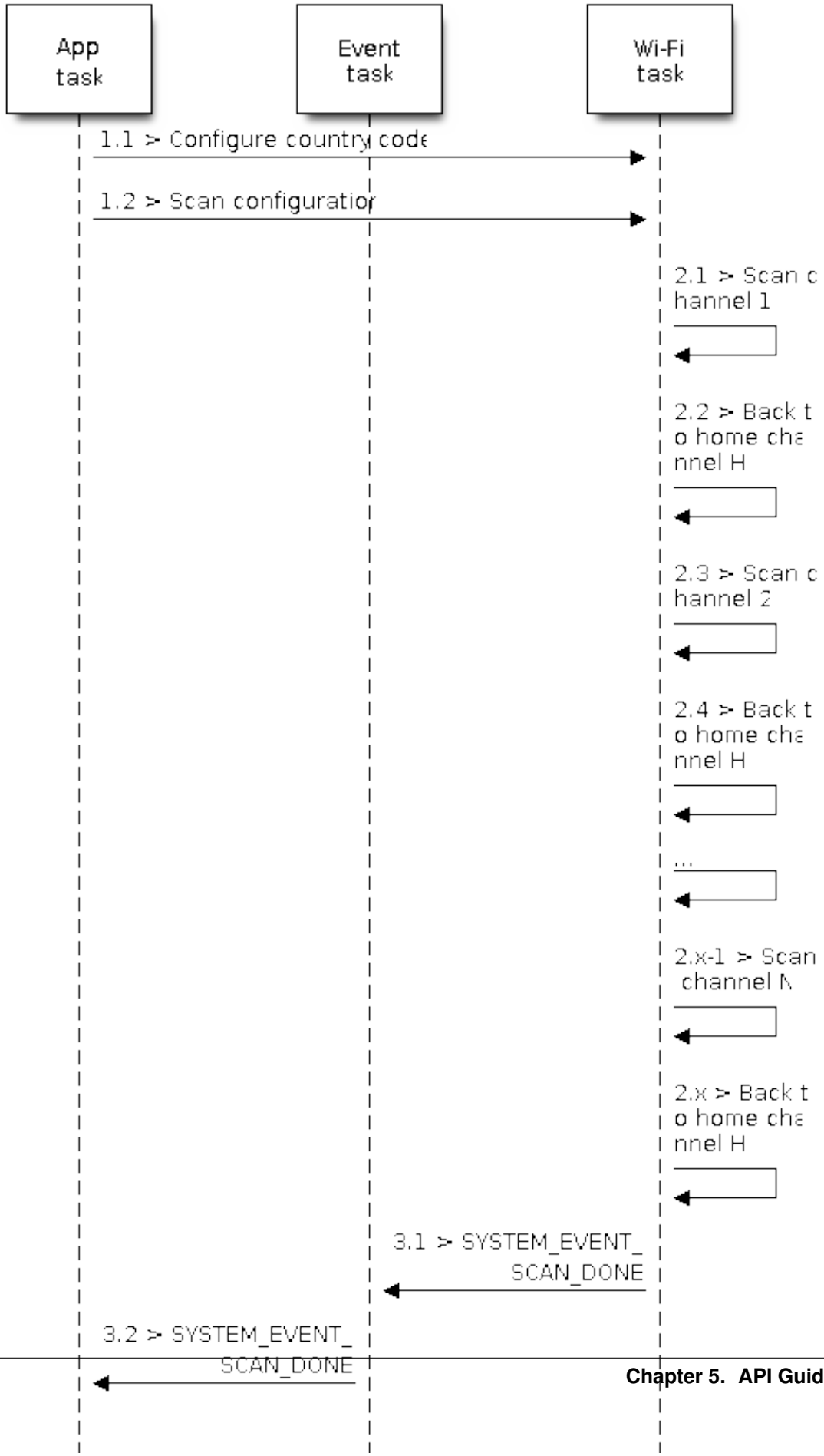
The scenario above is an all-channel background scan. Compared to *Scan All APs In All Channels(foreground)* , the difference in the all-channel background scan is that the Wi-Fi driver will scan the back-to-home channel for 30 ms before it switches to the next channel to give the Wi-Fi connection a chance to transmit/receive data.

Scan for a Specific AP in All Channels

Scenario:

This scan is similar to *Scan All APs In All Channels(foreground)*. The differences are:

- s1.1: In step 1.2, the target AP will be configured to SSID/BSSID.
- s2.1~s2.N: Each time the Wi-Fi driver scans an AP, it will check whether it is a target AP or not. If it is a target AP, then the scan-done event will arise and scanning will end; otherwise, the scan will continue. Please note that the first scanned channel may not be channel 1, because the Wi-Fi driver optimizes the scanning sequence.


App task
Event task
Wi-Fi task
1.1 > Configure country code
1.2 > Scan configuration
2.1 > Scan channel 1
2.2 > Back to home channel H
2.3 > Scan channel 2
2.4 > Back to home channel H
...
2.x-1 > Scan channel N
2.x > Back to home channel H
3.1 > SYSTEM_EVENT_SCAN_DONE
3.2 > SYSTEM_EVENT_SCAN_DONE

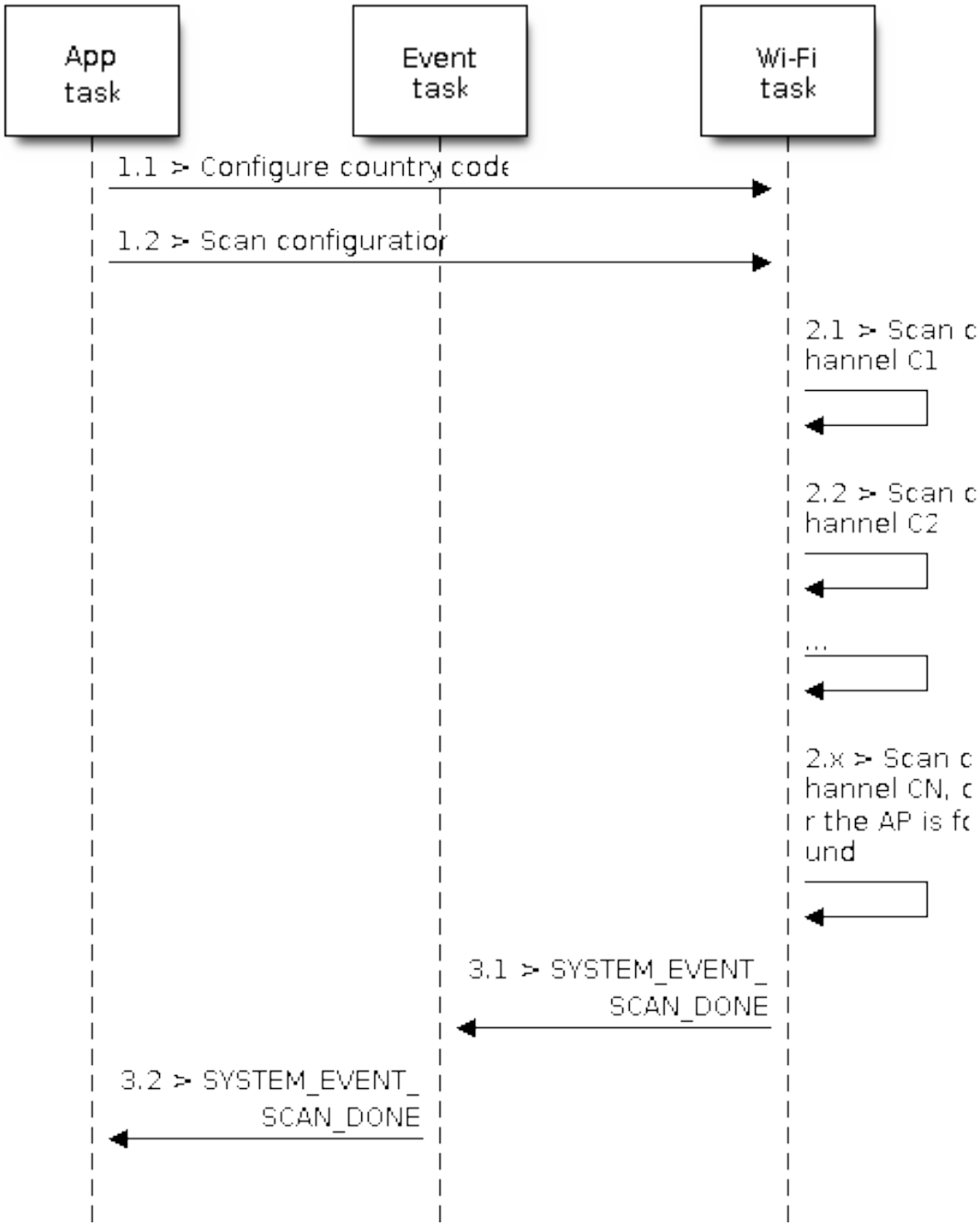


Fig. 29: Scan of specific Wi-Fi Channels

If there are more than one APs which match the target AP info, for example, if we happen to scan two APs whose SSID is "ap", then only the first AP will be returned. However, if the first AP is not the one you want, e.g., if its password is wrong, then the Wi-Fi driver will detect a four-way handshake failure and try to scan the next AP. If two APs have the same SSID, BSSID and password, then the Wi-Fi driver will choose the first one to connect to.

You can scan a specific AP, or all of them, in any given channel. These two scenarios are very similar.

Scan in Wi-Fi Connect

When esp_wifi_connect() is called, then the Wi-Fi driver will try to scan the configured AP first. The scan in "Wi-Fi Connect" is the same as *Scan for a Specific AP In All Channels*, except that no scan-done event will be generated when the scan is completed. If the target AP is found, then the Wi-Fi driver will start the Wi-Fi connection; otherwise, <*SYSTEM_EVENT_STA_DISCONNECTED*> will be generated. Refer to *Scan for a Specific AP in All Channels*

Scan In Blocked Mode

If the block parameter of esp_wifi_scan_start() is true, then the scan is a blocked one, and the application task will be blocked until the scan is done. The blocked scan is similar to an unblocked one, except that no scan-done event will arise when the blocked scan is completed.

Parallel Scan

Two application tasks may call esp_wifi_scan() 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.

5.20.10 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.

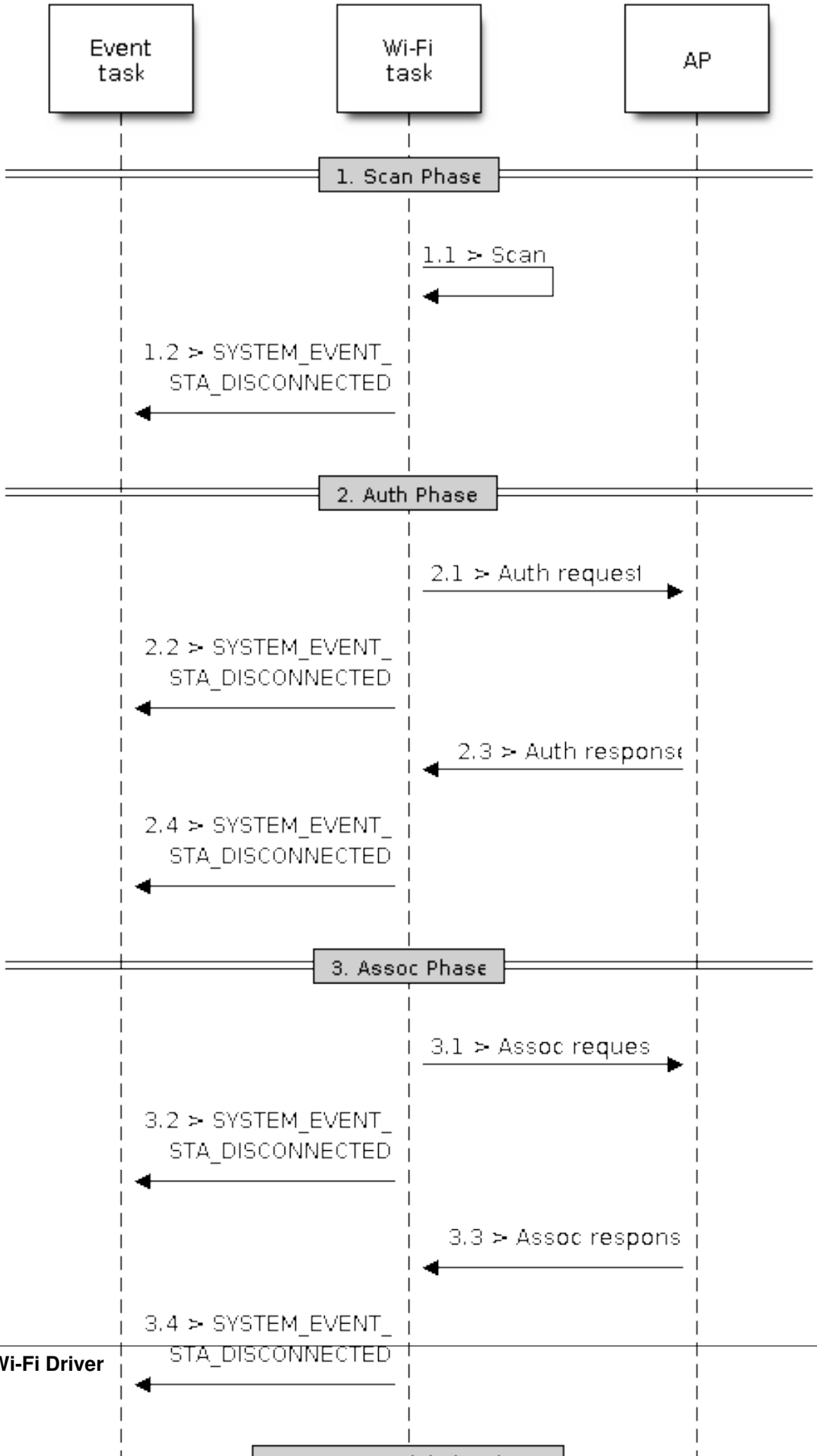
Scenario:

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.


Event task
Wi-Fi task
AP
1. Scan Phase
1.1 > Scan
1.2 > SYSTEM_EVENT_ STA_DISCONNECTED
2. Auth Phase
2.1 > Auth request
2.2 > SYSTEM_EVENT_ STA_DISCONNECTED
2.3 > Auth response
2.4 > SYSTEM_EVENT_ STA_DISCONNECTED
3. Assoc Phase
3.1 > Assoc reques
3.2 > SYSTEM_EVENT_ STA_DISCONNECTED
3.3 > Assoc respons
3.4 > SYSTEM_EVENT_ STA_DISCONNECTED

- 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 ieee802.11-2012. (For more information, refer to the standard mentioned above.) The last column is a description of the reason.

Reason code	Value	Mapped To	Description
UNSPECIFIED	1	1	Generally, it means an internal failure, e.g., the memory runs out, the internal TX fails, or the reason is received from the remote side, etc.
AUTH_EXPIRE	2	2	The previous authentication is no longer valid. For the ESP32 Station, this reason is reported when: • auth is timed out • the reason is received from the soft-AP. For the ESP32 SoftAP, this reason is reported when: • the soft-AP has not received any packets from the station in the past five minutes. • the soft-AP is stopped by calling esp_wifi_stop(). • the station is deauthed by calling esp_wifi_deauth_sta()
AUTH_LEAVE	3	3	De-authenticated, because the sending STA is leaving (or has left). For the ESP32 Station, this reason is reported when: • it is received from the soft-AP.
ASSOC_EXPIRE	4	4	Disassociated due to inactivity. For the ESP32 Station, this reason is reported when: • it is received from the soft-AP. For the ESP32 Soft-AP, this reason is reported when: • the soft-AP has not received any packets from the station in the past five minutes.
5.20. Wi-Fi Driver			• the soft-AP is stopped by calling esp_wifi_stop(). • the station is de-

5.20.11 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	NULL mode: 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 esp_wifi_deinit() to unload the whole Wi-Fi driver.
WIFI_MODE_STA	Station mode: in this mode, esp_wifi_start() will init the internal station data, while the station's interface is ready for the RX and TX Wi-Fi data. After esp_wifi_connect() is called, the STA will connect to the target AP.
WIFI_MODE_AP	Soft-AP mode: in this mode, esp_wifi_start() 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 broad- casting beacons, and the soft-AP is ready to get connected to other stations.
WIFI_MODE_APSTA	Station-AP coexistence mode: in this mode, esp_wifi_start() 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 *Wi-Fi Channel Management*.

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.
pass-word	Password of the target AP
bssid_set	If bssid_set is 0, the station connects to the AP whose SSID is the same as the field "ssid", while the field "bssid" is ignored. In all other cases, the station connects to the AP whose SSID is the same as the "ssid" field, while its BSSID is the same the "bssid" field .
bssid	This is valid only when bssid_set is 1; see field "bssid_set".
chan-nel	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_aabbcc, where "aabbcc" 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*>.
authmode	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_PROTOC 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. **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**

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.

5.20.12 ESP32 Wi-Fi Power-saving Mode

Currently, ESP32 Wi-Fi supports the Modem-sleep mode which refers to the legacy power-saving mode in the IEEE 802.11 protocol. Modem-sleep mode works in Station-only mode and the station must connect to the AP first. If the Modem-sleep mode is enabled, station will switch between active and sleep state periodically. In sleep state, RF, PHY and BB are turned off in order to reduce power consumption. Station can keep connection with AP in modem-sleep mode.

Modem-sleep mode includes minimum and maximum power save modes. In minimum power save mode, station wakes up every DTIM to receive beacon. Broadcast data will not be lost because it is transmitted after DTIM. However, it can not save much more power if DTIM is short for DTIM is determined by AP. In maximum power save mode, station wakes up every listen interval to receive beacon. Broadcast data may be lost because station may be in sleep state at DTIM time. If listen interval is longer, more power is saved but broadcast data is more easy to lose. Listen interval can be configured by calling API `esp_wifi_set_config()` before connecting to AP.

Call `esp_wifi_set_ps(WIFI_PS_MIN_MODEM)` to enable Modem-sleep minimum power save mode or `esp_wifi_set_ps(WIFI_PS_MAX_MODEM)` to enable Modem-sleep maximum power save mode after calling `esp_wifi_init()`. When station connects to AP, Modem-sleep will start. When station disconnects from AP, Modem-sleep will stop.

5.20.13 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().

5.20.14 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	**130 MBit/sec**
Raw 802.11 Packet TX	N/A	**130 MBit/sec**
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*>.

5.20.15 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 non-QoS data frame.

It cannot be used for sending encrypted or QoS frames.

Preconditions of Using esp_wifi_80211_tx

- The Wi-Fi mode is Station, or SoftAP, or Station+SoftAP.
- Either esp_wifi_set_promiscuous(true), or esp_wifi_start(), or both of these APIs return ESP_OK. This is because we need to make sure that Wi-Fi hardware is initialized before esp_wifi_80211_tx() is called. In ESP32, both esp_wifi_set_promiscuous(true) and esp_wifi_start() can trigger the initialization of Wi-Fi hardware.
- The parameters of esp_wifi_80211_tx are hereby correctly provided.

Data rate

- If there is no WiFi connection, the data rate is 1Mbps.
- If there is WiFi connection and the packet is from station to SoftAP or from SoftAP to station, the data rate is same as the WiFi connection. Otherwise the data rate is 1Mbps.

Side-Effects to Avoid in Different Scenarios

Theoretically, if we do not consider the side-effects the API imposes on the Wi-Fi driver or other stations/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
No WiFi connection	In this scenario, no Wi-Fi connection is set up, so there are no side-effects on the Wi-Fi driver. If en_sys_seq==true, the Wi-Fi driver is responsible for the sequence control. If en_sys_seq==false, the application needs to ensure that the buffer has the correct sequence. Theoretically, the MAC address can be any address. However, this may impact other stations/soft-APs with the same MAC/BSSID. Side-effect example#1 The application calls esp_wifi_80211_tx to send a beacon with BSSID == mac_x in SoftAP mode, but the mac_x is not the MAC of the SoftAP interface. Moreover, there is another soft-AP, say "other-AP", whose bssid is mac_x. If this happens, an "unexpected behavior" may occur, because the stations which connect to the "other-AP" cannot figure out whether the beacon is from the "other-AP" or the esp_wifi_80211_tx. To avoid the above-mentioned side-effects, we recommend that: • If esp_wifi_80211_tx is called in Station mode, the first MAC should be a multicast MAC or the exact target-device's MAC, while the second MAC should be that of the station interface. • If esp_wifi_80211_tx is called in 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. The recommendations above are only for avoiding side-effects and can be ignored when there are good reasons for doing this.
Have WiFi connection	When the Wi-Fi connection is already set up, and the sequence is controlled by the application, the latter may impact the sequence control of the Wi-Fi connection, as a whole. So, the en_sys_seq need to be true, otherwise ESP_ERR_WIFI_ARG is returned. The MAC-address recommendations in the "No WiFi connection" scenario also apply to this scenario. If the WiFi mode is station mode and the MAC address1 is the MAC of AP to which the station is connected, the MAC address2 is the MAC of station interface, we say the packets is from the station to SoftAP. On the other hand, if the WiFi mode is SoftAP mode and the MAC address1 is the MAC of the station who connects to this SoftAP, the MAC address2 is the MAC of SoftAP interface, we say the packet is from the SoftAP to station. To avoid conflicting with WiFi connections, the following checks are applied: • If the packet type is data and is from the station to SoftAP, the ToDS bit in ieee80211 frame control should be 1, the FromDS bit should be 0, otherwise the packet will be discarded by WiFi driver. • If the packet type is data and is from the softAP to station, the ToDS bit in ieee80211 frame control should be 0, the FromDS bit should be 1, otherwise the packet will be discarded by WiFi driver. • If the packet is from station to SoftAP or from SoftAP to station, the Power Management, More Data, Protected bits should be 0, other

5.20.16 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.

5.20.17 Wi-Fi Multiple Antennas

The Wi-Fi multiple antennas selecting can be depicted as following picture:

```
                 __________
                |Enabled   |
            ___|Antenna 0 |\\                                                  _________
                |__________| \\        GPIO[0] <----> antenna_select[0] ---|         |␣
↪--- antenna 0
RX/TX ___                     \\____\  GPIO[1] <----> antenna_select[1] ---| Antenna |␣
↪--- antenna 1
         \      __________   //    /  GPIO[2] <----> antenna_select[2] ---| Switch  |␣
↪...  ...
          \ ___|Enabled   | //        GPIO[3] <----> antenna_select[3] ---|_________|␣
↪--- antenna 15
           \   |Antenna 1 |//
                |__________|
```

ESP32 supports up to sixteen antennas through external antenna switch. The antenna switch can be controlled by up to four address pins - antenna_select[0:3]. Different input value of antenna_select[0:3] means selecting different antenna. E.g. the value '0b1011' means the antenna 11 is selected. The default value of antenna_select[3:0] is '0b0000', it means the antenna 0 is selected by default.

Up to four GPIOs are connected to the four active high antenna_select pins. ESP32 can select the antenna by control the GPIO[0:3]. The API `esp_wifi_set_ant_gpio()` is used to configure which GPIOs are connected to antenna_selects. If GPIO[x] is connected to antenna_select[x], then gpio_config->gpio_cfg[x].gpio_select should be set to 1 and gpio_config->gpio_cfg[x].gpio_num should be provided.

Although up to sixteen anteenas are supported, only one or two antennas can be simultaneously enabled for RX/TX. The API `esp_wifi_set_ant()` is used to configure which antennas are enabled.

The enabled antennas selecting algorithm is also configured by `esp_wifi_set_ant()`. The RX/TX antenna mode can be WIFI_ANT_MODE_ANT0, WIFI_ANT_MODE_ANT1 or WIFI_ANT_MODE_AUTO. If the antenna mode is WIFI_ANT_MODE_ANT0, the enabled antenna 0 is selected for RX/TX data. If the antenna mode is WIFI_ANT_MODE_ANT1, the enabled antenna 1 is selected for RX/TX data. Otherwise, WiFi automatically selects the antenna that has better signal from the enabled antennas.

If the RX antenna mode is WIFI_ANT_MODE_AUTO, the default antenna mode also needs to be set. Because the RX antenna switching only happens when some conditions are met, e.g. the RX antenna starts to switch if the RSSI is lower than -65dBm and if another antenna has better signal etc, RX uses the default antenna if the conditions are not met. If the default antenna mode is WIFI_ANT_MODE_ANT1, the enabled antenna 1 is used as the default RX antenna, otherwise the enabled antenna 0 is used as the default RX antenna.

Some limitations need to be considered:

- The TX antenna can be set to WIFI_ANT_MODE_AUTO only if the RX antenna mode is WIFI_ANT_MODE_AUTO because TX antenna selecting algorithm is based on RX antenna in WIFI_ANT_MODE_AUTO type.
- Currently BT doesn't support the multiple antennas feature, please don't use multiple antennas related APIs.

Following is the recommended scenarios to use the multiple antennas:

- In Wi-Fi mode WIFI_MODE_STA, both RX/TX antenna modes are configured to WIFI_ANT_MODE_AUTO. The WiFi driver selects the better RX/TX antenna automatically.
- The RX antenna mode is configured to WIFI_ANT_MODE_AUTO. The TX antenna mode is configured to WIFI_ANT_MODE_ANT0 or WIFI_ANT_MODE_ANT1. The applications can choose to always select a specified antenna for TX, or implement their own TX antenna selecting algorithm, e.g. selecting the TX antenna mode based on the channel switch information etc.
- Both RX/TX antenna modes are configured to WIFI_ANT_MODE_ANT0 or WIFI_ANT_MODE_ANT1.

Wi-Fi Multiple Antennas Configuration

Generally, following steps can be taken to configure the multiple antennas:

- Configure which GPIOs are connected to the antenna_selects, for example, if four antennas are supported and GPIO20/GPIO21 are connected to antenna_select[0]/antenna_select[1], the configurations look like:

```
wifi_ant_gpio_config_t config = {
    { .gpio_select = 1, .gpio_num = 20 },
    { .gpio_select = 1, .gpio_num = 21 }
};
```

- Configure which antennas are enabled and how RX/TX use the enabled antennas, for example, if antenna1 and antenna3 are enabled, the RX needs to select the better antenna automatically and uses antenna1 as its default antenna, the TX always selects the antenna3. The configuration looks like:

```
wifi_ant_config_t config = {
    .rx_ant_mode = WIFI_ANT_MODE_AUTO,
    .rx_ant_default = WIFI_ANT_ANT0,
    .tx_ant_mode = WIFI_ANT_MODE_ANT1,
    .enabled_ant0 = 1,
    .enabled_ant1 = 3
};
```

5.20.18 Wi-Fi Channel State Information

Channel state information (CSI) refers to the channel information of a Wi-Fi connection. In ESP32, this information consists of channel frequency responses of sub-carriers and is estimated when packets are received from the transmitter. Each channel frequency response of sub-carrier is recorded by two bytes of signed characters. The first one is imaginary part and the second one is real part. There are up to three fields of channel frequency responses according to the type of received packet. They are legacy long training field (LLTF), high throughput LTF (HT-LTF) and space time block code HT-LTF (STBC-HT-LTF). For different types of packets which are received on channels with different state, the sub-carrier index and total bytes of signed characters of CSI is shown in the following table.

channel	secondary channel	none			below					above				
packet information	signal mode	non HT	HT		non HT	HT				non HT	HT			
	channel bandwidth	20MHz	20MHz		20MHz	20MHz		40MHz		20MHz	20MHz		40MHz	
	STBC	non STBC	non STBC	STBC	non STBC	non STBC	STBC	non STBC	STBC	non STBC	non STBC	STBC	non STBC	STBC
sub-carrier index	LLTF	0~31,-32~-1	0~31,-32~-1	0~31,-32~-1	0~63	0~63	0~63	0~63	0~63	-64~-1	-64~-1	-64~-1	-64~-1	-64~-1
	HT-LTF	•	0~31,-32~-1	0~31,-32~-1	•	0~63	0~62	0~63,-64~-1	0~60,-60~-1	•	-64~-1	-62~-1	0~63,-64~-1	0~60,-60~-1
	STBC-HT-LTF	•	•	0~31,-32~-1	•	•	0~62	•	0~60,-60~-1	•	•	-62~-1	•	0~60,-60~-1
total bytes		128	256	384	128	256	380	384	612	128	256	376	384	612

All of the information in the table can be found in the structure wifi_csi_info_t.

- Secondary channel refers to secondary_channel field of rx_ctrl field.
- Signal mode of packet refers to sig_mode field of rx_ctrl field.
- Channel bandwidth refers to cwb field of rx_ctrl field.
- STBC refers to stbc field of rx_ctrl field.
- Total bytes refers to len field.
- The CSI data corresponding to each Long Training Field(LTF) type is stored in a buffer starting from the buf field. Each item is stored as two bytes: imaginary part followed by real part. The order of each item is the same as the sub-carrier in the table. The order of LTF is: LLTF, HT-LTF, STBC-HT-LTF. However all 3 LTFs may not be present, depending on the channel and packet information (see above).
- If first_word_invalid field of wifi_csi_info_t is true, it means that the first four bytes of CSI data is invalid due to a hardware limitation in ESP32.
- More information like RSSI, noise floor of RF, receiving time and antenna is in the rx_ctrl field.

Note:

- For STBC packet, CSI is provided for every space-time stream without CSD (cyclic shift delay). As each cyclic shift on the additional chains shall be -200ns, only the CSD angle of first space-time stream is recorded in sub-carrier 0 of HT-LTF and STBC-HT-LTF for there is no channel frequency response in sub-carrier 0. CSD[10:0] is 11 bits, ranging from -pi to pi.
- If LLTF, HT-LTF or STBC-HT-LTF is not enabled by calling API `esp_wifi_set_csi_config()`, the total bytes of CSI data will be fewer than that in the table. For example, if LLTF and HT-LTF is not enabled and STBC-HT-LTF is enabled, when a packet is received with the condition above/HT/40MHz/STBC, the total bytes of CSI data is 244 ((61 + 60) * 2 + 2 = 244, the result is aligned to four bytes and the last two bytes is invalid).

5.20.19 Wi-Fi Channel State Information Configure

To use Wi-Fi CSI, the following steps need to be done.

- Select Wi-Fi CSI in menuconfig. It is "Menuconfig –> Components config –> Wi-Fi –> WiFi CSI(Channel State Information)".
- Set CSI receiving callback function by calling API `esp_wifi_set_csi_rx_cb()`.
- Configure CSI by calling API `esp_wifi_set_csi_config()`.
- Enable CSI by calling API `esp_wifi_set_csi()`.

The CSI receiving callback function runs from Wi-Fi task. So, do not do lengthy operations in the callback function. Instead, post necessary data to a queue and handle it from a lower priority task. Because station does not receive any packet when it is disconnected and only receives packets from AP when it is connected, it is suggested to enable sniffer mode to receive more CSI data by calling `esp_wifi_set_promiscuous()`.

5.20.20 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:

lwip_dynamic_peek_memory = (lwip_udp_con_num * lwip_udp_conn) + (lwip_tcp_con_num * (lwip_tcp_tx_win_size + lwip_tcp_rx_win_size + lwip_tcp_conn))

Some TCP-based applications need only one TCP connection. However, they may choose to close this TCP connection and create a new one when an error (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:

wifi_dynamic_peek_memory = (wifi_rx_dynamic_buf_num * wifi_rx_pkt_size_max) + (wifi_tx_dynamic_buf_num * wifi_tx_pkt_size_max)

Generally, we do not need to care about the dynamic tx long buffers and dynamic tx long long buffers, because they are management frames which only have a small impact on the system.

5.20.21 Wi-Fi Menuconfig

Wi-Fi Buffer Configure

If you are going to modify the default number or type of buffer, it would be helpful to also have an overview of how the buffer is allocated/freed in the data path. The following diagram shows this process in the TX direction:

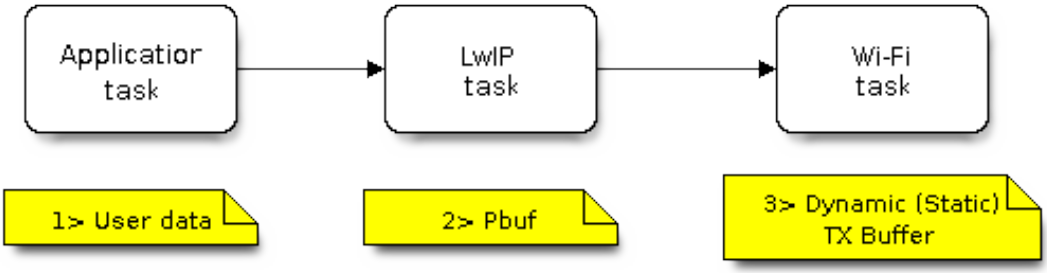


Fig. 31: TX Buffer Allocation

Description:

- The application allocates the data which needs to be sent out.
- The application calls TCPIP-/Socket-related APIs to send the user data. These APIs will allocate a PBUF used in LwIP, and make a copy of the user data.
- When LwIP calls a Wi-Fi API to send the PBUF, the Wi-Fi API will allocate a "Dynamic Tx Buffer" or "Static Tx Buffer", make a copy of the LwIP PBUF, and finally send the data.

The following diagram shows how buffer is allocated/freed in the RX direction:

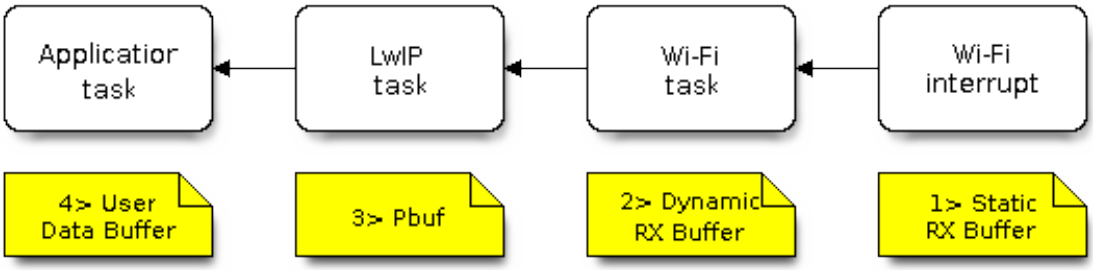


Fig. 32: RX Buffer Allocation

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 Static TX Buffer are mutually exclusive.
Static TX Buffer	Static	32 * 1600Bytes	Yes	This is a kind of

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.

5.21 ESP-MESH

This guide provides information regarding the ESP-MESH protocol. Please see the *MESH API Reference* for more information about API usage.

5.21.1 Overview

ESP-MESH is a networking protocol built atop the Wi-Fi protocol. ESP-MESH allows numerous devices (henceforth referred to as nodes) spread over a large physical area (both indoors and outdoors) to be interconnected under a single WLAN (Wireless Local-Area Network). ESP-MESH is self-organizing and self-healing meaning the network can be built and maintained autonomously.

The ESP-MESH guide is split into the following sections:

1. *Introduction*
2. *ESP-MESH Concepts*
3. *Building a Network*
4. *Managing a Network*
5. *Data Transmission*
6. *Performance*
7. *Further Notes*

5.21.2 Introduction

A traditional infrastructure Wi-Fi network is a point-to-multipoint network where a single central node known as the access point (AP) is directly connected to all other nodes known as stations. The AP is responsible for arbitrating and forwarding transmissions between the stations. Some APs also relay transmissions to/from an external IP network via a router. Traditional infrastructure Wi-Fi networks suffer the disadvantage of limited coverage area due to the requirement that every station must be in range to directly connect with the AP. Furthermore, traditional Wi-Fi networks are susceptible to overloading as the maximum number of stations permitted in the network is limited by the capacity of the AP.

ESP-MESH differs from traditional infrastructure Wi-Fi networks in that nodes are not required to connect to a central node. Instead, nodes are permitted to connect with neighboring nodes. Nodes are mutually responsible for relaying each others transmissions. This allows an ESP-MESH network to have much greater coverage area as nodes can still achieve interconnectivity without needing to be in range of the central node. Likewise, ESP-MESH is also less

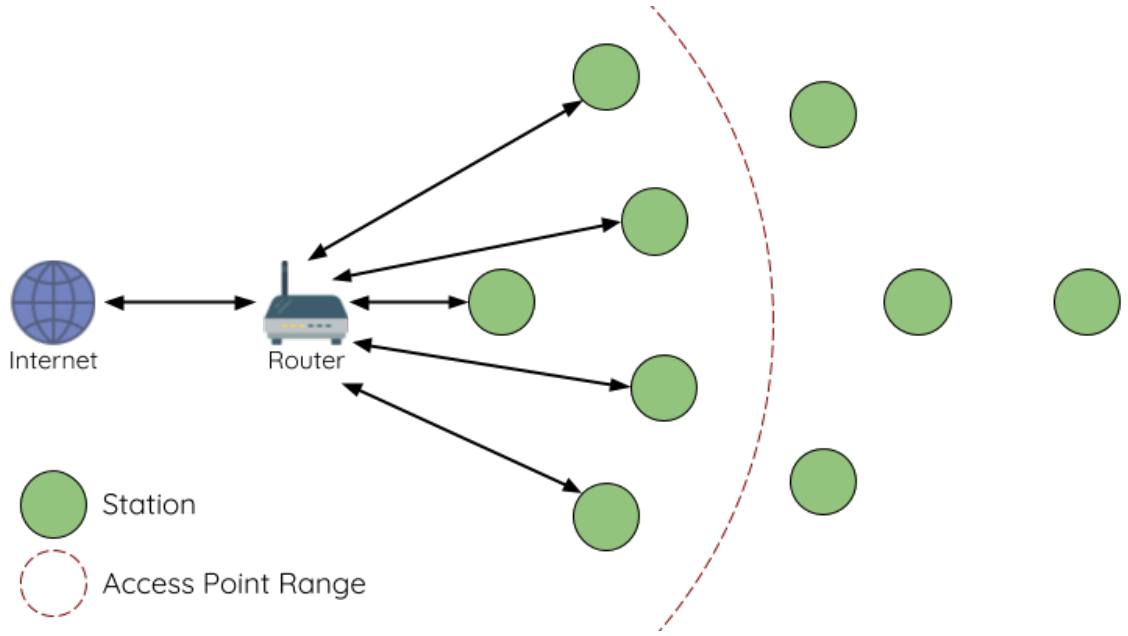


Fig. 33: Traditional Wi-Fi Network Architectures

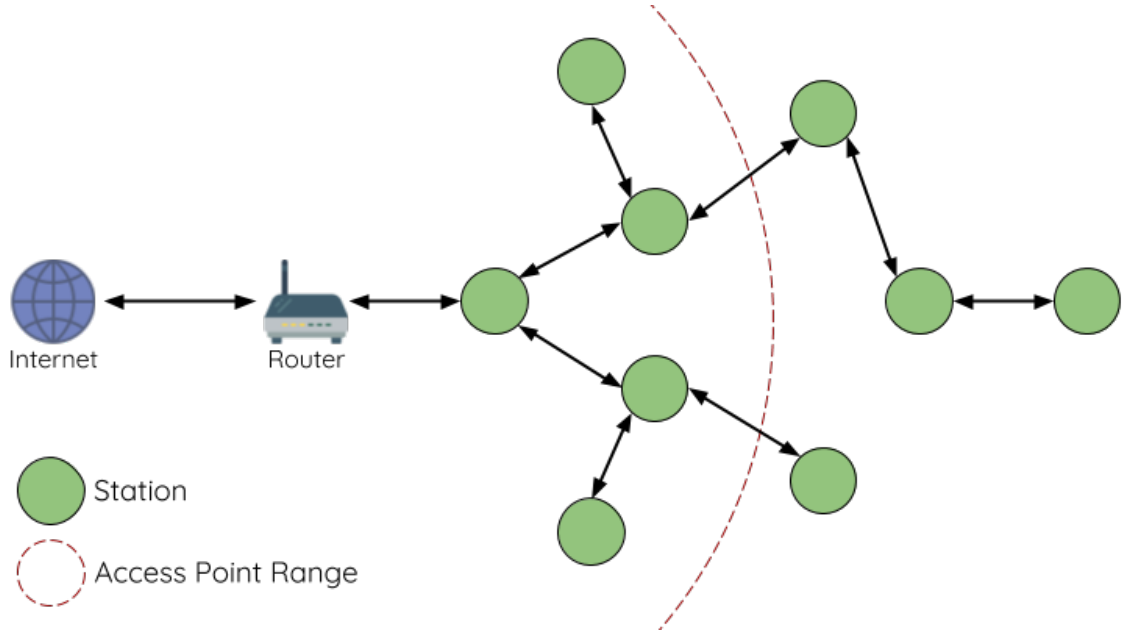


Fig. 34: ESP-MESH Network Architecture

susceptible to overloading as the number of nodes permitted on the network is no longer limited by a single central node.

5.21.3 ESP-MESH Concepts

Terminology

Term	Description
Node	Any device that **is** or **can be** part of an ESP-MESH network
Root Node	The top node in the network
Child Node	A node X is a child node when it is connected to another node Y where the connection makes node X more distant from the root node than node Y (in terms of number of connections).
Parent Node	The converse notion of a child node
Sub-Child Node	Any node reachable by repeated proceeding from parent to child
Sibling Nodes	Nodes that share the same parent node
Connection	A traditional Wi-Fi association between an AP and a station. A node in ESP-MESH will use its station interface to associate with the softAP interface of another node, thus forming a connection. The connection process includes the authentication and association processes in Wi-Fi.
Upstream Connection	The connection from a node to its parent node
Downstream Connection	The connection from a node to one of its child nodes
Wireless Hop	The portion of the path between source and destination nodes that corresponds to a single wireless connection. A data packet that traverses a single connection is known as **single-hop** whereas traversing multiple connections is known as **multi-hop**.
Subnetwork	A subnetwork is subdivision of an ESP-MESH network which consists of a node and all of its descendant nodes. Therefore the subnetwork of the root node consists of all nodes in an ESP-MESH network.
MAC Address	Media Access Control Address used to uniquely identify each node or router within an ESP-MESH network.
DS	Distribution System (External IP Network)

Tree Topology

ESP-MESH is built atop the infrastructure Wi-Fi protocol and can be thought of as a networking protocol that combines many individual Wi-Fi networks into a single WLAN. In Wi-Fi, stations are limited to a single connection with an AP (upstream connection) at any time, whilst an AP can be simultaneously connected to multiple stations (downstream connections). However ESP-MESH allows nodes to simultaneously act as a station and an AP. Therefore a node in ESP-MESH can have **multiple downstream connections using its softAP interface**, whilst simultaneously having **a single upstream connection using its station interface**. This naturally results in a tree network topology with a parent-child hierarchy consisting of multiple layers.

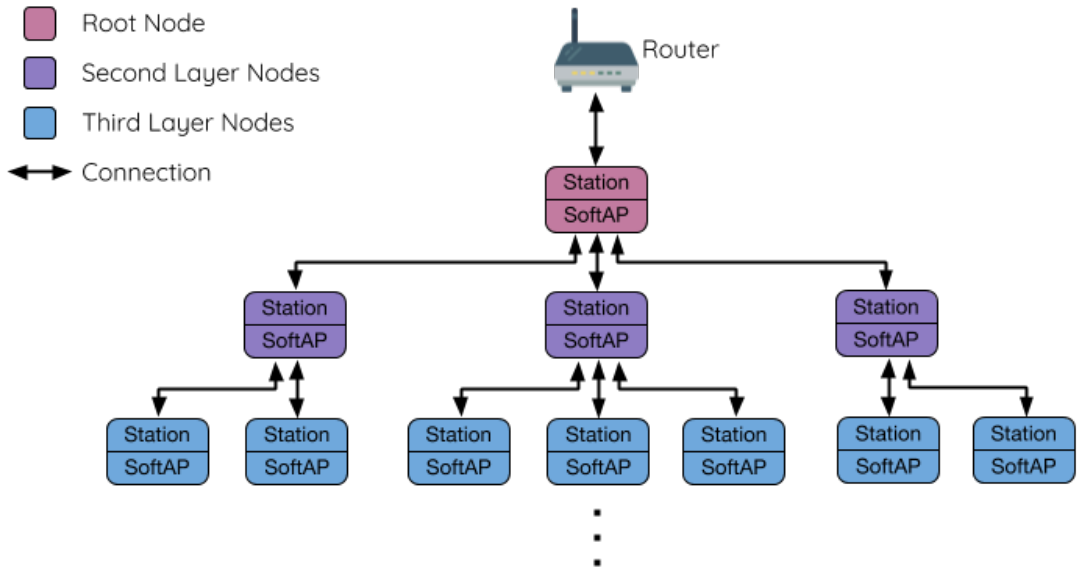


Fig. 35: ESP-MESH Tree Topology

ESP-MESH is a multiple hop (multi-hop) network meaning nodes can transmit packets to other nodes in the network through one or more wireless hops. Therefore, nodes in ESP-MESH not only transmit their own packets, but simultaneously serve as relays for other nodes. Provided that a path exists between any two nodes on the physical layer (via one or more wireless hops), any pair of nodes within an ESP-MESH network can communicate.

Note: The size (total number of nodes) in an ESP-MESH network is dependent on the maximum number of layers permitted in the network, and the maximum number of downstream connections each node can have. Both of these variables can be configured to limit the size of the network.

Node Types

Root Node: The root node is the top node in the network and serves as the only interface between the ESP-MESH network and an external IP network. The root node is connected to a conventional Wi-Fi router and relays packets to/from the external IP network to nodes within the ESP-MESH network. **There can only be one root node within an ESP-MESH network** and the root node's upstream connection may only be with the router. Referring to the diagram above, node A is the root node of the network.

Leaf Nodes: A leaf node is a node that is not permitted to have any child nodes (no downstream connections). Therefore a leaf node can only transmit or receive its own packets, but cannot forward the packets of other nodes. If a node is situated on the network's maximum permitted layer, it will be assigned as a leaf node. This prevents the node from forming any downstream connections thus ensuring the network does not add an extra layer. Some nodes without a softAP interface (station only) will also be assigned as leaf nodes due to the requirement of a softAP interface for any downstream connections. Referring to the diagram above, nodes L/M/N are situated on the networks maximum permitted layer hence have been assigned as leaf nodes .

Intermediate Parent Nodes: Connected nodes that are neither the root node or a leaf node are intermediate parent nodes. An intermediate parent node must have a single upstream connection (a single parent node), but can have zero to multiple downstream connections (zero to multiple child nodes). Therefore an intermediate parent node can transmit and receive packets, but also forward packets sent from its upstream and downstream connections. Referring

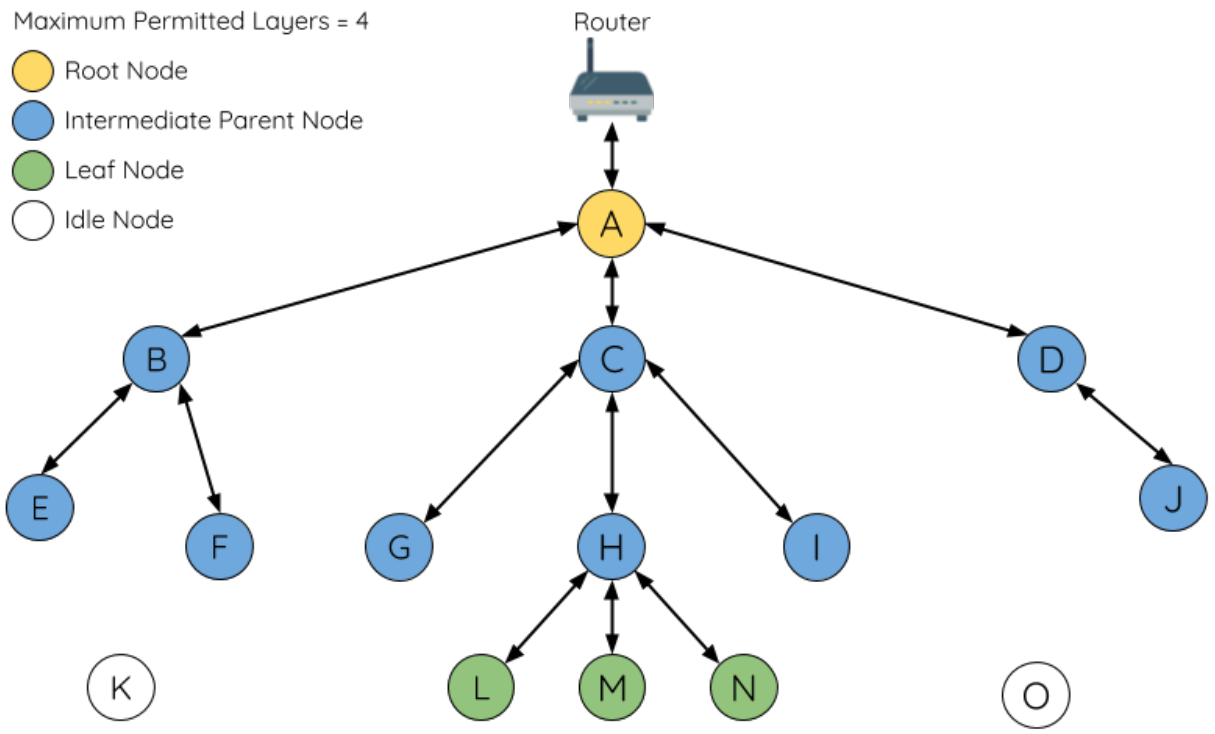


Fig. 36: ESP-MESH Node Types

to the diagram above, nodes B to J are intermediate parent nodes. **Intermediate parent nodes without downstream connections such as nodes E/F/G/I/J are not equivalent to leaf nodes** as they are still permitted to form downstream connections in the future.

Idle Nodes: Nodes that have yet to join the network are assigned as idle nodes. Idle nodes will attempt to form an upstream connection with an intermediate parent node or attempt to become the root node under the correct circumstances (see *Automatic Root Node Selection*). Referring to the diagram above, nodes K and O are idle nodes.

Beacon Frames & RSSI Thresholding

Every node in ESP-MESH that is able to form downstream connections (i.e. has a softAP interface) will periodically transmit Wi-Fi beacon frames. A node uses beacon frames to allow other nodes to detect its presence and know of its status. Idle nodes will listen for beacon frames to generate a list of potential parent nodes, one of which the idle node will form an upstream connection with. ESP-MESH uses the Vendor Information Element to store metadata such as:

- Node Type (Root, Intermediate Parent, Leaf, Idle)
- Current layer of Node
- Maximum number of layers permitted in the network
- Current number of child nodes
- Maximum number of downstream connections to accept

The signal strength of a potential upstream connection is represented by RSSI (Received Signal Strength Indication) of the beacon frames of the potential parent node. To prevent nodes from forming a weak upstream connection, ESP-MESH implements an RSSI threshold mechanism for beacon frames. If a node detects a beacon frame with an RSSI below a preconfigured threshold, the transmitting node will be disregarded when forming an upstream connection.

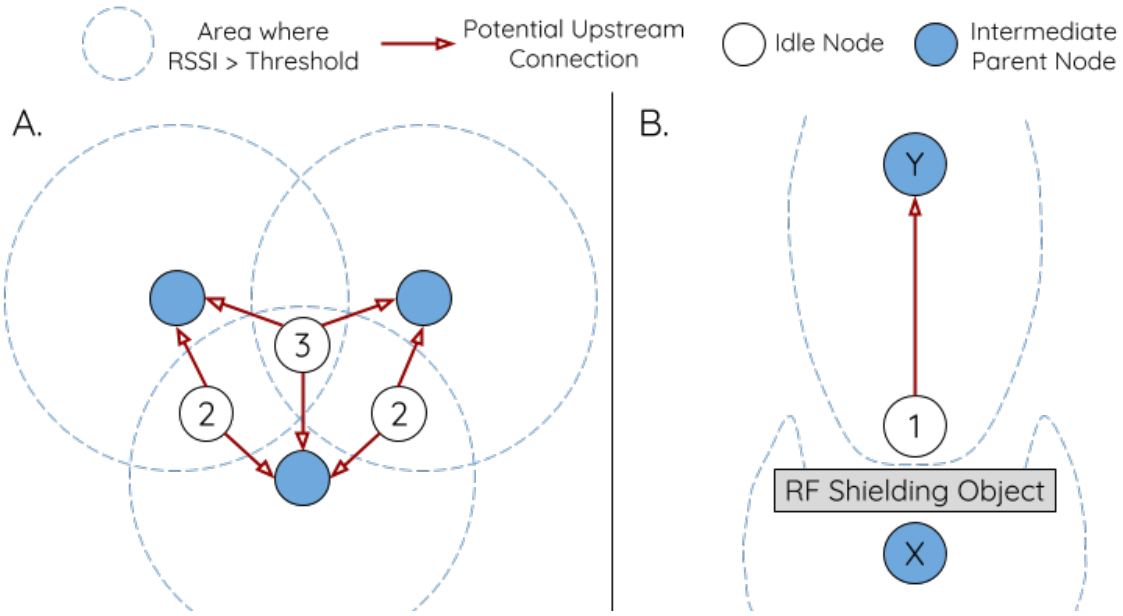


Fig. 37: Effects of RSSI Thresholding

Panel A of the illustration above demonstrates how the RSSI threshold affects the number of parent node candidates an idle node has.

Panel B of the illustration above demonstrates how an RF shielding object can lower the RSSI of a potential parent node. Due to the RF shielding object, the area in which the RSSI of node X is above the threshold is significantly reduced. This causes the idle node to disregard node X even though node X is physically adjacent. The idle node will instead form an upstream connection with the physically distant node Y due to a stronger RSSI.

Note: Nodes technically still receive all beacon frames on the MAC layer. The RSSI threshold is an ESP-MESH feature that simply filters out all received beacon frames that are below the preconfigured threshold.

Preferred Parent Node

When an idle node has multiple parent nodes candidates (potential parent nodes), the idle node will form an upstream connection with the **preferred parent node**. The preferred parent node is determined based on the following criteria:

- Which layer the parent node candidate is situated on
- The number of downstream connections (child nodes) the parent node candidate currently has

The selection of the preferred parent node will always prioritize the parent node candidate on the shallowest layer of the network (including the root node). This helps minimize the total number of layers in an ESP-MESH network when upstream connections are formed. For example, given a second layer node and a third layer node, the second layer node will always be preferred.

If there are multiple parent node candidates within the same layer, the parent node candidate with the least child nodes will be preferred. This criteria has the effect of balancing the number of downstream connections amongst nodes of the same layer.

Panel A of the illustration above demonstrates an example of how the idle node G selects a preferred parent node given the five parent node candidates B/C/D/E/F. Nodes on the shallowest layer are preferred, hence nodes B/C are

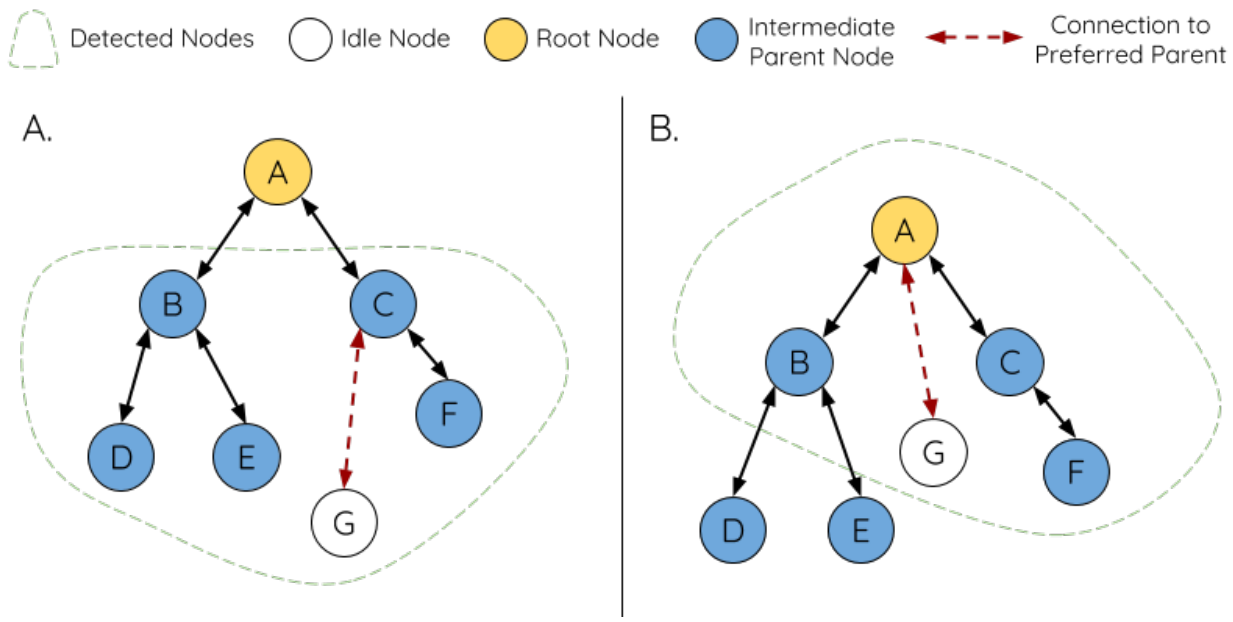


Fig. 38: Preferred Parent Node Selection

prioritized since they are second layer nodes whereas nodes D/E/F are on the third layer. Node C is selected as the preferred parent node due it having fewer downstream connections (fewer child nodes) compared to node B.

Panel B of the illustration above demonstrates the case where the root node is within range of the idle node G. In other words, the root node's beacon frames are above the RSSI threshold when received by node G. The root node is always the shallowest node in an ESP-MESH network hence is always the preferred parent node given multiple parent node candidates.

Note: Users may also define their own algorithm for selecting a preferred parent node, or force a node to only connect with a specific parent node (see the Mesh Manual Networking Example).

Routing Tables

Each node within an ESP-MESH network will maintain its individual routing table used to correctly route ESP-MESH packets (see *ESP-MESH Packet*) to the correct destination node. The routing table of a particular node will **consist of the MAC addresses of all nodes within the particular node's subnetwork** (including the MAC address of the particular node itself). Each routing table is internally partitioned into multiple subtables with each subtable corresponding to the subnetwork of each child node.

Using the diagram above as an example, the routing table of node B would consist of the MAC addresses of nodes B to I (i.e. equivalent to the subnetwork of node B). Node B's routing table is internally partitioned into two subtables containing of nodes C to F and nodes G to I (i.e. equivalent to the subnetworks of nodes C and G respectively).

ESP-MESH utilizes routing tables to determine whether an ESP-MESH packet should be forwarded upstream or downstream based on the following rules.

1. If the packet's destination MAC address is within the current node's routing table and is not the current node, select the subtable that contains the destination MAC address and forward the data packet downstream to the child node corresponding to the subtable.

2. If the destination MAC address is not within the current node's routing table, forward the data packet upstream to

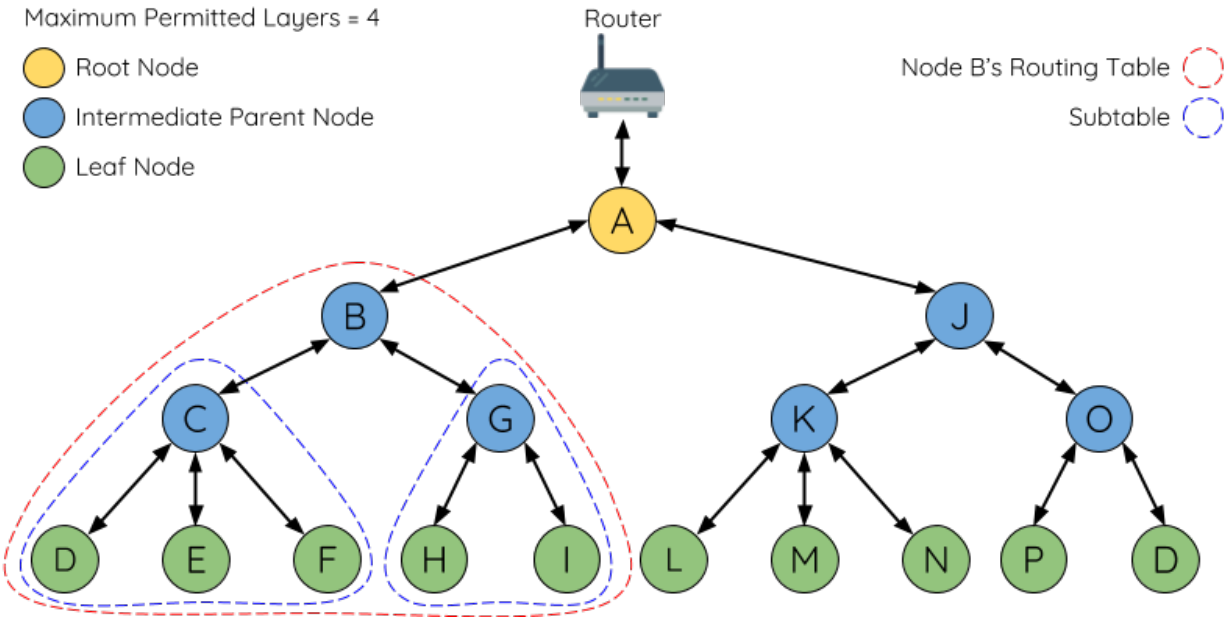


Fig. 39: ESP-MESH Routing Tables Example

the current node's parent node. Doing so repeatedly will result in the packet arriving at the root node where the routing table should contain all nodes within the network.

Note: Users can call `esp_mesh_get_routing_table()` to obtain a node's routing table, or `esp_mesh_get_routing_table_size()` to obtain the size of a node's routing table.

`esp_mesh_get_subnet_nodes_list()` can be used to obtain the corresponding subtable of a specific child node. Likewise, `esp_mesh_get_subnet_nodes_num()` can be used to obtain the size of the subtable.

5.21.4 Building a Network

General Process

Warning: Before the ESP-MESH network building process can begin, certain parts of the configuration must be uniform across each node in the network (see `mesh_cfg_t`). Each node must be configured with **the same Mesh Network ID, router configuration, and softAP configuration**.

An ESP-MESH network building process involves selecting a root node, then forming downstream connections layer by layer until all nodes have joined the network. The exact layout of the network can be dependent on factors such as root node selection, parent node selection, and asynchronous power-on reset. However, the ESP-MESH network building process can be generalized into the following steps:

1. Root Node Selection

The root node can be designated during configuration (see section on *User Designated Root Node*), or dynamically elected based on the signal strength between each node and the router (see *Automatic Root Node Selection*). Once

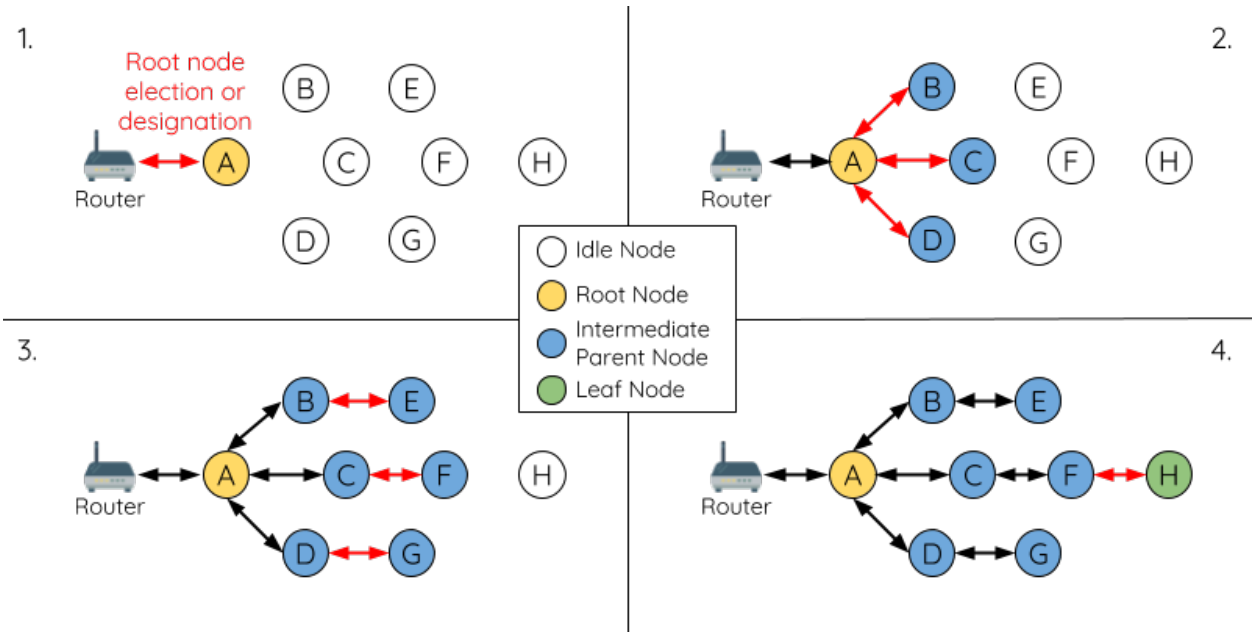


Fig. 40: ESP-MESH Network Building Process

selected, the root node will connect with the router and begin allowing downstream connections to form. Referring to the figure above, node A is selected to be the root node hence node A forms an upstream connection with the router.

2. Second Layer Formation

Once the root node has connected to the router, idle nodes in range of the root node will begin connecting with the root node thereby forming the second layer of the network. Once connected, the second layer nodes become intermediate parent nodes (assuming maximum permitted layers > 2) hence the next layer to form. Referring to the figure above, nodes B to D are in range of the root node. Therefore nodes B to D form upstream connections with the root node and become intermediate parent nodes.

3. Formation of remaining layers

The remaining idle nodes will connect with intermediate parent nodes within range thereby forming a new layer in the network. Once connected, the idles nodes become intermediate parent node or leaf nodes depending on the networks maximum permitted layers. This step is repeated until there are no more idle nodes within the network or until the maximum permitted layer of the network has been reached. Referring to the figure above, nodes E/F/G connect with nodes B/C/D respectively and become intermediate parent nodes themselves.

4. Limiting Tree Depth

To prevent the network from exceeding the maximum permitted number of layers, nodes on the maximum layer will automatically become leaf nodes once connected. This prevents any other idle node from connecting with the leaf node thereby prevent a new layer form forming. However if an idle node has no other potential parent node, it will remain idle indefinitely. Referring to the figure above, the network's maximum permitted layers is set to four. Therefore when node H connects, it becomes a leaf node to prevent any downstream connections from forming.

Automatic Root Node Selection

The automatic selection of a root node involves an election process amongst all idle nodes based on their signal strengths with the router. Each idle node will transmit their MAC addresses and router RSSI values via Wi-Fi beacon frames. **The MAC address is used to uniquely identify each node in the network** whilst the **router RSSI** is used to indicate a node's signal strength with reference to the router.

Each node will then simultaneously scan for the beacon frames from other idle nodes. If a node detects a beacon frame with a stronger router RSSI, the node will begin transmitting the contents of that beacon frame (i.e. voting for the node with the stronger router RSSI). The process of transmission and scanning will repeat for a preconfigured minimum number of iterations (10 iterations by default) and result in the beacon frame with the strongest router RSSI being propagated throughout the network.

After all iterations, each node will individually check for its **vote percentage** (`number of votes/number of nodes participating in election`) to determine if it should become the root node. **If a node has a vote percentage larger than a preconfigured threshold (90% by default), the node will become a root node**.

The following diagram demonstrates how an ESP-MESH network is built when the root node is automatically selected.

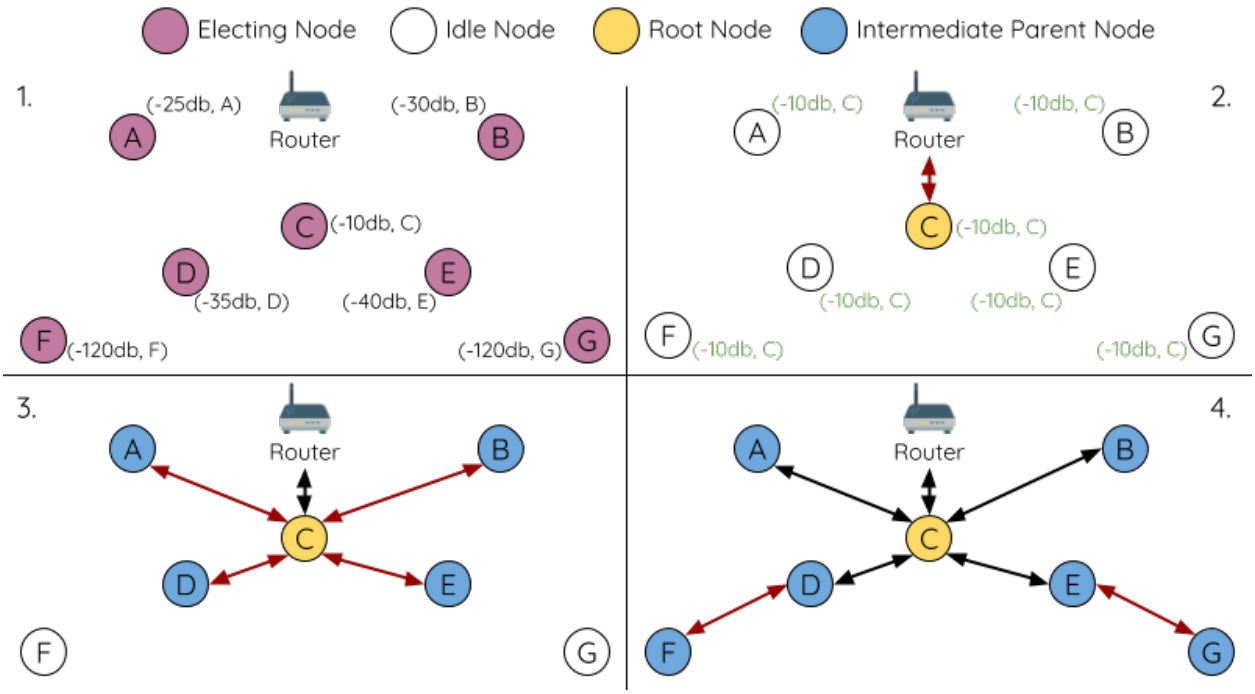


Fig. 41: Root Node Election Example

1. On power-on reset, each node begins transmitting beacon frames consisting of their own MAC addresses and their router RSSIs.

2. Over multiple iterations of transmission and scanning, the beacon frame with the strongest router RSSI is propagated throughout the network. Node C has the strongest router RSSI (-10db) hence its beacon frame is propagated throughout the network. All nodes participating in the election vote for node C thus giving node C a vote percentage of 100%. Therefore node C becomes a root node and connects with the router.

3. Once Node C has connected with the router, nodes A/B/D/E connect with node C as it is the preferred parent node (i.e. the shallowest node). Nodes A/B/D/E form the second layer of the network.

4. Node F and G connect with nodes D and E respectively and the network building process is complete.

Note: The minimum number of iterations for the election process can be configured using `esp_mesh_set_attempts()`. Users should adjust the number of iterations based on the number of nodes within the network (i.e. the larger the network the larger number of scan iterations required).

Warning: **Vote percentage threshold** can also be configured using `esp_mesh_set_vote_percentage()`. Setting a low vote percentage threshold **can result in two or more nodes becoming root nodes** within the same ESP-MESH network leading to the building of multiple networks. If such is the case, ESP-MESH has internal mechanisms to autonomously resolve the **root node conflict**. The networks of the multiple root nodes will be combined into a single network with a single root node. However, root node conflicts where two or more root nodes have the same router SSID but different router BSSID are not handled.

User Designated Root Node

The root node can also be designated by user which will entail the designated root node to directly connect with the router and forgo the election process. When a root node is designated, all other nodes within the network must also forgo the election process to prevent the occurrence of a root node conflict. The following diagram demonstrates how an ESP-MESH network is built when the root node is designated by the user.

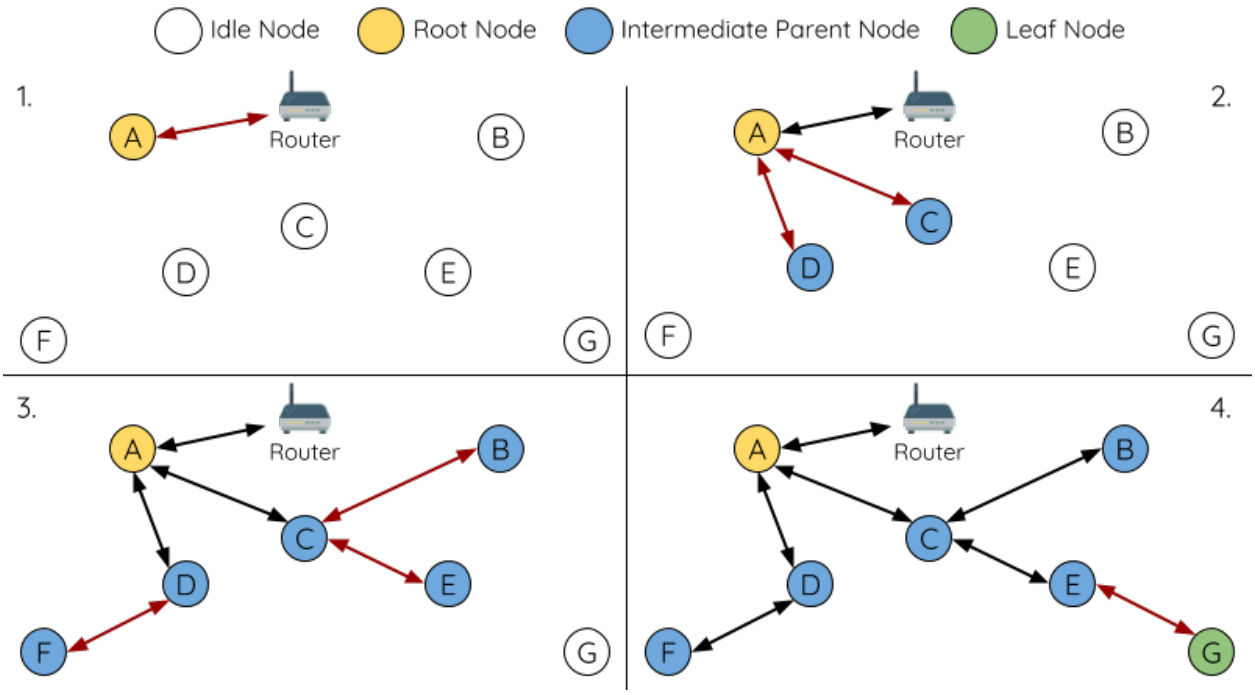


Fig. 42: Root Node Designation Example (Root Node = A, Max Layers = 4)

1. Node A is designated the root node by the user therefore directly connects with the router. All other nodes forgo the election process.

2. Nodes C/D connect with node A as their preferred parent node. Both nodes form the second layer of the network.

3. Likewise, nodes B/E connect with node C, and node F connects with node D. Nodes B/E/F form the third layer of the network.

4. Node G connects with node E, forming the fourth layer of the network. However the maximum permitted number of layers in this network is configured as four, therefore node G becomes a leaf node to prevent any new layers from forming.

Note: When designating a root node, the root node should call `esp_mesh_set_parent()` in order to directly connect with the router. Likewise, all other nodes should call `esp_mesh_fix_root()` to forgo the election process.

Parent Node Selection

By default, ESP-MESH is self organizing meaning that each node will autonomously select which potential parent node to form an upstream connection with. The autonomously selected parent node is known as the preferred parent node. The criteria used for selecting the preferred parent node is designed to reduce the number of layers in the ESP-MESH network and to balance the number of downstream connections between potential parent nodes (see section on *Preferred Parent Node*).

However ESP-MESH also allows users to disable self-organizing behavior which will allow users to define their own criteria for parent node selection, or to configure nodes to have designated parent nodes (see the Mesh Manual Networking Example).

Asynchronous Power-on Reset

ESP-MESH network building can be affected by the order in which nodes power-on. If certain nodes within the network power-on asynchronously (i.e. separated by several minutes), **the final structure of the network could differ from the ideal case where all nodes are powered on synchronously**. Nodes that are delayed in powering on will adhere to the following rules:

Rule 1: If a root node already exists in the network, the delayed node will not attempt to elect a new root node, even if it has a stronger RSSI with the router. The delayed node will instead join the network like any other idle node by connecting with a preferred parent node. If the delayed node is the designated root node, all other nodes in the network will remain idle until the delayed node powers-on.

Rule 2: If a delayed node forms an upstream connection and becomes an intermediate parent node, it may also become the new preferred parent of other nodes (i.e. being a shallower node). This will cause the other nodes to switch their upstream connections to connect with the delayed node (see *Parent Node Switching*).

Rule 3: If an idle node has a designated parent node which is delayed in powering-on, the idle node will not attempt to form any upstream connections in the absence of its designated parent node. The idle node will remain idle indefinitely until its designated parent node powers-on.

The following example demonstrates the effects of asynchronous power-on with regards to network building.

1. Nodes A/C/D/F/G/H are powered-on synchronously and begin the root node election process by broadcasting their MAC addresses and router RSSIs. Node A is elected as the root node as it has the strongest RSSI.

2. Once node A becomes the root node, the remaining nodes begin forming upstream connections layer by layer with their preferred parent nodes. The result is a network with five layers.

3. Node B/E are delayed in powering-on but neither attempt to become the root node even though they have stronger router RSSIs (-20db and -10db) compared to node A. Instead both delayed nodes form upstream connections with their preferred parent nodes A and C respectively. Both Nodes B/E become intermediate parent nodes after connecting.

4. Nodes D/G switch their upstream connections as node B is the new preferred parent node due to it being on a shallower layer (second layer node). Due to the switch, the resultant network has three layers instead of the original five layers.

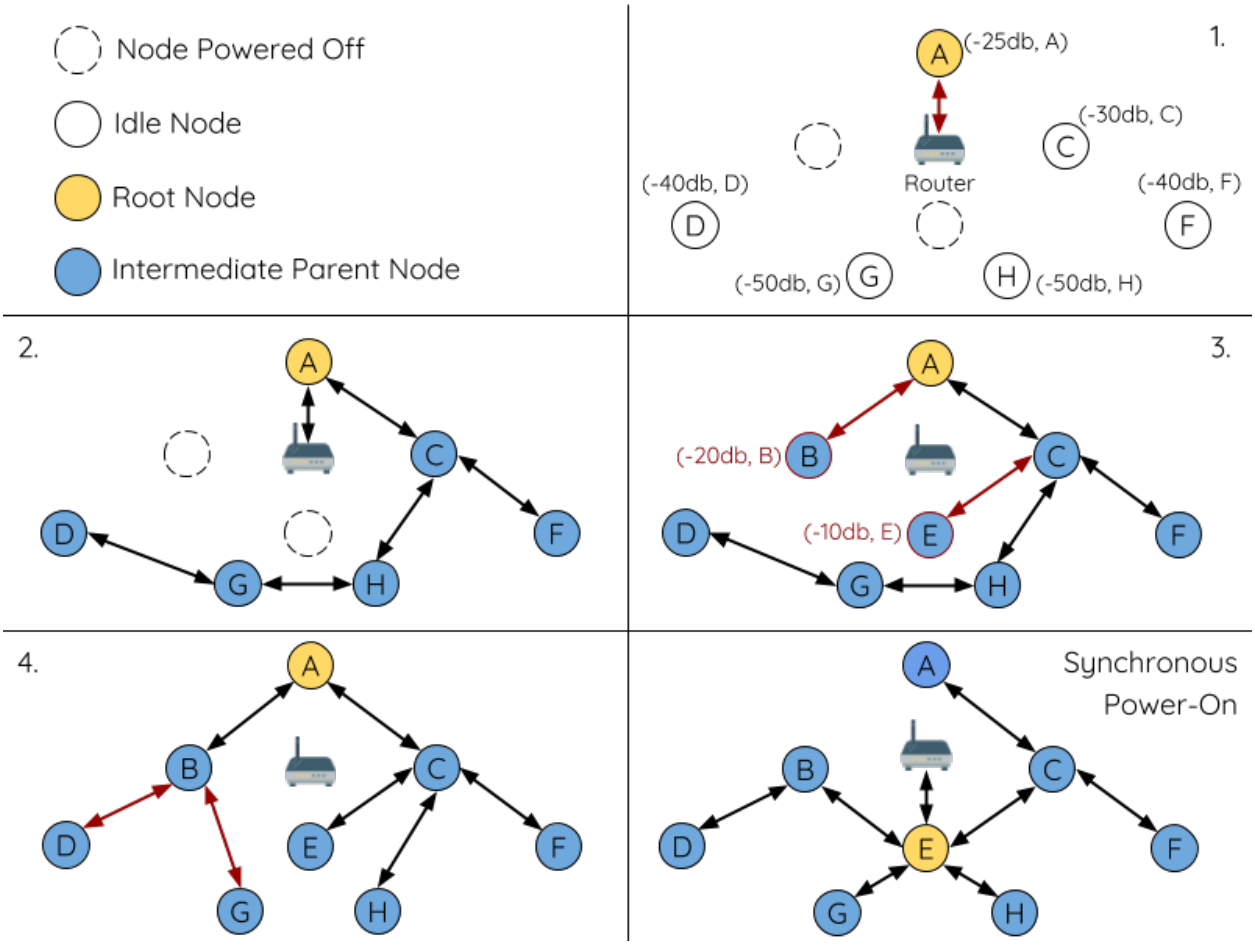


Fig. 43: Network Building with Asynchronous Power On Example

Synchronous Power-On: Had all nodes powered-on synchronously, node E would have become the root node as it has the strongest router RSSI (-10db). This would result in a significantly different network layout compared to the network formed under the conditions of asynchronous power-on. **However the synchronous power-on network layout can still be reached if the user manually switches the root node** (see `esp_mesh_waive_root()`).

Note: Differences in parent node selection caused by asynchronous power-on are autonomously corrected for to some extent in ESP-MESH (see *Parent Node Switching*)

Loop-back Avoidance, Detection, and Handling

A loop-back is the situation where a particular node forms an upstream connection with one of its descendant nodes (a node within the particular node's subnetwork). This results in a circular connection path thereby breaking the tree topology. ESP-MESH prevents loop-back during parent selection by excluding nodes already present in the selecting node's routing table (see *Routing Tables*) thus prevents a particular node from attempting to connect to any node within its subnetwork.

In the event that a loop-back occurs, ESP-MESH utilizes a path verification mechanism and energy transfer mechanism to detect the loop-back occurrence. The parent node of the upstream connection that caused the loop-back will then inform the child node of the loop-back and initiate a disconnection.

5.21.5 Managing a Network

ESP-MESH is a self healing network meaning it can detect and correct for failures in network routing. Failures occur when a parent node with one or more child nodes breaks down, or when the connection between a parent node and its child nodes becomes unstable. Child nodes in ESP-MESH will autonomously select a new parent node and form an upstream connection with it to maintain network interconnectivity. ESP-MESH can handle both Root Node Failures and Intermediate Parent Node Failures.

Root Node Failure

If the root node breaks down, the nodes connected with it (second layer nodes) will promptly detect the failure of the root node. The second layer nodes will initially attempt to reconnect with the root node. However after multiple failed attempts, the second layer nodes will initialize a new round of root node election. **The second layer node with the strongest router RSSI will be elected as the new root node** whilst the remaining second layer nodes will form an upstream connection with the new root node (or a neighboring parent node if not in range).

If the root node and multiple downstream layers simultaneously break down (e.g. root node, second layer, and third layer), the shallowest layer that is still functioning will initialize the root node election. The following example illustrates an example of self healing from a root node break down.

1. Node C is the root node of the network. Nodes A/B/D/E are second layer nodes connected to node C.

2. Node C breaks down. After multiple failed attempts to reconnect, the second layer nodes begin the election process by broadcasting their router RSSIs. Node B has the strongest router RSSI.

3. Node B is elected as the root node and begins accepting downstream connections. The remaining second layer nodes A/D/E form upstream connections with node B thus the network is healed and can continue operating normally.

Note: If a designated root node breaks down, the remaining nodes **will not autonomously attempt to elect a new root node** as an election process will never be attempted whilst a designated root node is used.

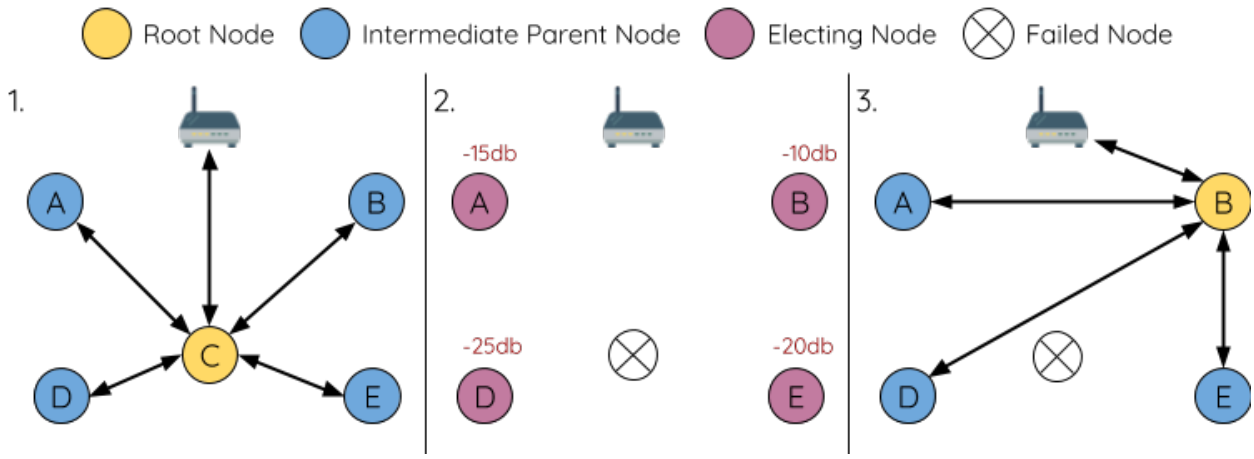


Fig. 44: Self Healing From Root Node Failure

Intermediate Parent Node Failure

If an intermediate parent node breaks down, the disconnected child nodes will initially attempt to reconnect with the parent node. After multiple failed attempts to reconnect, each child node will begin to scan for potential parent nodes (see *Beacon Frames & RSSI Thresholding*).

If other potential parent nodes are available, each child node will individually select a new preferred parent node (see *Preferred Parent Node*) and form an upstream connection with it. If there are no other potential parent nodes for a particular child node, it will remain idle indefinitely.

The following diagram illustrates an example of self healing from an Intermediate Parent Node break down.

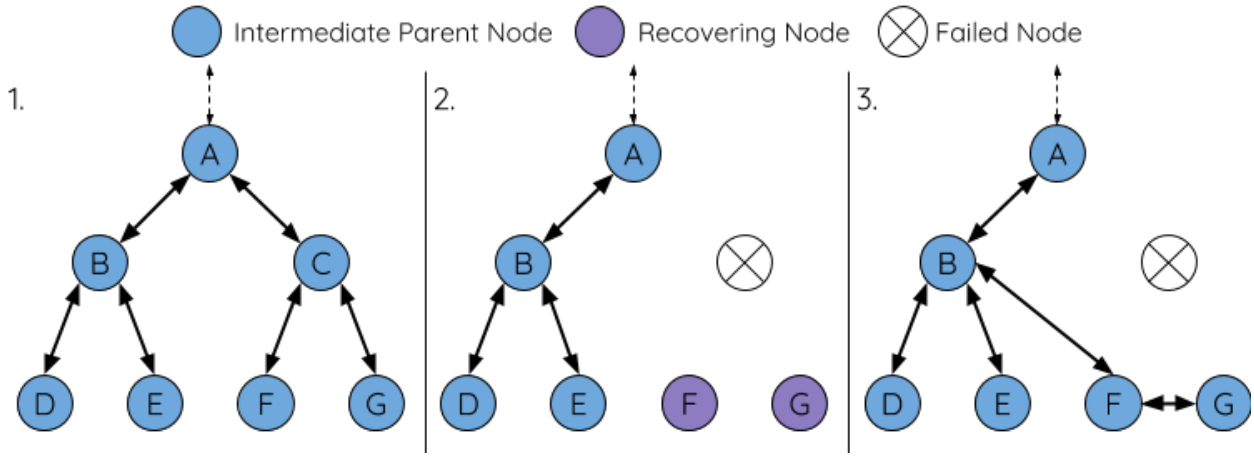


Fig. 45: Self Healing From Intermediate Parent Node Failure

1. The following branch of the network consists of nodes A to G.

2. Node C breaks down. Nodes F/G detect the break down and attempt to reconnect with node C. After multiple failed attempts to reconnect, nodes F/G begin to select a new preferred parent node.

3. Node G is out of range from any other parent node hence remains idle for the time being. Node F is in range of nodes B/E, however node B is selected as it is the shallower node. Node F becomes an intermediate parent node after connecting with Node B thus node G can connect with node F. The network is healed, however the network routing as been affected and an extra layer has been added.

Note: If a child node has a designated parent node that breaks down, the child node will make no attempt to connect with a new parent node. The child node will remain idle indefinitely.

Root Node Switching

ESP-MESH does not automatically switch the root node unless the root node breaks down. Even if the root node's router RSSI degrades to the point of disconnection, the root node will remain unchanged. Root node switching is the act of explicitly starting a new election such that a node with a stronger router RSSI will be elected as the new root node. This can be a useful method of adapting to degrading root node performance.

To trigger a root node switch, the current root node must explicitly call `esp_mesh_waive_root()` to trigger a new election. The current root node will signal all nodes within the network to begin transmitting and scanning for beacon frames (see *Automatic Root Node Selection*) **whilst remaining connected to the network (i.e. not idle)**. If another node receives more votes than the current root node, a root node switch will be initiated. **The root node will remain unchanged otherwise**.

A newly elected root node sends a **switch request** to the current root node which in turn will respond with an acknowledgment signifying both nodes are ready to switch. Once the acknowledgment is received, the newly elected root node will disconnect from its parent and promptly form an upstream connection with the router thereby becoming the new root node of the network. The previous root node will disconnect from the router **whilst maintaining all of its downstream connections** and enter the idle state. The previous root node will then begin scanning for potential parent nodes and selecting a preferred parent.

The following diagram illustrates an example of a root node switch.

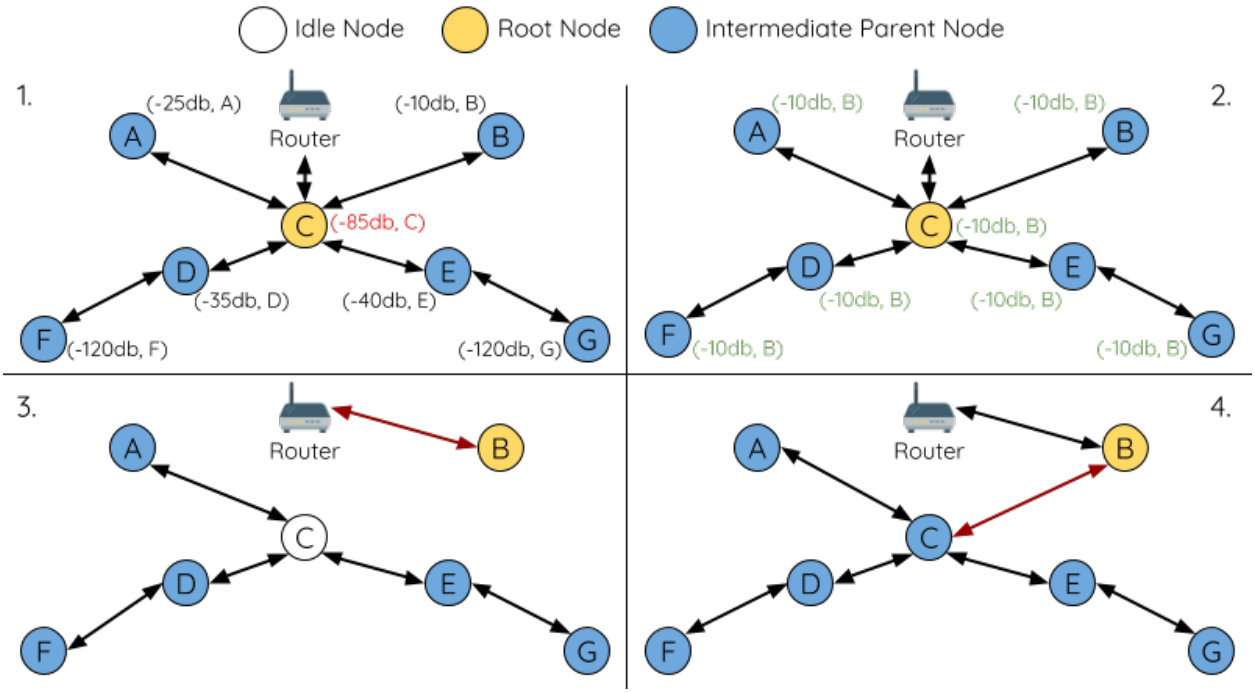


Fig. 46: Root Node Switch Example

1. Node C is the current root node but has degraded signal strength with the router (-85db). The node C triggers a new election and all nodes begin transmitting and scanning for beacon frames **whilst still being connected**.

2. After multiple rounds of transmission and scanning, node B is elected as the new root node. Node B sends node C a **switch request** and node C responds with an acknowledgment.

3. Node B disconnects from its parent and connects with the router becoming the networks new root node. Node C disconnects from the router, enters the idle state, and begins scanning for and selecting a new preferred parent node. **Node C maintains all its downstream connections throughout this process**.

4. Node C selects node B as its preferred parent node, forms an upstream connection, and becomes a second layer node. The network layout is similar after the switch as node C still maintains the same subnetwork. However each node in node C's subnetwork has been placed one layer deeper as a result of the switch. *Parent Node Switching* may adjust the network layout afterwards if any nodes have a new preferred parent node as a result of the root node switch.

Note: Root node switching must require an election hence is only supported when using a self-organized ESP-MESH network. In other words, root node switching cannot occur if a designated root node is used.

Parent Node Switching

Parent Node Switching entails a child node switching its upstream connection to another parent node of a shallower layer. **Parent Node Switching occurs autonomously** meaning that a child node will change its upstream connection automatically if a potential parent node of a shallower layer becomes available (i.e. due to a *Asynchronous Power-on Reset*).

All potential parent nodes periodically transmit beacon frames (see *Beacon Frames & RSSI Thresholding*) allowing for a child node to scan for the availability of a shallower parent node. Due to parent node switching, a self-organized ESP-MESH network can dynamically adjust its network layout to ensure each connection has a good RSSI and that the number of layers in the network is minimized.

5.21.6 Data Transmission

ESP-MESH Packet

ESP-MESH network data transmissions use ESP-MESH packets. ESP-MESH packets are **entirely contained within the frame body of a Wi-Fi data frame**. A multi-hop data transmission in an ESP-MESH network will involve a single ESP-MESH packet being carried over each wireless hop by a different Wi-Fi data frame.

The following diagram shows the structure of an ESP-MESH packet and its relation with a Wi-Fi data frame.

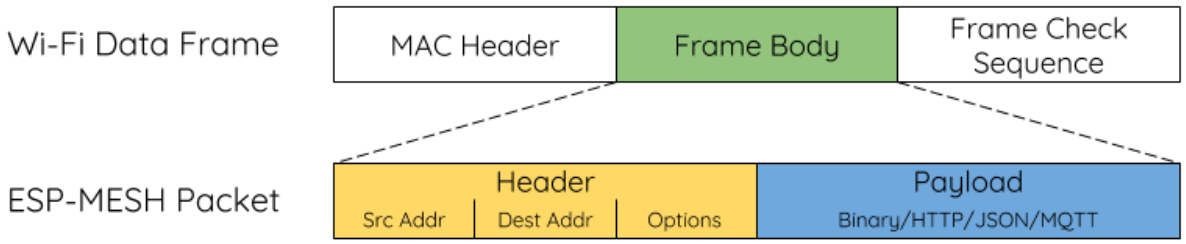


Fig. 47: ESP-MESH Packet

The header of an ESP-MESH packet contains the MAC addresses of the source and destination nodes. The options field contains information pertaining to the special types of ESP-MESH packets such as a

group transmission or a packet originating from the external IP network (see *MESH_OPT_SEND_GROUP* and *MESH_OPT_RECV_DS_ADDR*).

The payload of an ESP-MESH packet contains the actual application data. This data can be raw binary data, or encoded under an application layer protocol such as HTTP, MQTT, and JSON (see *mesh_proto_t*).

Note: When sending an ESP-MESH packet to the external IP network, the destination address field of the header will contain the IP address and port of the target server rather than the MAC address of a node (see *mesh_addr_t*). Furthermore the root node will handle the formation of the outgoing TCP/IP packet.

Group Control & Multicasting

Multicasting is a feature that allows a single ESP-MESH packet to be transmitted simultaneously to multiple nodes within the network. Multicasting in ESP-MESH can be achieved by either specifying a list of target nodes, or specifying a preconfigured group of nodes. Both methods of multicasting are called via *esp_mesh_send()*.

To multicast by specifying a list of target nodes, users must first set the ESP-MESH packet's destination address to the **Multicast-Group Address** (`01:00:5E:xx:xx:xx`). This signifies that the ESP-MESH packet is a multicast packet with a group of addresses, and that the address should be obtained from the header options. Users must then list the MAC addresses of the target nodes as options (see *mesh_opt_t* and *MESH_OPT_SEND_GROUP*). This method of multicasting requires no prior setup but can incur a large amount of overhead data as each target node's MAC address must be listed in the options field of the header.

Multicasting by group allows a ESP-MESH packet to be transmitted to a preconfigured group of nodes. Each grouping is identified by a unique ID, and a node can be placed into a group via *esp_mesh_set_group_id()*. Multicasting to a group involves setting the destination address of the ESP-MESH packet to the target group ID. Furthermore, the *MESH_DATA_GROUP* flag must set. Using groups to multicast incurs less overhead, but requires nodes to previously added into groups.

Note: During a multicast, all nodes within the network still receive the ESP-MESH packet on the MAC layer. However, nodes not included in the MAC address list or the target group will simply filter out the packet.

Broadcasting

Broadcasting is a feature that allows a single ESP-MESH packet to be transmitted simultaneously to all nodes within the network. Each node essentially forwards a broadcast packet to all of its upstream and downstream connections such that the packet propagates throughout the network as quickly as possible. However, ESP-MESH utilizes the following methods to avoid wasting bandwidth during a broadcast.

1. When an intermediate parent node receives a broadcast packet from its parent, it will forward the packet to each of its child nodes whilst storing a copy of the packet for itself.

2. When an intermediate parent node is the source node of the broadcast, it will transmit the broadcast packet upstream to is parent node and downstream to each of its child nodes.

3. When an intermediate parent node receives a broadcast packet from one of its child nodes, it will forward the packet to its parent node and each of its remaining child nodes whilst storing a copy of the packet for itself.

4. When a leaf node is the source node of a broadcast, it will directly transmit the packet to its parent node.

5. When the root node is the source node of a broadcast, the root node will transmit the packet to all of its child nodes.

6. When the root node receives a broadcast packet from one of its child nodes, it will forward the packet to each of its remaining child nodes whilst storing a copy of the packet for itself.

7. When a node receives a broadcast packet with a source address matching its own MAC address, the node will discard the broadcast packet.

8. When an intermediate parent node receives a broadcast packet from its parent node which was originally transmitted from one of its child nodes, it will discard the broadcast packet

Upstream Flow Control

ESP-MESH relies on parent nodes to control the upstream data flow of their immediate child nodes. To prevent a parent node's message buffer from overflowing due to an overload of upstream transmissions, a parent node will allocate a quota for upstream transmissions known as a **receiving window** for each of its child nodes. **Each child node must apply for a receiving window before it is permitted to transmit upstream**. The size of a receiving window can be dynamically adjusted. An upstream transmission from a child node to the parent node consists of the following steps:

1. Before each transmission, the child node sends a window request to its parent node. The window request consists of a sequence number which corresponds to the child node's data packet that is pending transmission.

2. The parent node receives the window request and compares the sequence number with the sequence number of the previous packet sent by the child node. The comparison is used to calculate the size of the receiving window which is transmitted back to the child node.

3. The child node transmits the data packet in accordance with the window size specified by the parent node. If the child node depletes its receiving window, it must obtain another receiving windows by sending a request before it is permitted to continue transmitting.

Note: ESP-MESH does not support any downstream flow control.

Warning: Due to *Parent Node Switching*, packet loss may occur during upstream transmissions.

Due to the fact that the root node acts as the sole interface to an external IP network, it is critical that downstream nodes are aware of the root node's connection status with the external IP network. Failing to do so can lead to nodes attempting to pass data upstream to the root node whilst it is disconnected from the IP network. This results in unnecessary transmissions and packet loss. ESP-MESH address this issue by providing a mechanism to stabilize the throughput of outgoing data based on the connection status between the root node and the external IP network. The root node can broadcast its external IP network connection status to all other nodes by calling `esp_mesh_post_toDS_state()`.

Bi-Directional Data Stream

The following diagram illustrates the various network layers involved in an ESP-MESH Bidirectional Data Stream.

Due to the use of *Routing Tables*, **ESP-MESH is able to handle pack forwarding entirely on the mesh layer**. A TCP/IP layer is only required on the root node when it transmits/receives a packet to/from an external IP network.

5.21.7 Performance

The performance of an ESP-MESH network can be evaluated based on multiple metrics such as the following:

Network Building Time: The amount of time taken to build an ESP-MESH network from scratch.

Healing Time: The amount of time taken for the network to detect a node break down and carry out appropriate actions to heal the network (such as generating a new root node or forming new connections).

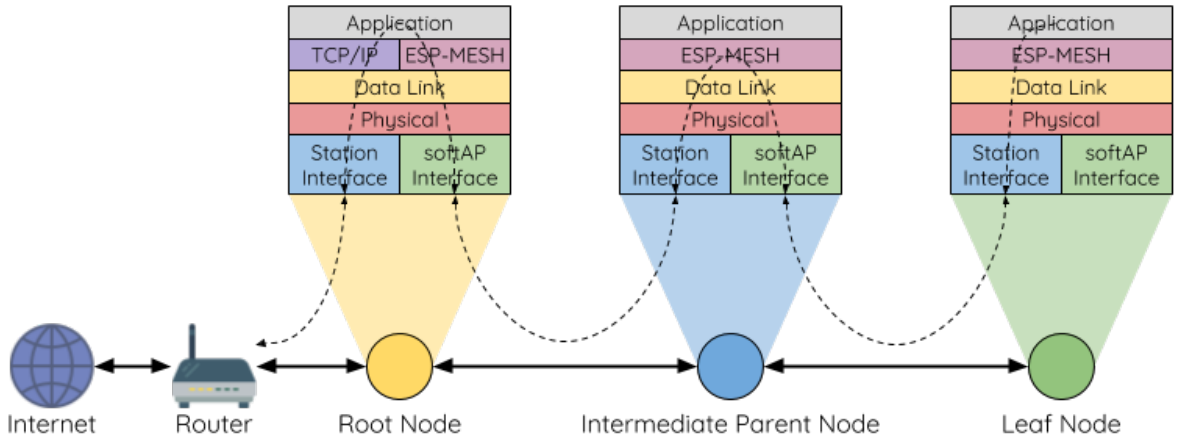


Fig. 48: ESP-MESH Bidirectional Data Stream

Per-hop latency: The latency of data transmission over one wireless hop. In other words, the time taken to transmit a data packet from a parent node to a child node or vice versa.

Network Node Capacity: The total number of nodes the ESP-MESH network can simultaneously support. This number is determined by the maximum number of downstream connections a node can accept and the maximum number of layers permissible in the network.

The following table lists the common performance figures of an ESP-MESH network. However users should note that performance numbers can vary greatly between installations based on network configuration and operating environment.

Function	Description
Networking Building Time	< 60 seconds
Healing time	Root Node Break Down: < 10 seconds Child Node Break Down: < 5 seconds
Per-hop latency	10 to 30 milliseconds

Note: The following test conditions were used to generate the performance figures above.

- Number of test devices: **100**
- Maximum Downstream Connections to Accept: **6**
- Maximum Permissible Layers: **6**

Note: Throughput depends on packet error rate and hop count.

Note: The throughput of root node's access to the external IP network is directly affected by the number of nodes in the ESP-MESH network and the bandwidth of the router.

5.21.8 Further Notes

- Data transmission uses Wi-Fi WPA2-PSK encryption
- Mesh networking IE uses AES encryption

Router and internet icon made by Smashicons from www.flaticon.com

5.22 BluFi

5.22.1 Overview

The BluFi for ESP32 is a Wi-Fi network configuration function via Bluetooth channel. It provides a secure protocol to pass Wi-Fi configuration and credentials to the ESP32. Using this information ESP32 can then e.g. connect to an AP or establish a SoftAP.

Fragmenting, data encryption, checksum verification in the BluFi layer are the key elements of this process.

You can customize symmetric encryption, asymmetric encryption and checksum support customization. Here we use the DH algorithm for key negotiation, 128-AES algorithm for data encryption, and CRC16 algorithm for checksum verification.

5.22.2 The BluFi Flow

The BluFi networking flow includes the configuration of the SoftAP and Station.

The following uses Station as an example to illustrate the core parts of the procedure, including broadcast, connection, service discovery, negotiation of the shared key, data transmission, connection status backhaul.

1. Set the ESP32 into GATT Server mode and then it will send broadcasts with specific *advertising data*. You can customize this broadcast as needed, which is not a part of the BluFi Profile.
2. Use the App installed on the mobile phone to search for this particular broadcast. The mobile phone will connect to ESP32 as the GATT Client once the broadcast is confirmed. The App used during this part is up to you.
3. After the GATT connection is successfully established, the mobile phone will send a data frame for key negotiation to ESP32 (see the section *The Frame Formats Defined in BluFi* for details).
4. After ESP32 receives the data frame of key negotiation, it will parse the content according to the user-defined negotiation method.
5. The mobile phone works with ESP32 for key negotiation using the encryption algorithms such as DH, RSA or ECC.
6. After the negotiation process is completed, the mobile phone will send a control frame for security-mode setup to ESP32.
7. When receiving this control frame, ESP32 will be able to encrypt and decrypt the communication data using the shared key and the security configuration.
8. The mobile phone sends the data frame defined in the section of *The Frame Formats Defined in BluFi*with the Wi-Fi configuration information to ESP32, including SSID, password, etc.
9. The mobile phone sends a control frame of Wi-Fi connection request to ESP32. When receiving this control frame, ESP32 will regard the communication of essential information as done and get ready to connect to the Wi-Fi.
10. After connecting to the Wi-Fi, ESP32 will send a control frame of Wi-Fi connection status report to the mobile phoneto report the connection status. At this point the networking procedure is completed.

Note:

1. After ESP32 receives the control frame of security-mode configuration, it will execute the operations in accordance with the defined security mode.
2. The data lengths before and after symmetric encryption/decryption must stay the same. It also supports in-place encryption and decryption.

5.22.3 The flow chat of BluFi

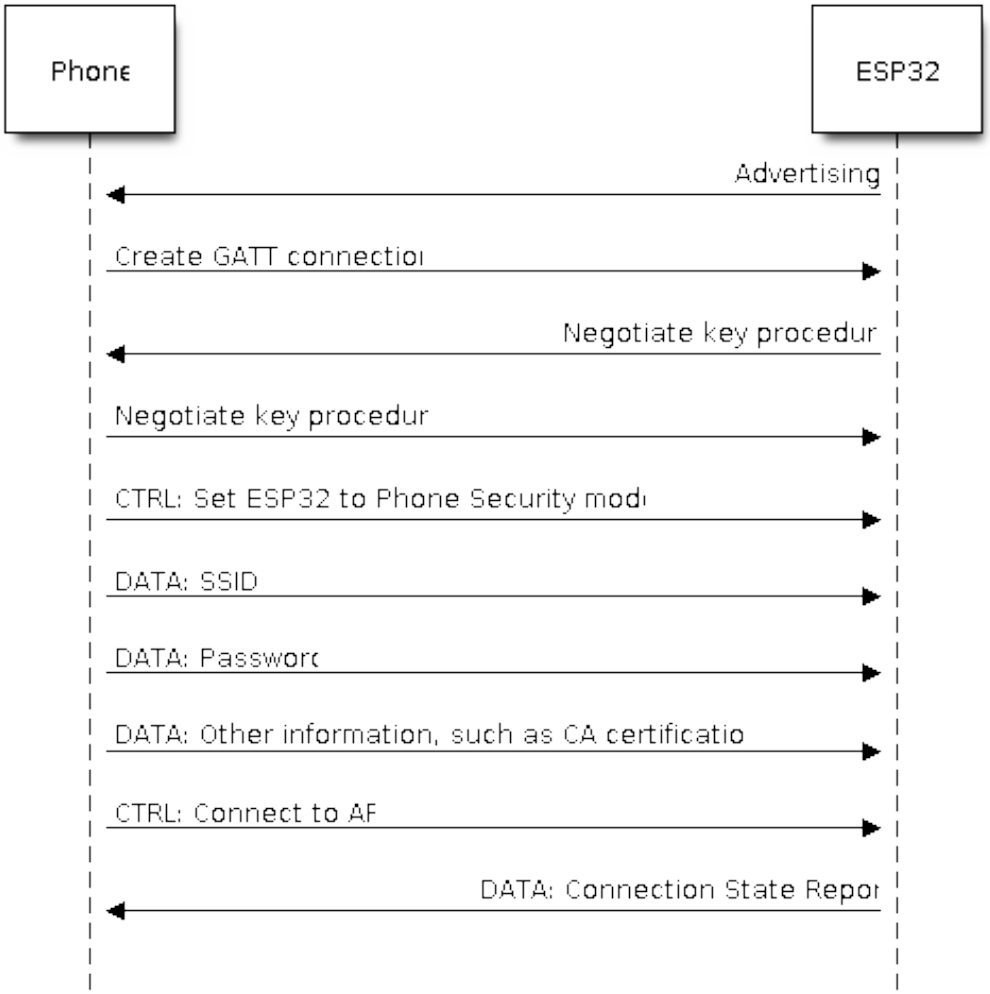


Fig. 49: BluFi Flow Chart

5.22.4 The Frame Formats Defined in BluFi

The frame formats for the communication between the mobile phone App and ESP32 are defined as follows:

The frame format with no fragment (8 bit)

LSB - Type	Frame Control	Sequence Number	Data Length	Data	MSB - CheckSum
1	1	1	1	${Data Length}	2

If the **Frame Ctrl** bit is enabled, the **Total length** bit indicates the length of remaining part of the frame. It can tell the remote how much memory needs to be alloced.

The frame format with fragments8 bit

LSB - Type	FrameControl(Frag)	SequenceNumber	DataLength	Data		MSB - CheckSum
				Total Content Length	Content	
1	1	1	1	2	${Data Length} - 2	2

Normally, the control frame does not contain data bits, except for Ack Frame.

The format of Ack Frame8 bit

LSB - Type (Ack)	Frame Control	SequenceNumber	Data Length	Data	MSB - CheckSum
				Acked Sequence Number	
1	1	1	1	1	2

1. Type

   The **Type** field, taking 1 byte, is divided into **Type** and **Subtype**, that Type uses the lower 2 bits and **Subtype** uses the upper 6 bits.

   - The control frame is not encrypted for the time being and supports to be verified;
   - The data frame supports to be encrypted and verified.

   1.1 Control Frame (0x0b'00)

Control Frame / 0x0b'00	Implication	Explanation	Note
0x0b'00000	Ack	The data field of the Ack frame uses the same sequence value of the frame to reply to.	The data field consumes a byte and its value is the same as the sequence field of the frame to reply to.
0x1b'00001	Set ESP32 to the security mode.	To inform ESP32 of the security mode to use when sending data, which is allowed to be reset multiple times during the process. Each setting affects the subsequent security mode used. If it is not set, ESP32 will send the control frame and data frame with no checksum and encryption by default. The data transmission from the mobile phone to ESP32 is controlled by this control frame.	The data field consumes a byte. The higher 4 bits are for the security mode setting of the control frame, and the lower 4 bits are for the security mode setting of the data frame.
			b'0000: no checksum and no encryption;
			b'0001: with checksum but no encryption;
			b'0010: no checksum but with encryption;
			b'0011: with both checksum and encryption.
0x2b'00010	Set the opmode of Wi-Fi.	The frame contains opmode settings for configuring for the Wi-Fi mode of ESP32.	data[0] is for opmode settings, including:
			0x00: NULL
			0x01: STA;
			0x02: SoftAP;
			0x03: SoftAP&STA.
			Please set the SSID/Password/Max Connection Number of the AP mode in the first place if an AP gets involved.
0x3b'00011	Connect ESP32 to the AP.	To notify ESP32 that the essential information has been sent and it is allowed to connect to the AP.	No data field is contained.
0x4b'00100	Disconnect ESP32 from the AP.		No data field is contained.
0x5b'00101	To get the information of ESP32's Wi-Fi mode and its status.		No data field is contained. When receiving this control frame, ESP32 will send back a follow-up frame of Wi-Fi connection state report to the mobile phone with the information of the current opmode, connection status, SSID and so on. The types of information sent to the mobile phone is defined by the application installed on the phone.

1.2 Data Frame (0x1b'01)

Data Frame	Implication	Explanation
0x0b'000000	Send the negotiation data.	The negotiation data will be sent to the callback function
0x1b'000001	Send the BSSID for STA mode.	To send the BSSID of the AP for the STA device to conn
0x2b'000010	Send the SSID for STA mode.	To send the SSID of the AP for the STA device to connec
0x3b'000011	Send the password for STA mode.	To send the password of the AP for the STA device to co
0x4b'000100	Send the SSID for SoftAP mode.	
0x5b'000101	Send the password for SoftAPmode.	
0x6b'000110	Set the maximum connection number for SoftAP mode.	
0x7b'000111	Set the authentication mode for the SoftAP.	
0x8b'001000	Set the channel amount for SoftAP mode.	
0x9b'001001	Username	It provides the username of the GATT client when using
0xab'001010	CA Certification	It provides the CA Certification when using encryption o
0xbb'001011	Client Certification	It provides the client certification when using encryption
0xcb'001100	Server Certification	It provides the sever certification when using encryption
0xdb'001101	ClientPrivate Key	It provides the private key of the client when using encry
0xeb'001110	ServerPrivate Key	It provides the private key of the sever when using encryp
0xfb'001111	Wi-Fi Connection State Report	To notify the phone of the ESP32's Wi-Fi status, including STA status and SoftAP status. It is for the STA device to connect to the mobile phone or the SoftAP. However, when the mobile phone receives the Wi-Fi status, it can reply to other frames in addition to this frame.
0x10b'010000	Version	
0x11B'010001	Wi-Fi List	To send the Wi-Fi list to ESP32.
0x12B'010010	Report Error	To notify the mobile phone that there is an error with BluFi.
0x13B'010011	Custom Data	To send or receive custom data.

2. Frame Control

   Control field, takes 1 byte and each bit has a different meaning.

Bit	Meaning
0x01	Indicates whether the frame is encrypted.
	1 means encryption, and 0 means unencrypted.
	The encrypted part of the frame includes the full clear data before the DATA field is encrypted (no checksum).
	Control frame is not encrypted, so this bit is 0.
0x02	The data field that indicates whether a frame contains a checksum (such as SHA1,MD5,CRC, etc.) for the end of the frame data field includes SEQUCNE + data length + clear text. Both the control frame and the data frame can contain a check bit or not.
0x04	Represents the data direction.
	0 means the mobile phone to ESP32;
	1 means ESP32 to the mobile phone.
0x08	Indicates whether the other person is required to reply to an ACK.
	0 indicates no requirement;
	1 indicates to reply Ack.
0x10	Indicates whether there are subsequent data fragments.
	0 indicates that there are no subsequent data fragments for this frame;
	1 indicates that there are subsequent data fragments and used to transmit longer data.
	In the case of a frag frame, the total length of the current content section + subsequent content section is given, in the first 2 bytes of the data field (that is, the content data of the maximum support 64K).
0x10~0x80 re-served	

3. Sequence Control

   Sequence control field. When a frame is sent,the value of sequence fied is automatically incremented by 1 regardless of the type of frame, which prevents Replay Attack. The sequence is cleared after each reconnection.

4. Length

   The length of the data field that does not include CheckSum.

5. Data

   The instruction of the data field is different according to various values of Type or Subtype. Please refer to the table above.

6. CheckSum

   This field takes 2 bytes that is used to check "sequence + data length + clear text data".

5.22.5 The Security Implementation of ESP32

1. Securing data

   To ensure that the transmission of the Wi-Fi SSID and password is secure, the message needs to be encrypted using symmetric encryption algorithms, such as AES, DES and so on. Before using symmetric encryption algorithms, the devices are required to negotiate (or generate) a shared key using an asymmetric encryption algorithm (DH, RSA, ECC, etc).

2. Ensuring data integrity

   To ensure data integrity, you need to add a checksum algorithm, such as SHA1, MD5, CRC, etc.

3. Securing identity (signature)

   Algorithm like RSA can be used to secure identity. But for DH, it needs other algorithms as an companion for signature.

4. Replay attack prevention

   It is added to the Sequence field and used during the checksum verification.

   For the coding of ESP32, you can determine and develop the security processing, such as key negotiation. The mobile application sends the negotiation data to ESP32 and then the data will be sent to the application layer for processing. If the application layer does not process it, you can use the DH encryption algorithm provided by BluFi to negotiate the key.

   The application layer needs to register several security-related functions to BluFi:

```
typedef void (*esp_blufi_negotiate_data_handler_t)(uint8_t *data, int len, uint8_t␣
↪**output_data, int *output_len, bool *need_free)
```

This function is for ESP32 to receive normal data during negotiation, and after processing is completed, the data will be transmitted using Output_data and Output_len.

BluFi will send output_data from Negotiate_data_handler after Negotiate_data_handler is called.

Here are two "*", because the length of the data to be emitted is unknown that requires the function to allocate itself (malloc) or point to the global variable, and to inform whether the memory needs to be freed by NEED_FREE.

```
typedef int (* esp_blufi_encrypt_func_t)(uint8_t iv8, uint8_t *crypt_data, int cyprt_
↪len)
```

The data to be encrypted and decrypted must use the same length. The IV8 is a 8 bit sequence value of frames, which can be used as a 8 bit of IV.

```
typedef int (* esp_blufi_decrypt_func_t)(uint8_t iv8, uint8_t *crypt_data, int crypt_
↪len)
```

The data to be encrypted and decrypted must use the same length. The IV8 is a 8 bit sequence value of frames, which can be used as a 8 bit of IV.

```
typedef uint16_t (*esp_blufi_checksum_func_t)(uint8_t iv8, uint8_t *data, int len)
```

This function is used to compute CheckSum and return a value of CheckSum. BluFi uses the returned value to compare the CheckSum of the frame.

5.22.6 GATT Related Instructions

UUID

BluFi Service UUID: 0xFFFF16 bit

BluFi (the mobile -> ESP32): 0xFF01, writable

Blufi (ESP32 -> the mobile phone): 0xFF02, readable and callable

Note:

1. The Ack mechanism is already defined in the profile, but there is no implementation based on the code for the time being.
2. Other parts have been implemented.

5.23 Support for external RAM

5.23.1 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.

5.23.2 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.

5.23.3 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 *`CONFIG_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.

5.23.4 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 *CONFIG_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.

CHAPTER 6

Contributions Guide

We welcome contributions to the esp-idf project!

6.1 How to Contribute

Contributions to esp-idf - fixing bugs, adding features, adding documentation - are welcome. We accept contributions via Github Pull Requests.

6.2 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.

6.3 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.

6.4 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.

6.5 Related Documents

6.5.1 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
```

CMake Code Style

- Indent with four spaces.
- Maximum line length 120 characters. When splitting lines, try to focus on readability where possible (for example, by pairing up keyword/argument pairs on individual lines).
- Don't put anything in the optional parentheses after `endforeach()`, `endif()`, etc.
- Use lowercase (`with_underscores`) for command, function, and macro names.
- For locally scoped variables, use lowercase (`with_underscores`).
- For globally scoped variables, use uppercase (`WITH_UNDERSCORES`).
- Otherwise follow the defaults of the cmake-lint project.

Configuring the code style for a project using EditorConfig

EditorConfig helps developers define and maintain consistent coding styles between different editors and IDEs. The EditorConfig project consists of a file format for defining coding styles and a collection of text editor plugins that enable editors to read the file format and adhere to defined styles. EditorConfig files are easily readable and they work nicely with version control systems.

For more information, see EditorConfig Website.

Documenting code

Please see the guide here: *Documenting Code*.

Structure and naming

Language features

To be written.

6.5.2 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.

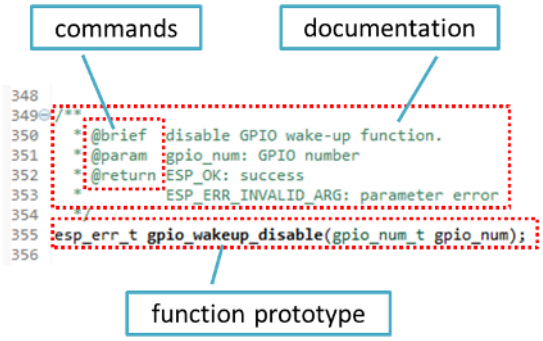
Introduction

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.

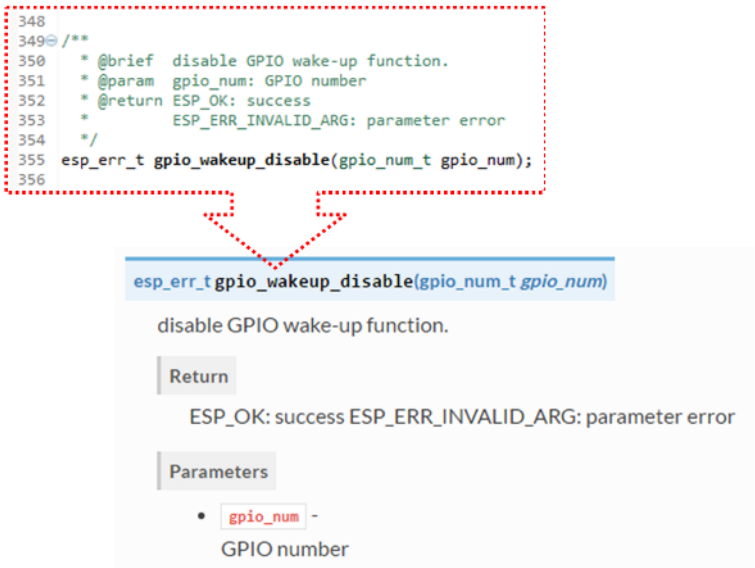


Doxygen supports couple of formatting styles. It also gives you great flexibility on level of details to include in documentation. To get familiar with available features, please check data reach and very well organized Doxygen Manual.

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.

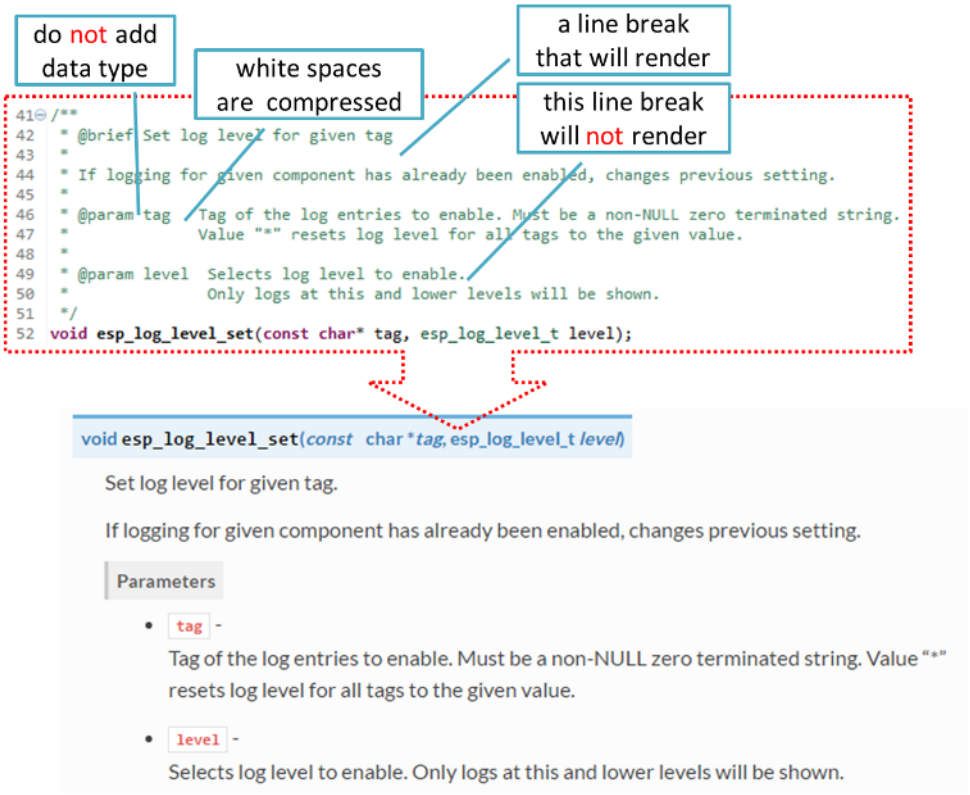
With these tools the above piece of code renders like below:



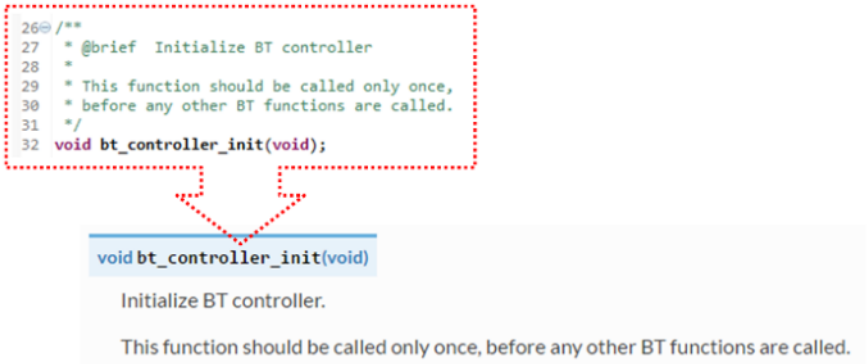
Go for it!

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.



4. If function has void input or does not return any value, then skip `@param` or `@return`



5. When documenting a `define` as well as members of a `struct` or `enum`, place specific comment like below after each member.

6. 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
*
```

7. Overview of functionality of documented header file, or group of files that make a library, should be placed in the same directory in a separate `README.rst` file. If directory contains header files for different APIs, then the file name should be `apiname-readme.rst`.

Go one extra mile

There is couple of tips, how you can make your documentation even better and more useful to the reader.

1. Add code snippets to illustrate implementation. To do so, enclose snippet using `@code{c}` and `@endcode` commands.

```
*
* @code{c}
* // Example of using nvs_get_i32:
* int32_t max_buffer_size = 4096; // default value
* esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_buffer_size);
* assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
* // if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
* // have its default value.
* @endcode
*
```

   The code snippet should be enclosed in a comment block of the function that it illustrates.

2. To highlight some important information use command `@attention` or `@note`.

```
*
* @attention
*     1. This API only impact WIFI_MODE_STA or WIFI_MODE_APSTA mode
*     2. If the ESP32 is connected to an AP, call esp_wifi_disconnect to␣
↪disconnect.
*
```

   Above example also shows how to use a numbered list.

3. To provide common description to a group of similar functions, enclose them using `/**@{*/` and `/**@}*/` markup commands:

```
/**@{*/
/**
 * @brief common description of similar functions
 *
 */
void  first_similar_function (void);
void second_similar_function (void);
/**@}*/
```

   For practical example see nvs_flash/include/nvs.h.

4. You may want to go even further and skip some code like e.g. repetitive defines or enumerations. In such case enclose the code within `/** @cond */` and `/** @endcond */` commands. Example of such implementation is provided in driver/include/driver/gpio.h.

5. Use markdown to make your documentation even more readable. You will add headers, links, tables and more.

```
*
* [ESP32 Technical Reference](https://espressif.com/sites/default/files/
↪documentation/esp32_technical_reference_manual_en.pdf)
*
```

Note: Code snippets, notes, links, etc. will not make it to the documentation, if not enclosed in a comment block associated with one of documented objects.

6. Prepare one or more complete code examples together with description. Place description in a separate file `README.md` in specific folder of examples directory.

Linking Examples

When linking to examples on GitHub do not use absolute / hadcoded URLs. Instead, use docutils custom roles that will generate links for you. These auto-generated links point to the tree or blob for the git commit ID (or tag) of the repository. This is needed to ensure that links do not get broken when files in master branch are moved around or deleted.

The following roles are provided:

- `:idf:`path`` - points to directory inside ESP-IDF
- `:idf_file:`path`` - points to file inside ESP-IDF
- `:idf_raw:`path`` - points to raw view of the file inside ESP-IDF
- `:component:`path`` - points to directory inside ESP-IDF components dir
- `:component_file:`path`` - points to file inside ESP-IDF components dir
- `:component_raw:`path`` - points to raw view of the file inside ESP-IDF components dir
- `:example:`path`` - points to directory inside ESP-IDF examples dir
- `:example_file:`path`` - points to file inside ESP-IDF examples dir
- `:example_raw:`path`` - points to raw view of the file inside ESP-IDF examples dir

Example implementation:

```
* :example:`get-started/hello_world`
* :example:`Hello World! <get-started/hello_world>`
```

How it renders:

- get-started/hello_world
- Hello World!

A check is added to the CI build script, which searches RST files for presence of hard-coded links (identified by tree/master, blob/master, or raw/master part of the URL). This check can be run manually: `cd docs` and then `make gh-linkcheck`.

Linking Language Versions

Switching between documentation in different languages may be done using `:link_to_translation:` custom role. The role placed on a page of documentation provides a link to the same page in a language specified as a parameter. Examples below show how to enter links to Chinese and English versions of documentation:

```
:link_to_translation:`zh_CN:`
:link_to_translation:`en:English`
```

The language is specified using standard abbreviations like `en` or `zh_CN`. The text after last semicolon is not standardized and may be entered depending on the context where the link is placed, e.g.:

```
:link_to_translation:`en:see description in English`
```

Add Illustrations

Consider adding diagrams and pictures to illustrate described concepts.

Sometimes it is better to add an illustration than writing a lengthy paragraph to describe a complex idea, a data structure or an algorithm. This repository is using blockdiag suite of tools to generate diagram images from simple text files.

The following types of diagrams are supported:

- Block diagram
- Sequence diagram
- Activity diagram
- Logical network diagram

With this suite of tools it is possible to generate beautiful diagram images from simple text format (similar to graphviz's DOT format). The diagram elements are laid out automatically. The diagram code is then converted into ".png" graphics and integrated "behind the scenes" into **Sphinx** documents.

For the diagram preparation you can use an on-line interactive shell that instantly shows the rendered image.

Below are couple of diagram examples:

- Simple **block diagram** / `blockdiag` - Wi-Fi Buffer Configuration
- Slightly more complicated **block diagram** - Wi-Fi programming model
- **Sequence diagram** / `seqdiag` - Scan for a Specific AP in All Channels
- **Packet diagram** / `packetdiag` - NVS Page Structure

Try them out by modifying the source code and see the diagram instantly rendering below.

Note: There may be slight differences in rendering of font used by the interactive shell compared to the font used in the esp-idf documentation.

Put it all together

Once documentation is ready, follow instruction in *API Documentation Template* and create a single file, that will merge all individual pieces of prepared documentation. Finally add a link to this file to respective `.. toctree::` in `index.rst` file located in `/docs` folder or subfolders.

OK, but I am new to Sphinx!

1. No worries. All the software you need is well documented. It is also open source and free. Start by checking Sphinx documentation. If you are not clear how to write using rst markup language, see reStructuredText Primer. You can also use markdown (.md) files, and find out about more about the specific markdown syntax that we use on'Recommonmark parser's documentation page <https://recommonmark.readthedocs.io/en/latest/>'_.
2. Check the source files of this documentation to understand what is behind of what you see now on the screen. Sources are maintained on GitHub in espressif/esp-idf repository in docs folder. You can go directly to the source file of this page by scrolling up and clicking the link in the top right corner. When on GitHub, see what's really inside, open source files by clicking `Raw` button.
3. You will likely want to see how documentation builds and looks like before posting it on the GitHub. There are two options to do so:
   - Install Sphinx, Breathe, Blockdiag and Doxygen to build it locally, see chapter below.
   - Set up an account on Read the Docs and build documentation in the cloud. Read the Docs provides document building and hosting for free and their service works really quick and great.
4. To preview documentation before building, use Sublime Text editor together with OmniMarkupPreviewer plugin.

Setup for building documentation locally

You can setup environment to build documentation locally on your PC by installing:

1. Doxygen - https://www.stack.nl/~dimitri/doxygen/
2. Sphinx - https://github.com/sphinx-doc/sphinx/#readme-for-sphinx
3. Docment theme "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
6. Recommonmark - https://github.com/rtfd/recommonmark

The package "sphinx_rtd_theme" is added to have the same "look and feel" of ESP32 Programming Guide documentation like on the "Read the Docs" hosting site.

Do not worry about being confronted with several packages to install. Besides Doxygen, all remaining packages are written in Python. Therefore installation of all of them is combined into one simple step.

Installation of Doxygen is OS dependent:

Linux

```
sudo apt-get install doxygen
```

Windows - install in MSYS2 console

```
pacman -S doxygen
```

MacOS

```
brew install doxygen
```

Note: If you are installing on Windows system (Linux and MacOS users should skip this note), **before** going further, execute two extra steps below. These steps are required to install dependencies of "blockdiag" discussed under *Add Illustrations*.

1. Update all the system packages:

   ```
   $ pacman -Syu
   ```

   This process will likely require restarting of the MSYS2 MINGW32 console and repeating above commands, until update is complete.

2. Install *pillow*, that is one of dependences of the *blockdiag*:

   ```
   $ pacman -S mingw32/mingw-w64-i686-python2-pillow
   ```

   Check the log on the screen that `mingw-w64-i686-python2-pillow-4.3.0-1` is installed. Previous versions of *pillow* will not work.

A downside of Windows installation is that fonts of the *blockdiag pictures <add-illustrations>* do not render correctly, you will see some random characters instead. Until this issue is fixed, you can use the interactive shell to see how the complete picture looks like.

All remaining applications are Python packages and you can install them in one step as follows:

```
cd ~/esp/esp-idf/docs
pip install -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/_buld/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*

6.5.3 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 that using `*.inc` files, you may want to describe API in you 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 build, check how this section has been rendered. *Correct annotations* in respective header files, if required.

6.5.4 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.

"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 LO-

CAL LAW THAT MOST CLOSELY APPROXIMATES AN ABSOLUTE WAIVER OF ALL CIVIL LIABILITY IN CONNECTION WITH THE CONTRIBUTION.

7. Term

7.1 This Agreement shall come into effect upon Your acceptance of the terms and conditions.

7.2 In the event of a termination of this Agreement Sections 4, 5, 6, 7 and 8 shall survive such termination and shall remain in full force thereafter. For the avoidance of doubt, Contributions that are already licensed under a free and open source license at the date of the termination shall remain in full force after the termination of this Agreement.

8. Miscellaneous

8.1 This Agreement and all disputes, claims, actions, suits or other proceedings arising out of this agreement or relating in any way to it shall be governed by the laws of People's Republic of China excluding its private international law provisions.

8.2 This Agreement sets out the entire agreement between You and Us for Your Contributions to Us and overrides all other agreements or understandings.

8.3 If any provision of this Agreement is found void and unenforceable, such provision will be replaced to the extent possible with a provision that comes closest to the meaning of the original provision and that is enforceable. The terms and conditions set forth in this Agreement shall apply notwithstanding any failure of essential purpose of this Agreement or any limited remedy to the maximum extent possible under law.

8.4 You agree to notify Us of any facts or circumstances of which you become aware that would make this Agreement inaccurate in any respect.

You

Date:	
Name:	
Title:	
Address:	

Us

Date:	
Name:	
Title:	
Address:	

CHAPTER 7

ESP-IDF Versions

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.

7.1 Releases

Documentation for the current stable version can always be found at this URL:

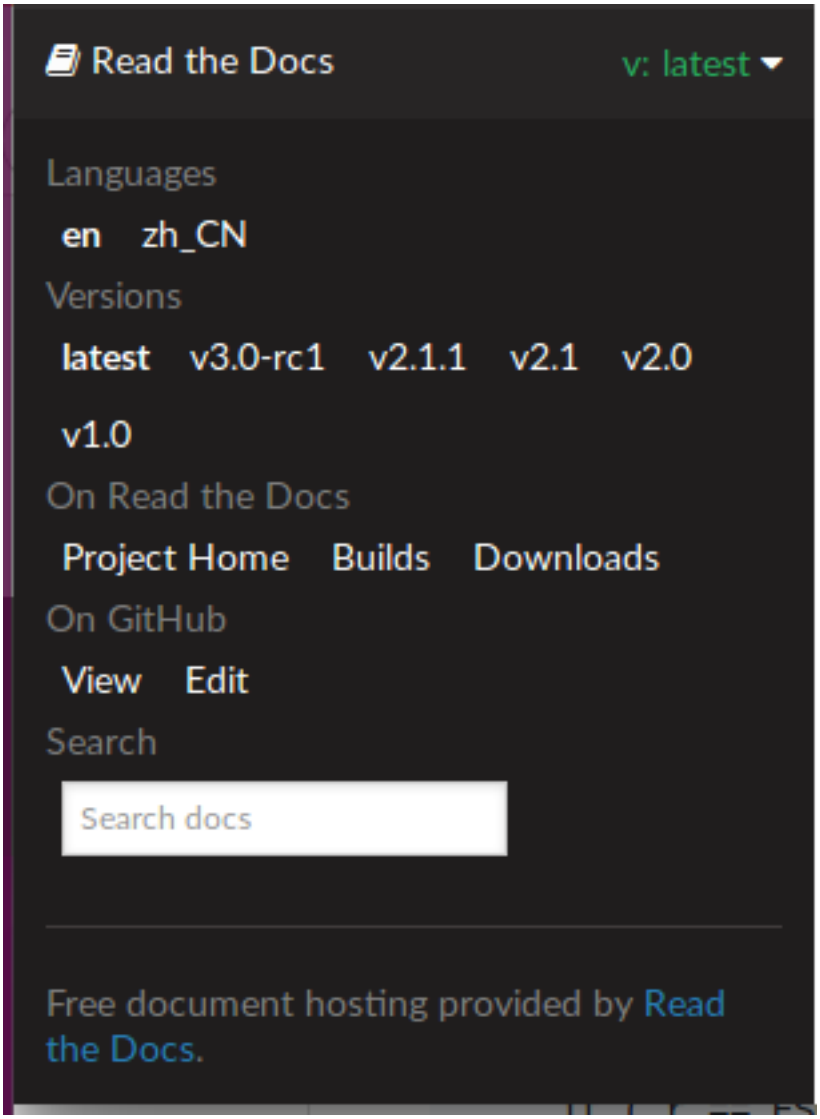
https://docs.espressif.com/projects/esp-idf/en/stable/

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.



7.2 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.

7.3 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.

7.4 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
`v3.2-dev-306-gbeb3611ca`	Master branch pre-release, in development for version 3.2. 306 commits after v3.2 development started. Commit identifier `beb3611ca`.
`v3.0.2`	Stable release, tagged `v3.0.2`.
`v3.1-beta1-75-g346d6b0ea`	Beta version in development (on a *release branch*). 75 commits after `v3.1-beta1` pre-release tag. Commit identifier `346d6b0ea`.
`v3.0.1-dirty`	Stable release, tagged `v3.0.1`. There are modifications in the local ESP-IDF directory ("`dirty`").

7.5 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.

7.6 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.

7.6.1 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.

7.6.2 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*.

7.6.3 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 unsupportable "snapshots" that may have undocumented bugs. For a semi-stable version, try *Updating to a Release Branch* instead.

7.6.4 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: The 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 8

Resources

- The esp32.com forum is a place to ask questions and find community resources.
- Check the Issues section on GitHub if you find a bug or have a feature request. Please check existing Issues before opening a new one.
- A comprehensive collection of solutions, practical applications, components and drivers based on ESP-IDF is available in ESP32 IoT Solution repository. In most of cases descriptions are provided both in English and in .
- To develop applications using Arduino platform, refer to Arduino core for ESP32 WiFi chip.
- Several books have been written about ESP32 and they are listed on Espressif web site.
- If you're interested in contributing to ESP-IDF, please check the *Contributions Guide*.
- For additional ESP32 product related information, please refer to documentation section of Espressif site.
- Mirror of this documentation is available under: https://espressif-docs.readthedocs-hosted.com/projects/esp-idf/en/latest/ and https://dl.espressif.com/doc/esp-idf/latest/.

CHAPTER 9

Copyrights and Licenses

9.1 Software Copyrights

All original source code in this repository is Copyright (C) 2015-2018 Espressif Systems. This source code is licensed under the Apache License 2.0 as described in the file LICENSE.

Additional third party copyrighted code is included under the following licenses.

Where source code headers specify Copyright & License information, this information takes precedence over the summaries made here.

9.1.1 Firmware Components

These third party libraries can be included into the application (firmware) produced by ESP-IDF.

- Newlib is licensed under the BSD License and is Copyright of various parties, as described in COPYING.NEWLIB.
- Xtensa header files are Copyright (C) 2013 Tensilica Inc and are licensed under the MIT License as reproduced in the individual header files.
- Original parts of FreeRTOS (components/freertos) are Copyright (C) 2015 Real Time Engineers Ltd and is licensed under the GNU General Public License V2 with the FreeRTOS Linking Exception, as described in license.txt.
- Original parts of LWIP (components/lwip) are Copyright (C) 2001, 2002 Swedish Institute of Computer Science and are licensed under the BSD License as described in COPYING file.
- wpa_supplicant Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- FreeBSD net80211 Copyright (c) 2004-2008 Sam Leffler, Errno Consulting and licensed under the BSD license.
- JSMN JSON Parser (components/jsmn) Copyright (c) 2010 Serge A. Zaitsev and licensed under the MIT license.
- argtable3 argument parsing library Copyright (C) 1998-2001,2003-2011,2013 Stewart Heitmann and licensed under 3-clause BSD license.

- linenoise line editing library Copyright (c) 2010-2014 Salvatore Sanfilippo, Copyright (c) 2010-2013 Pieter Noordhuis, licensed under 2-clause BSD license.
- libcoap COAP library Copyright (c) 2010-2017 Olaf Bergmann and others, is licensed under 2-clause BSD license.
- libexpat XML parsing library Copyright (c) 1998-2000 Thai Open Source Software Center Ltd and Clark Cooper, Copyright (c) 2001-2016 Expat maintainers, is licensed under MIT license.
- FatFS library, Copyright (C) 2017 ChaN, is licensed under a BSD-style license .
- cJSON library, Copyright (c) 2009-2017 Dave Gamble and cJSON contributors, is licensed under MIT license.
- libsodium library, Copyright (c) 2013-2018 Frank Denis, is licensed under ISC license.
- micro-ecc library, Copyright (c) 2014 Kenneth MacKay, is licensed under 2-clause BSD license.
- nghttp2 library, Copyright (c) 2012, 2014, 2015, 2016 Tatsuhiro Tsujikawa, Copyright (c) 2012, 2014, 2015, 2016 nghttp2 contributors, is licensed under MIT license.
- Mbed TLS library, Copyright (C) 2006-2018 ARM Limited, is licensed under Apache License 2.0.
- SPIFFS library, Copyright (c) 2013-2017 Peter Andersson, is licensed under MIT license.
- SD/MMC driver is derived from OpenBSD SD/MMC driver, Copyright (c) 2006 Uwe Stuehler, and is licensed under BSD license.

9.1.2 Build Tools

This is the list of licenses for tools included in this repository, which are used to build applications. The tools do not become part of the application (firmware), so their license does not affect licensing of the application.

- esptool.py is Copyright (C) 2014-2016 Fredrik Ahlberg, Angus Gratton and is licensed under the GNU General Public License v2, as described in LICENSE file.
- KConfig is Copyright (C) 2002 Roman Zippel and others, and is licensed under the GNU General Public License V2.

9.2 ROM Source Code Copyrights

ESP32 mask ROM hardware includes binaries compiled from portions of the following third party software:

- Newlib , licensed under the BSD License and is Copyright of various parties, as described in COPYING.NEWLIB.
- Xtensa libhal, Copyright (c) Tensilica Inc and licensed under the MIT license (see below).
- TinyBasic Plus, Copyright Mike Field & Scott Lawrence and licensed under the MIT license (see below).
- miniz, by Rich Geldreich - placed into the public domain.
- wpa_supplicant Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- TJpgDec Copyright (C) 2011, ChaN, all right reserved. See below for license.

9.3 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.

9.4 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.

9.5 TJpgDec License

TJpgDec - Tiny JPEG Decompressor R0.01 (C)ChaN, 2011 The TJpgDec is a generic JPEG decompressor module for tiny embedded systems. This is a free software that opened for education, research and commercial developments under license policy of following terms.

Copyright (C) 2011, ChaN, all right reserved.

- The TJpgDec module is a free software and there is NO WARRANTY.
- No restriction on use. You can use, modify and redistribute it for personal, non-profit or commercial products UNDER YOUR RESPONSIBILITY.
- Redistributions of source code must retain the above copyright notice.

CHAPTER 10

About

This is documentation of ESP-IDF, the framework to develop applications for ESP32 chip by Espressif.

The ESP32 is 2.4 GHz Wi-Fi and Bluetooth combo, 32 bit dual core chip with 600 DMIPS processing power.

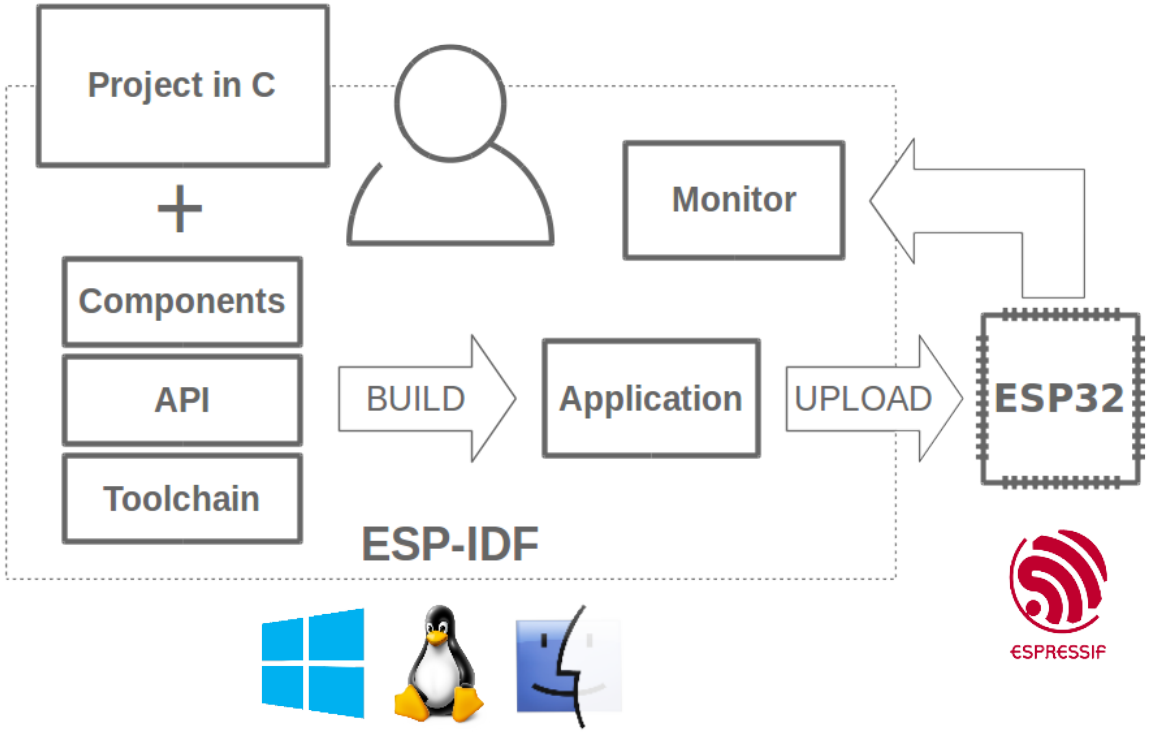


Fig. 1: Espressif IoT Integrated Development Framework

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.

CHAPTER 11

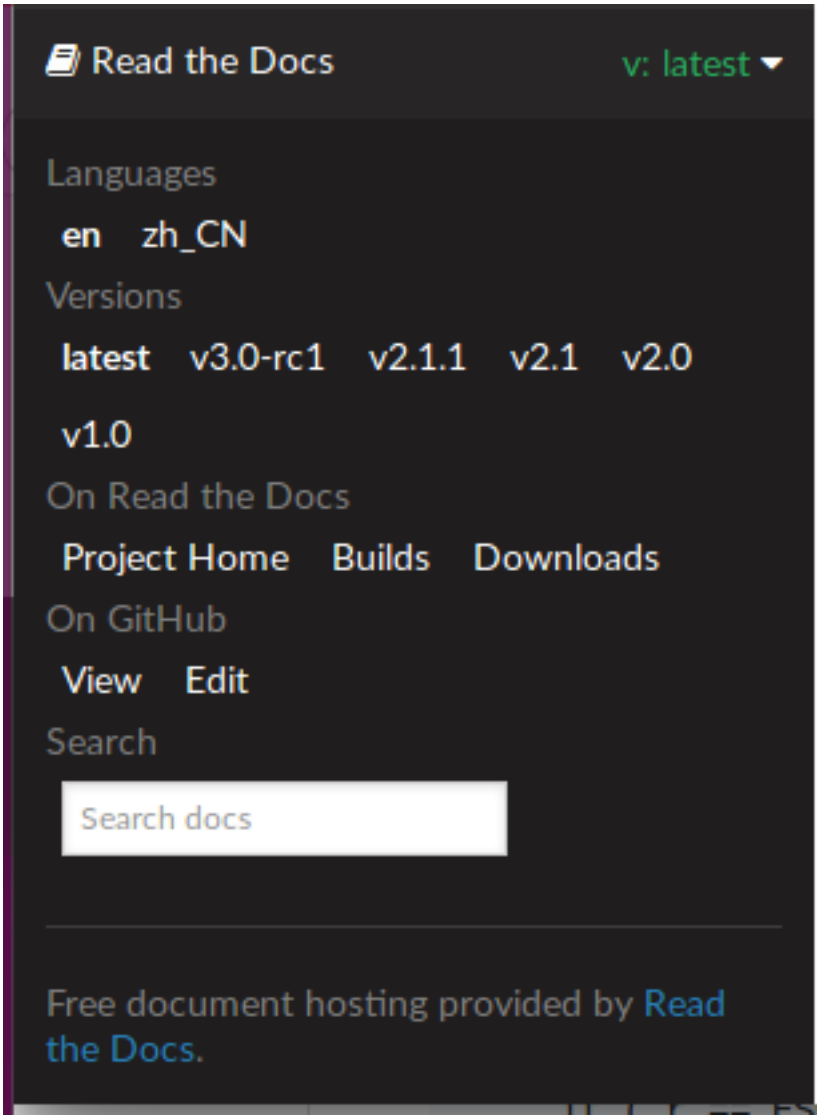
Switch Between Languages/

The documentation now has two language versions, however, please refer to the English version if there is any discrepancy.

- English/
- Chinese/

You can easily changed from one language to another by the panel on the sidebar like below. Just click on the **Read the Docs** title button on the left-bottom conner if it is folded.

Read the Docs


Read the Docs
v: latest
Languages
en
zh_CN
Versions
latest
v3.0-rc1
v2.1.1
v2.1
v2.0
v1.0
On Read the Docs
Project Home
Builds
Downloads
On GitHub
View
Edit
Search
Search docs
Free document hosting provided by Read the Docs.

- genindex

Index

A

ADC1_CHANNEL_0 (*C++ enumerator*), 369
ADC1_CHANNEL_0_GPIO_NUM (*C macro*), 373
ADC1_CHANNEL_1 (*C++ enumerator*), 369
ADC1_CHANNEL_1_GPIO_NUM (*C macro*), 373
ADC1_CHANNEL_2 (*C++ enumerator*), 369
ADC1_CHANNEL_2_GPIO_NUM (*C macro*), 374
ADC1_CHANNEL_3 (*C++ enumerator*), 369
ADC1_CHANNEL_3_GPIO_NUM (*C macro*), 374
ADC1_CHANNEL_4 (*C++ enumerator*), 369
ADC1_CHANNEL_4_GPIO_NUM (*C macro*), 374
ADC1_CHANNEL_5 (*C++ enumerator*), 369
ADC1_CHANNEL_5_GPIO_NUM (*C macro*), 374
ADC1_CHANNEL_6 (*C++ enumerator*), 369
ADC1_CHANNEL_6_GPIO_NUM (*C macro*), 374
ADC1_CHANNEL_7 (*C++ enumerator*), 369
ADC1_CHANNEL_7_GPIO_NUM (*C macro*), 374
ADC1_CHANNEL_MAX (*C++ enumerator*), 369
adc1_channel_t (*C++ type*), 369
adc1_config_channel_atten (*C++ function*), 364
adc1_config_width (*C++ function*), 363
adc1_get_raw (*C++ function*), 364
ADC1_GPIO32_CHANNEL (*C macro*), 374
ADC1_GPIO33_CHANNEL (*C macro*), 374
ADC1_GPIO34_CHANNEL (*C macro*), 374
ADC1_GPIO35_CHANNEL (*C macro*), 374
ADC1_GPIO36_CHANNEL (*C macro*), 373
ADC1_GPIO37_CHANNEL (*C macro*), 373
ADC1_GPIO38_CHANNEL (*C macro*), 373
ADC1_GPIO39_CHANNEL (*C macro*), 374
adc1_pad_get_io_num (*C++ function*), 363
adc1_ulp_enable (*C++ function*), 366
ADC2_CHANNEL_0 (*C++ enumerator*), 369
ADC2_CHANNEL_0_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_1 (*C++ enumerator*), 369
ADC2_CHANNEL_1_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_2 (*C++ enumerator*), 369
ADC2_CHANNEL_2_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_3 (*C++ enumerator*), 369
ADC2_CHANNEL_3_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_4 (*C++ enumerator*), 369
ADC2_CHANNEL_4_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_5 (*C++ enumerator*), 369
ADC2_CHANNEL_5_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_6 (*C++ enumerator*), 369
ADC2_CHANNEL_6_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_7 (*C++ enumerator*), 369
ADC2_CHANNEL_7_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_8 (*C++ enumerator*), 369
ADC2_CHANNEL_8_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_9 (*C++ enumerator*), 370
ADC2_CHANNEL_9_GPIO_NUM (*C macro*), 374
ADC2_CHANNEL_MAX (*C++ enumerator*), 370
adc2_channel_t (*C++ type*), 369
adc2_config_channel_atten (*C++ function*), 366
adc2_get_raw (*C++ function*), 367
ADC2_GPIO0_CHANNEL (*C macro*), 374
ADC2_GPIO12_CHANNEL (*C macro*), 374
ADC2_GPIO13_CHANNEL (*C macro*), 374
ADC2_GPIO14_CHANNEL (*C macro*), 374
ADC2_GPIO15_CHANNEL (*C macro*), 374
ADC2_GPIO25_CHANNEL (*C macro*), 374
ADC2_GPIO26_CHANNEL (*C macro*), 374
ADC2_GPIO27_CHANNEL (*C macro*), 374
ADC2_GPIO2_CHANNEL (*C macro*), 374
ADC2_GPIO4_CHANNEL (*C macro*), 374
adc2_pad_get_io_num (*C++ function*), 366
adc2_vref_to_gpio (*C++ function*), 367
ADC_ATTEN_0db (*C macro*), 368
ADC_ATTEN_11db (*C macro*), 368
ADC_ATTEN_2_5db (*C macro*), 368
ADC_ATTEN_6db (*C macro*), 368
ADC_ATTEN_DB_0 (*C++ enumerator*), 368
ADC_ATTEN_DB_11 (*C++ enumerator*), 368
ADC_ATTEN_DB_2_5 (*C++ enumerator*), 368
ADC_ATTEN_DB_6 (*C++ enumerator*), 368
ADC_ATTEN_MAX (*C++ enumerator*), 368

adc_atten_t (*C++ type*), 368
adc_bits_width_t (*C++ type*), 368
ADC_CHANNEL_0 (*C++ enumerator*), 370
ADC_CHANNEL_1 (*C++ enumerator*), 370
ADC_CHANNEL_2 (*C++ enumerator*), 370
ADC_CHANNEL_3 (*C++ enumerator*), 370
ADC_CHANNEL_4 (*C++ enumerator*), 370
ADC_CHANNEL_5 (*C++ enumerator*), 370
ADC_CHANNEL_6 (*C++ enumerator*), 370
ADC_CHANNEL_7 (*C++ enumerator*), 370
ADC_CHANNEL_8 (*C++ enumerator*), 370
ADC_CHANNEL_9 (*C++ enumerator*), 370
ADC_CHANNEL_MAX (*C++ enumerator*), 370
adc_channel_t (*C++ type*), 370
ADC_ENCODE_11BIT (*C++ enumerator*), 370
ADC_ENCODE_12BIT (*C++ enumerator*), 370
ADC_ENCODE_MAX (*C++ enumerator*), 371
adc_gpio_init (*C++ function*), 365
ADC_I2S_DATA_SRC_ADC (*C++ enumerator*), 371
ADC_I2S_DATA_SRC_IO_SIG (*C++ enumerator*), 371
ADC_I2S_DATA_SRC_MAX (*C++ enumerator*), 371
adc_i2s_encode_t (*C++ type*), 370
adc_i2s_mode_init (*C++ function*), 366
adc_i2s_source_t (*C++ type*), 371
adc_power_off (*C++ function*), 365
adc_power_on (*C++ function*), 365
adc_set_clk_div (*C++ function*), 365
adc_set_data_inv (*C++ function*), 365
adc_set_data_width (*C++ function*), 363
adc_set_i2s_data_source (*C++ function*), 365
ADC_UNIT_1 (*C++ enumerator*), 370
ADC_UNIT_2 (*C++ enumerator*), 370
ADC_UNIT_ALTER (*C++ enumerator*), 370
ADC_UNIT_BOTH (*C++ enumerator*), 370
ADC_UNIT_MAX (*C++ enumerator*), 370
adc_unit_t (*C++ type*), 370
ADC_WIDTH_10Bit (*C macro*), 368
ADC_WIDTH_11Bit (*C macro*), 368
ADC_WIDTH_12Bit (*C macro*), 368
ADC_WIDTH_9Bit (*C macro*), 368
ADC_WIDTH_BIT_10 (*C++ enumerator*), 368
ADC_WIDTH_BIT_11 (*C++ enumerator*), 368
ADC_WIDTH_BIT_12 (*C++ enumerator*), 369
ADC_WIDTH_BIT_9 (*C++ enumerator*), 368
ADC_WIDTH_MAX (*C++ enumerator*), 369
ADV_CHNL_37 (*C++ enumerator*), 238
ADV_CHNL_38 (*C++ enumerator*), 238
ADV_CHNL_39 (*C++ enumerator*), 238
ADV_CHNL_ALL (*C++ enumerator*), 238
ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY (*C++ enumerator*), 238
ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST (*C++ enumerator*), 238
ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY (*C++ enumerator*), 238
ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST (*C++ enumerator*), 238
ADV_TYPE_DIRECT_IND_HIGH (*C++ enumerator*), 238
ADV_TYPE_DIRECT_IND_LOW (*C++ enumerator*), 238
ADV_TYPE_IND (*C++ enumerator*), 238
ADV_TYPE_NONCONN_IND (*C++ enumerator*), 238
ADV_TYPE_SCAN_IND (*C++ enumerator*), 238

B

BLE_ADDR_TYPE_PUBLIC (*C++ enumerator*), 211
BLE_ADDR_TYPE_RANDOM (*C++ enumerator*), 211
BLE_ADDR_TYPE_RPA_PUBLIC (*C++ enumerator*), 211
BLE_ADDR_TYPE_RPA_RANDOM (*C++ enumerator*), 211
BLE_SCAN_DUPLICATE_DISABLE (*C++ enumerator*), 240
BLE_SCAN_DUPLICATE_ENABLE (*C++ enumerator*), 240
BLE_SCAN_DUPLICATE_MAX (*C++ enumerator*), 240
BLE_SCAN_FILTER_ALLOW_ALL (*C++ enumerator*), 239
BLE_SCAN_FILTER_ALLOW_ONLY_WLST (*C++ enumerator*), 239
BLE_SCAN_FILTER_ALLOW_UND_RPA_DIR (*C++ enumerator*), 239
BLE_SCAN_FILTER_ALLOW_WLIST_PRA_DIR (*C++ enumerator*), 240
BLE_SCAN_TYPE_ACTIVE (*C++ enumerator*), 239
BLE_SCAN_TYPE_PASSIVE (*C++ enumerator*), 239
BT_CONTROLLER_INIT_CONFIG_DEFAULT (*C macro*), 206

C

CONFIG_A2D_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_A2DP_ENABLE, 860, 891
CONFIG_A2DP_SINK_TASK_STACK_SIZE, 861, 891
CONFIG_A2DP_SOURCE_TASK_STACK_SIZE, 861, 891
CONFIG_ADC2_DISABLE_DAC, 858, 879
CONFIG_ADC_CAL_EFUSE_TP_ENABLE, 868
CONFIG_ADC_CAL_EFUSE_VREF_ENABLE, 868
CONFIG_ADC_CAL_LUT_ENABLE, 868
CONFIG_ADC_FORCE_XPD_FSM, 858, 879
CONFIG_APPL_INITIAL_TRACE_LEVEL, 862, 892
CONFIG_AVCT_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_AVDT_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_AVRC_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_AWS_IOT_MQTT_HOST, 859

CONFIG_AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL, 859
CONFIG_AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL, 859
CONFIG_AWS_IOT_MQTT_NUM_SUBSCRIBE_HANDLERS, 859
CONFIG_AWS_IOT_MQTT_PORT, 859
CONFIG_AWS_IOT_MQTT_RX_BUF_LEN, 859
CONFIG_AWS_IOT_MQTT_TX_BUF_LEN, 859
CONFIG_AWS_IOT_OVERRIDE_THING_SHADOW_RX_BUFFER, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_JSON_TOKEN_EXPECTED, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SHADOW_TOPIC_LENGTH, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_ACKS, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_THINGNAMES, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_RX_BUFFER, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_THING_NAME, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_UNIQUECLIENTID_BYTES, 859, 882
CONFIG_BLE_ACTIVE_SCAN_REPORT_ADV_SCAN_RSPNSE, 862, 893
CONFIG_BLE_ADV_REPORT_DISCARD_THRSHOLD, 862, 893
CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_NUM, 862, 893
CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_SUPPORTED, 862, 893
CONFIG_BLE_ESTABLISH_LINK_CONNECTION_TIMEOUT, 862, 893
CONFIG_BLE_HOST_QUEUE_CONGESTION_CHECK, 862, 893
CONFIG_BLE_MESH_SCAN_DUPLICATE_EN, 862, 893
CONFIG_BLE_SCAN_DUPLICATE, 862, 893
CONFIG_BLE_SMP_ENABLE, 861, 891
CONFIG_BLUEDROID_MEM_DEBUG, 860, 891
CONFIG_BLUEDROID_PINNED_TO_CORE, 860, 891
CONFIG_BLUEDROID_PINNED_TO_CORE_CHOICE, 860, 891
CONFIG_BLUFI_INITIAL_TRACE_LEVEL, 862, 893
CONFIG_BNEP_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_BOOTLOADER_APP_TEST, 847, 1085
CONFIG_BOOTLOADER_DATA_FACTORY_RESET, 847, 1085
CONFIG_BOOTLOADER_FACTORY_RESET, 847, 1084
CONFIG_BOOTLOADER_HOLD_TIME_GPIO, 847, 1085
CONFIG_BOOTLOADER_NUM_PIN_APP_TEST, 847, 1085
CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET, 847, 1085
CONFIG_BOOTLOADER_OTA_DATA_ERASE, 847, 1085
CONFIG_BOOTLOADER_SPI_WP_PIN, 847
CONFIG_BOOTLOADER_VDDSDIO_BOOST, 847
CONFIG_BROWNOUT_DET, 866, 1005
CONFIG_BROWNOUT_DET_LVL, 866
CONFIG_BROWNOUT_DET_LVL_SEL, 866, 1005
CONFIG_BT_ACL_CONNECTIONS, 862, 893
CONFIG_BT_BLE_ALLOCATION_FROM_SPIRAM_FIRST, 862, 893
CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY, 862, 893
CONFIG_BT_ENABLED, 860
CONFIG_BT_HCI_UART_BAUDRATE, 860, 890
CONFIG_BT_HCI_UART_NO, 860, 890
CONFIG_BT_RESERVE_DRAM, 862
CONFIG_BT_SPP_ENABLED, 861, 891
CONFIG_BT_STACK_NO_LOG, 861, 892
CONFIG_BTC_INITIAL_TRACE_LEVEL, 862, 893
CONFIG_BTC_TASK_STACK_SIZE, 860, 891
CONFIG_BTDM_CONTROLLER_AUTO_LATENCY, 860
CONFIG_BTDM_CONTROLLER_AUTO_LATENCY_EFF, 860
CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH, 860
CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH_EFF, 860
CONFIG_BTDM_CONTROLLER_HCI_MODE_CHOICE, 860
CONFIG_BTDM_CONTROLLER_MODEM_SLEEP, 860, 890
CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE, 860
CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE_CHOICE, 860
CONFIG_BTDM_LOW_POWER_CLOCK, 860, 890
CONFIG_BTDM_MODEM_SLEEP_MODE, 860, 890
CONFIG_BTIF_INITIAL_TRACE_LEVEL, 862, 892, 893
CONFIG_BTM_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_CLASSIC_BT_ENABLED, 860, 891
CONFIG_COMPATIBLE_PRE_V2_1_BOOTLOADERS, 866
CONFIG_CONSOLE_UART, 865
CONFIG_CONSOLE_UART_BAUDRATE, 866
CONFIG_CONSOLE_UART_NUM, 865, 866
CONFIG_CONSOLE_UART_RX_GPIO, 866
CONFIG_CONSOLE_UART_TX_GPIO, 866
CONFIG_CXX_EXCEPTIONS, 999

CONFIG_CXX_EXCEPTIONS_EMG_POOL_SIZE, 854, 999
CONFIG_DISABLE_BASIC_ROM_CONSOLE, 866
CONFIG_DMA_RX_BUF_NUM, 868
CONFIG_DMA_TX_BUF_NUM, 868
CONFIG_DUPLICATE_SCAN_CACHE_SIZE, 862, 893
CONFIG_EMAC_L2_TO_L3_RX_BUF_MODE, 868
CONFIG_EMAC_TASK_PRIORITY, 868
CONFIG_EMAC_TASK_STACK_SIZE, 868
CONFIG_ENABLE_STATIC_TASK_CLEAN_UP_HOOK, 868
CONFIG_ESP32_APPTRACE_DESTINATION, 864
CONFIG_ESP32_APPTRACE_ENABLE, 864
CONFIG_ESP32_APPTRACE_LOCK_ENABLE, 864
CONFIG_ESP32_APPTRACE_ONPANIC_HOST_FLUSH_TMO, 864, 1076
CONFIG_ESP32_APPTRACE_PENDING_DATA_SIZE_MAX, 864, 1076
CONFIG_ESP32_APPTRACE_POSTMORTEM_FLUSH_TRAX_THRESH, 864, 1076
CONFIG_ESP32_CORE_DUMP_LOG_LEVEL, 865
CONFIG_ESP32_CORE_DUMP_UART_DELAY, 865
CONFIG_ESP32_COREDUMP_TO_FLASH_OR_UART, 865
CONFIG_ESP32_DEBUG_OCDAWARE, 866, 1000, 1071
CONFIG_ESP32_DEBUG_STUBS_ENABLE, 866
CONFIG_ESP32_DEEP_SLEEP_WAKEUP_DELAY, 866
CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ, 819, 865
CONFIG_ESP32_DPORT_DIS_INTERRUPT_LVL, 867
CONFIG_ESP32_DPORT_WORKAROUND, 864
CONFIG_ESP32_ENABLE_COREDUMP, 865
CONFIG_ESP32_GCOV_ENABLE, 864
CONFIG_ESP32_PANIC, 62, 123, 866, 1000
CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE, 867
CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION, 867, 1088
CONFIG_ESP32_PHY_MAX_TX_POWER, 867
CONFIG_ESP32_PHY_MAX_WIFI_TX_POWER, 867
CONFIG_ESP32_PTHREAD_TASK_PRIO_DEFAULT, 858
CONFIG_ESP32_PTHREAD_TASK_STACK_SIZE_DEFAULT, 858
CONFIG_ESP32_REV_MIN, 864
CONFIG_ESP32_RTC_CLK_CAL_CYCLES, 866
CONFIG_ESP32_RTC_CLOCK_SOURCE, 866
CONFIG_ESP32_RTC_EXTERNAL_CRYSTAL_ADDITIONAL_CURRENT, 866
CONFIG_ESP32_RTC_XTAL_BOOTSTRAP_CYCLES, 866
CONFIG_ESP32_TIME_SYSCALL, 866
CONFIG_ESP32_TRAX, 865
CONFIG_ESP32_TRAX_TWOBANKS, 865
CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED, 867
CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED, 867
CONFIG_ESP32_WIFI_CSI_ENABLED, 867
CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM, 867
CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM, 867
CONFIG_ESP32_WIFI_IRAM_OPT, 867
CONFIG_ESP32_WIFI_MGMT_SBUF_NUM, 867
CONFIG_ESP32_WIFI_NVS_ENABLED, 867
CONFIG_ESP32_WIFI_RX_BA_WIN, 867
CONFIG_ESP32_WIFI_RX_IRAM_OPT, 867
CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN, 867
CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM, 867
CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM, 867
CONFIG_ESP32_WIFI_TASK_CORE_ID, 867
CONFIG_ESP32_WIFI_TX_BA_WIN, 867
CONFIG_ESP32_WIFI_TX_BUFFER, 867
CONFIG_ESP32_WIFI_TX_BUFFER_TYPE, 867
CONFIG_ESP32_XTAL_FREQ, 866
CONFIG_ESP32_XTAL_FREQ_SEL, 48, 113, 866
CONFIG_ESP_ERR_TO_NAME_LOOKUP, 866, 998
CONFIG_ESP_GRATUITOUS_ARP, 863
CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS, 862
CONFIG_ESP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES, 863, 908
CONFIG_ESP_TIMER_PROFILING, 866
CONFIG_ESPTOOLPY_AFTER, 844
CONFIG_ESPTOOLPY_BAUD, 844
CONFIG_ESPTOOLPY_BAUD_OTHER_VAL, 844
CONFIG_ESPTOOLPY_BEFORE, 844
CONFIG_ESPTOOLPY_COMPRESSED, 844
CONFIG_ESPTOOLPY_FLASHFREQ, 844
CONFIG_ESPTOOLPY_FLASHMODE, 844
CONFIG_ESPTOOLPY_FLASHSIZE, 599, 844
CONFIG_ESPTOOLPY_FLASHSIZE_DETECT, 844
CONFIG_ESPTOOLPY_PORT, 844
CONFIG_FATFS_API_ENCODING, 860
CONFIG_FATFS_CHOOSE_CODEPAGE, 860
CONFIG_FATFS_CODEPAGE, 860
CONFIG_FATFS_FS_LOCK, 860
CONFIG_FATFS_LONG_FILENAMES, 860
CONFIG_FATFS_MAX_LFN, 860
CONFIG_FATFS_PER_FILE_CACHE, 860
CONFIG_FATFS_TIMEOUT_MS, 860
CONFIG_FLASH_ENCRYPTION_DISABLE_PLAINTEXT, 849, 1016

CONFIG_FLASH_ENCRYPTION_ENABLED, 849
CONFIG_FLASH_ENCRYPTION_INSECURE, 849
CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_CACHE, 849
CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_DECRYPT, 849
CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_ENCRYPT, 849
CONFIG_FLASHMODE, 844
CONFIG_FREERTOS_ASSERT, 867
CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION, 867, 1026
CONFIG_FREERTOS_CHECK_STACKOVERFLOW, 867
CONFIG_FREERTOS_CORETIMER, 867
CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS, 868
CONFIG_FREERTOS_HZ, 867
CONFIG_FREERTOS_IDLE_TASK_STACKSIZE, 868
CONFIG_FREERTOS_IDLE_TIME_BEFORE_SLEEP, 868
CONFIG_FREERTOS_INTERRUPT_BACKTRACE, 867
CONFIG_FREERTOS_ISR_STACKSIZE, 868
CONFIG_FREERTOS_LEGACY_HOOKS, 769, 868
CONFIG_FREERTOS_LEGACY_IDLE_HOOK, 769, 868
CONFIG_FREERTOS_LEGACY_TICK_HOOK, 769, 868
CONFIG_FREERTOS_MAX_TASK_NAME_LEN, 868
CONFIG_FREERTOS_PORTMUX_DEBUG, 868, 938
CONFIG_FREERTOS_PORTMUX_DEBUG_RECURSIVE, 868, 938
CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE, 868
CONFIG_FREERTOS_RUN_TIME_STATS_CLK, 868
CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS, 867, 1025
CONFIG_FREERTOS_UNICORE, 46, 110, 602, 867, 1025
CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS, 868
CONFIG_FREERTOS_USE_TICKLESS_IDLE, 819, 868
CONFIG_FREERTOS_USE_TRACE_FACILITY, 868
CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK, 867, 1070, 1071
CONFIG_GAP_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_GARP_TMR_INTERVAL, 863
CONFIG_GATT_INITIAL_TRACE_LEVEL, 862, 892
CONFIG_GATTC_CACHE_NVS_FLASH, 861, 891
CONFIG_GATTC_ENABLE, 861, 891
CONFIG_GATTS_ENABLE, 861, 891
CONFIG_HCI_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_HEAP_CORRUPTION_DETECTION, 785, 858
CONFIG_HEAP_TASK_TRACKING, 858
CONFIG_HEAP_TRACING, 786, 858
CONFIG_HEAP_TRACING_STACK_DEPTH, 858
CONFIG_HEAP_TRACING_STACK_DEPTH (*C macro*), 791
CONFIG_HFP_AUDIO_DATA_PATH, 861, 891
CONFIG_HFP_ENABLE, 861, 891
CONFIG_HFP_ROLE, 861, 891
CONFIG_HID_INITIAL_TRACE_LEVEL, 862, 892
CONFIG_INT_WDT, 866
CONFIG_INT_WDT_CHECK_CPU1, 866
CONFIG_INT_WDT_TIMEOUT_MS, 866
CONFIG_IP_LOST_TIMER_INTERVAL, 860
CONFIG_IPC_TASK_STACK_SIZE, 802, 865
CONFIG_L2_TO_L3_COPY, 862
CONFIG_L2CAP_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_LIBSODIUM_USE_MBEDTLS_SHA, 862
CONFIG_LOG_BOOTLOADER_LEVEL, 847
CONFIG_LOG_COLORS, 859
CONFIG_LOG_DEFAULT_LEVEL, 808, 859
CONFIG_LWIP_AUTOIP_MAX_CONFLICTS, 863, 907
CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL, 863, 907
CONFIG_LWIP_AUTOIP_TRIES, 863, 907
CONFIG_LWIP_BROADCAST_PING, 864, 912
CONFIG_LWIP_DHCP_DOES_ARP_CHECK, 863
CONFIG_LWIP_DHCP_MAX_NTP_SERVERS, 863
CONFIG_LWIP_DHCPS_LEASE_UNIT, 863, 906
CONFIG_LWIP_DHCPS_MAX_STATION_NUM, 863, 907
CONFIG_LWIP_ETHARP_TRUST_IP_MAC, 863
CONFIG_LWIP_IP_FRAG, 863
CONFIG_LWIP_IP_REASSEMBLY, 863
CONFIG_LWIP_IPV6_AUTOCONFIG, 863
CONFIG_LWIP_IRAM_OPTIMIZATION, 863
CONFIG_LWIP_LOOPBACK_MAX_PBUFS, 863, 908
CONFIG_LWIP_MAX_ACTIVE_TCP, 863, 908
CONFIG_LWIP_MAX_LISTENING_TCP, 863, 908
CONFIG_LWIP_MAX_RAW_PCBS, 864, 912
CONFIG_LWIP_MAX_SOCKETS, 863
CONFIG_LWIP_MAX_UDP_PCBS, 864, 911
CONFIG_LWIP_MULTICAST_PING, 864, 912
CONFIG_LWIP_SO_RCVBUF, 863
CONFIG_LWIP_SO_REUSE, 863
CONFIG_LWIP_SO_REUSE_RXTOALL, 863
CONFIG_LWIP_STATS, 863
CONFIG_LWIP_WND_SCALE, 863, 908
CONFIG_MAIN_TASK_STACK_SIZE, 865
CONFIG_MAKE_WARN_UNDEFINED_VARIABLES, 843
CONFIG_MBEDTLS_AES_C, 857, 874
CONFIG_MBEDTLS_BLOWFISH_C, 858, 874
CONFIG_MBEDTLS_CAMELLIA_C, 858, 874

CONFIG_MBEDTLS_CCM_C, 858, 874
CONFIG_MBEDTLS_DEBUG, 857
CONFIG_MBEDTLS_DES_C, 858, 874
CONFIG_MBEDTLS_ECDH_C, 858, 876
CONFIG_MBEDTLS_ECDSA_C, 858, 876
CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED, 858, 876
CONFIG_MBEDTLS_ECP_NIST_OPTIM, 858, 876
CONFIG_MBEDTLS_GCM_C, 858, 874
CONFIG_MBEDTLS_HARDWARE_AES, 857
CONFIG_MBEDTLS_HARDWARE_MPI, 857
CONFIG_MBEDTLS_HARDWARE_SHA, 857
CONFIG_MBEDTLS_HAVE_TIME, 857
CONFIG_MBEDTLS_HAVE_TIME_DATE, 857
CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_PSK, 857, 871
CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_RSA, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_PSK, 857, 871
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_PSK, 857, 871
CONFIG_MBEDTLS_KEY_EXCHANGE_RSA, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_RSA_PSK, 857, 871
CONFIG_MBEDTLS_MPI_USE_INTERRUPT, 857
CONFIG_MBEDTLS_PEM_PARSE_C, 858, 875
CONFIG_MBEDTLS_PEM_WRITE_C, 858, 875
CONFIG_MBEDTLS_PSK_MODES, 857, 871
CONFIG_MBEDTLS_RC4_MODE, 858, 874
CONFIG_MBEDTLS_RIPEMD160_C, 858
CONFIG_MBEDTLS_SSL_ALPN, 857
CONFIG_MBEDTLS_SSL_MAX_CONTENT_LEN, 857
CONFIG_MBEDTLS_SSL_PROTO_DTLS, 857
CONFIG_MBEDTLS_SSL_PROTO_SSL3, 857
CONFIG_MBEDTLS_SSL_PROTO_TLS1, 857
CONFIG_MBEDTLS_SSL_PROTO_TLS1_1, 857
CONFIG_MBEDTLS_SSL_PROTO_TLS1_2, 857
CONFIG_MBEDTLS_SSL_RENEGOTIATION, 857
CONFIG_MBEDTLS_SSL_SESSION_TICKETS, 857
CONFIG_MBEDTLS_TLS_CLIENT, 857
CONFIG_MBEDTLS_TLS_ENABLED, 857
CONFIG_MBEDTLS_TLS_MODE, 857
CONFIG_MBEDTLS_TLS_SERVER, 857
CONFIG_MBEDTLS_X509_CRL_PARSE_C, 858, 875
CONFIG_MBEDTLS_X509_CSR_PARSE_C, 858, 875
CONFIG_MBEDTLS_XTEA_C, 858, 874
CONFIG_MCA_INITIAL_TRACE_LEVEL, 862, 892
CONFIG_MEMMAP_TRACEMEM, 865
CONFIG_MEMMAP_TRACEMEM_TWOBANKS, 865
CONFIG_MESH_DUPLICATE_SCAN_CACHE_SIZE, 862, 893
CONFIG_MONITOR_BAUD, 844
CONFIG_MONITOR_BAUD_OTHER_VAL, 844
CONFIG_MP_BLOB_SUPPORT, 859
CONFIG_NEWLIB_NANO_FORMAT, 865
CONFIG_NEWLIB_STDIN_LINE_ENDING, 865
CONFIG_NEWLIB_STDOUT_LINE_ENDING, 865
CONFIG_NO_BLOBS, 866
CONFIG_NUMBER_OF_UNIVERSAL_MAC_ADDRESS, 865
CONFIG_NVS_ENCRYPTION, 859
CONFIG_OPENSSL_ASSERT, 857
CONFIG_OPENSSL_DEBUG, 856
CONFIG_OPENSSL_DEBUG_LEVEL, 856
CONFIG_OPENSSL_LOWLEVEL_DEBUG, 857
CONFIG_OPTIMIZATION_ASSERTION_LEVEL, 854
CONFIG_OPTIMIZATION_COMPILER, 854
CONFIG_OSI_INITIAL_TRACE_LEVEL, 862, 893
CONFIG_PAN_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME, 854
CONFIG_PARTITION_TABLE_FILENAME, 854
CONFIG_PARTITION_TABLE_MD5, 854, 1089
CONFIG_PARTITION_TABLE_OFFSET, 854
CONFIG_PARTITION_TABLE_TYPE, 854
CONFIG_PM_DFS_INIT_AUTO, 819, 867

CONFIG_PM_ENABLE, 819, 867
CONFIG_PM_PROFILING, 867
CONFIG_PM_TRACE, 867
CONFIG_PM_USE_RTC_TIMER_REF, 867
CONFIG_PPP_CHAP_SUPPORT, 864, 911
CONFIG_PPP_DEBUG_ON, 864, 911
CONFIG_PPP_MPPE_SUPPORT, 864, 911
CONFIG_PPP_MSCHAP_SUPPORT, 864, 911
CONFIG_PPP_PAP_SUPPORT, 864, 911
CONFIG_PYTHON, 843
CONFIG_RFCOMM_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_SDP_INITIAL_TRACE_LEVEL, 861, 892
CONFIG_SECURE_BOOT_ALLOW_JTAG, 849
CONFIG_SECURE_BOOT_ALLOW_ROM_BASIC, 849
CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION, 849
CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES, 849
CONFIG_SECURE_BOOT_ENABLED, 849, 1092
CONFIG_SECURE_BOOT_INSECURE, 849
CONFIG_SECURE_BOOT_SIGNING_KEY, 849
CONFIG_SECURE_BOOT_TEST_MODE, 849
CONFIG_SECURE_BOOT_VERIFICATION_KEY, 849
CONFIG_SECURE_BOOTLOADER_KEY_ENCODING, 849, 1092
CONFIG_SECURE_BOOTLOADER_MODE, 849, 1092
CONFIG_SECURE_SIGNED_APPS, 849
CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT, 849
CONFIG_SECURE_SIGNED_ON_BOOT, 849
CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT, 849
CONFIG_SECURE_SIGNED_ON_UPDATE, 849
CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT, 849
CONFIG_SMP_ENABLE, 862, 893
CONFIG_SMP_INITIAL_TRACE_LEVEL, 862, 892
CONFIG_SMP_SLAVE_CON_PARAMS_UPD_ENABLE, 861, 891
CONFIG_SPI_FLASH_ENABLE_COUNTERS, 868
CONFIG_SPI_FLASH_ERASE_YIELD_DURATION_MS, 868
CONFIG_SPI_FLASH_ERASE_YIELD_TICKS, 869
CONFIG_SPI_FLASH_LOG_FAILED_WRITE, 868
CONFIG_SPI_FLASH_ROM_DRIVER_PATCH, 868
CONFIG_SPI_FLASH_VERIFY_WRITE, 868
CONFIG_SPI_FLASH_WARN_SETTING_ZERO_TO_ONE, 868
CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS, 868
CONFIG_SPI_FLASH_YIELD_DURING_ERASE, 868
CONFIG_SPI_MASTER_IN_IRAM, 506, 858, 880
CONFIG_SPI_MASTER_ISR_IN_IRAM, 506, 859, 880
CONFIG_SPIFFS_API_DBG, 859, 885
CONFIG_SPIFFS_CACHE, 859, 883
CONFIG_SPIFFS_CACHE_DBG, 860, 885
CONFIG_SPIFFS_CACHE_STATS, 859, 883
CONFIG_SPIFFS_CACHE_WR, 859, 883
CONFIG_SPIFFS_CHECK_DBG, 860, 885
CONFIG_SPIFFS_DBG, 859, 885
CONFIG_SPIFFS_GC_DBG, 860, 885
CONFIG_SPIFFS_GC_MAX_RUNS, 859
CONFIG_SPIFFS_GC_STATS, 859
CONFIG_SPIFFS_MAX_PARTITIONS, 859
CONFIG_SPIFFS_META_LENGTH, 859
CONFIG_SPIFFS_OBJ_NAME_LEN, 859
CONFIG_SPIFFS_PAGE_CHECK, 859
CONFIG_SPIFFS_PAGE_SIZE, 859
CONFIG_SPIFFS_TEST_VISUALISATION, 860, 885
CONFIG_SPIFFS_USE_MAGIC, 859
CONFIG_SPIFFS_USE_MAGIC_LENGTH, 859
CONFIG_SPIFFS_USE_MTIME, 859
CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY, 865, 917, 1190
CONFIG_SPIRAM_BOOT_INIT, 865, 916
CONFIG_SPIRAM_CACHE_WORKAROUND, 865, 916, 975, 991, 1191
CONFIG_SPIRAM_IGNORE_NOTFOUND, 865, 916
CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL, 865, 916
CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL, 865, 917
CONFIG_SPIRAM_MEMTEST, 865, 916
CONFIG_SPIRAM_SIZE, 865, 916
CONFIG_SPIRAM_SPEED, 865, 916
CONFIG_SPIRAM_SUPPORT, 865
CONFIG_SPIRAM_TYPE, 865, 916
CONFIG_SPIRAM_USE, 865, 916
CONFIG_STACK_CHECK, 854
CONFIG_STACK_CHECK_MODE, 854, 1005
CONFIG_SUPPORT_STATIC_ALLOCATION, 868, 1020, 1025
CONFIG_SUPPRESS_SELECT_DEBUG_OUTPUT, 857
CONFIG_SW_COEXIST_ENABLE, 866
CONFIG_SW_COEXIST_PREFERENCE, 866
CONFIG_SW_COEXIST_PREFERENCE_VALUE, 866
CONFIG_SYSTEM_EVENT_QUEUE_SIZE, 865
CONFIG_SYSTEM_EVENT_TASK_STACK_SIZE, 865
CONFIG_SYSVIEW_ENABLE, 864, 913, 1082
CONFIG_SYSVIEW_EVT_IDLE_ENABLE, 864, 914
CONFIG_SYSVIEW_EVT_ISR_ENTER_ENABLE, 864, 913
CONFIG_SYSVIEW_EVT_ISR_EXIT_ENABLE, 864, 913

CONFIG_SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE, 864, 913
CONFIG_SYSVIEW_EVT_OVERFLOW_ENABLE, 864, 913
CONFIG_SYSVIEW_EVT_TASK_CREATE_ENABLE, 864, 914
CONFIG_SYSVIEW_EVT_TASK_START_EXEC_ENABLE, 864, 913
CONFIG_SYSVIEW_EVT_TASK_START_READY_ENABLE, 864, 914
CONFIG_SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE, 864, 913
CONFIG_SYSVIEW_EVT_TASK_STOP_READY_ENABLE, 864, 914
CONFIG_SYSVIEW_EVT_TASK_TERMINATE_ENABLE, 864, 914
CONFIG_SYSVIEW_EVT_TIMER_ENTER_ENABLE, 864, 914
CONFIG_SYSVIEW_EVT_TIMER_EXIT_ENABLE, 864, 914
CONFIG_SYSVIEW_TS_SOURCE, 864, 913, 1082
CONFIG_TASK_WDT, 799, 866
CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU0, 799, 866
CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU1, 799, 866
CONFIG_TASK_WDT_PANIC, 866, 999
CONFIG_TASK_WDT_TIMEOUT_S, 866
CONFIG_TCP_MAXRTX, 863, 908
CONFIG_TCP_MSL, 863, 908
CONFIG_TCP_MSS, 863, 908
CONFIG_TCP_OVERSIZE, 863, 908
CONFIG_TCP_QUEUE_OOSEQ, 863, 908
CONFIG_TCP_RCV_SCALE, 863, 908
CONFIG_TCP_RECVMBOX_SIZE, 863, 908
CONFIG_TCP_SND_BUF_DEFAULT, 863, 908
CONFIG_TCP_SYNMAXRTX, 863, 908
CONFIG_TCP_WND_DEFAULT, 863, 908
CONFIG_TCPIP_RECVMBOX_SIZE, 863
CONFIG_TCPIP_TASK_STACK_SIZE, 864
CONFIG_TIMER_QUEUE_LENGTH, 868
CONFIG_TIMER_TASK_PRIORITY, 868
CONFIG_TIMER_TASK_STACK_DEPTH, 868
CONFIG_TIMER_TASK_STACK_SIZE, 865
CONFIG_TOOLPREFIX, 843
CONFIG_TRACEMEM_RESERVE_DRAM, 865
CONFIG_UDP_RECVMBOX_SIZE, 864, 911
CONFIG_ULP_COPROC_ENABLED, 866
CONFIG_ULP_COPROC_RESERVE_MEM, 866
CONFIG_USE_ONLY_LWIP_SELECT, 646, 863
CONFIG_WARN_WRITE_STRINGS, 855
CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST, 865, 917
CONFIG_WL_SECTOR_MODE, 860
CONFIG_WL_SECTOR_SIZE, 860

D

DAC_CHANNEL_1 (*C++ enumerator*), 376
DAC_CHANNEL_1_GPIO_NUM (*C macro*), 377
DAC_CHANNEL_2 (*C++ enumerator*), 376
DAC_CHANNEL_2_GPIO_NUM (*C macro*), 377
DAC_CHANNEL_MAX (*C++ enumerator*), 376
dac_channel_t (*C++ type*), 376
DAC_GPIO25_CHANNEL (*C macro*), 377
DAC_GPIO26_CHANNEL (*C macro*), 377
dac_i2s_disable (*C++ function*), 376
dac_i2s_enable (*C++ function*), 376
dac_output_disable (*C++ function*), 376
dac_output_enable (*C++ function*), 376
dac_output_voltage (*C++ function*), 375
dac_pad_get_io_num (*C++ function*), 375
DEFAULT_HTTP_BUF_SIZE (*C macro*), 597
dmaworkaround_cb_t (*C++ type*), 515

E

eAbortSleep (*C++ enumerator*), 692
eBlocked (*C++ enumerator*), 691
eDeleted (*C++ enumerator*), 691
eIncrement (*C++ enumerator*), 691
eNoAction (*C++ enumerator*), 691
eNoTasksWaitingTimeout (*C++ enumerator*), 692
eNotifyAction (*C++ type*), 691
environment variable
  CONFIG_A2D_INITIAL_TRACE_LEVEL, 861, 892, 898
  CONFIG_A2DP_ENABLE, 860, 891, 894
  CONFIG_A2DP_SINK_TASK_STACK_SIZE, 861, 891, 894
  CONFIG_A2DP_SOURCE_TASK_STACK_SIZE, 861, 891, 894
  CONFIG_ADC2_DISABLE_DAC, 858, 879
  CONFIG_ADC_CAL_EFUSE_TP_ENABLE, 868, 939
  CONFIG_ADC_CAL_EFUSE_VREF_ENABLE, 868, 939
  CONFIG_ADC_CAL_LUT_ENABLE, 868, 939
  CONFIG_ADC_FORCE_XPD_FSM, 858, 879
  CONFIG_APPL_INITIAL_TRACE_LEVEL, 862, 892, 900
  CONFIG_AVCT_INITIAL_TRACE_LEVEL, 861, 892, 898
  CONFIG_AVDT_INITIAL_TRACE_LEVEL, 861, 892, 898
  CONFIG_AVRC_INITIAL_TRACE_LEVEL, 861, 892, 899
  CONFIG_AWS_IOT_MQTT_HOST, 859, 881
  CONFIG_AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL, 859, 881

CONFIG_AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL, 859, 881
CONFIG_AWS_IOT_MQTT_NUM_SUBSCRIBE_HANDLERS, 859, 881
CONFIG_AWS_IOT_MQTT_PORT, 859, 881
CONFIG_AWS_IOT_MQTT_RX_BUF_LEN, 859, 881
CONFIG_AWS_IOT_MQTT_TX_BUF_LEN, 859, 881
CONFIG_AWS_IOT_OVERRIDE_THING_SHADOW_RX_BUFFER, 859, 882
CONFIG_AWS_IOT_SDK, 880
CONFIG_AWS_IOT_SHADOW_MAX_JSON_TOKEN_EXPECTED, 859, 882, 883
CONFIG_AWS_IOT_SHADOW_MAX_SHADOW_TOPIC_LENGTH, 859, 882, 883
CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_ACKS, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SIMULTANEOUS_THINGNAMES, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_RX_BUFFER, 859, 882
CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_THING_NAME, 859, 882, 883
CONFIG_AWS_IOT_SHADOW_MAX_SIZE_OF_UNIQUECLIENTID, 859, 882
CONFIG_BLE_ACTIVE_SCAN_REPORT_ADV_SCAN_RSPONSE, 862, 893, 903
CONFIG_BLE_ADV_REPORT_DISCARD_THRSHOLD, 862, 893, 903
CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_NUM, 862, 893, 903
CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_SUPPORTED, 862, 893, 903
CONFIG_BLE_ESTABLISH_LINK_CONNECTION_TIMEOUT, 862, 893, 903
CONFIG_BLE_HOST_QUEUE_CONGESTION_CHECK, 862, 893, 902
CONFIG_BLE_MESH_SCAN_DUPLICATE_EN, 862, 893, 903
CONFIG_BLE_SCAN_DUPLICATE, 862, 893, 902
CONFIG_BLE_SMP_ENABLE, 861, 891, 895
CONFIG_BLUEDROID_ENABLED, 891
CONFIG_BLUEDROID_MEM_DEBUG, 860, 891, 893
CONFIG_BLUEDROID_PINNED_TO_CORE, 860, 891
CONFIG_BLUEDROID_PINNED_TO_CORE_CHOICE, 860, 891, 893
CONFIG_BLUFI_INITIAL_TRACE_LEVEL, 862, 893, 902
CONFIG_BNEP_INITIAL_TRACE_LEVEL, 861, 892, 897
CONFIG_BOOTLOADER_APP_TEST, 847, 848, 1085
CONFIG_BOOTLOADER_DATA_FACTORY_RESET, 847, 848, 1085
CONFIG_BOOTLOADER_FACTORY_RESET, 847, 848, 1084
CONFIG_BOOTLOADER_HOLD_TIME_GPIO, 847, 848, 1085
CONFIG_BOOTLOADER_NUM_PIN_APP_TEST, 847, 848, 1085
CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET, 847, 848, 1085
CONFIG_BOOTLOADER_OTA_DATA_ERASE, 847, 848, 1085
CONFIG_BOOTLOADER_SPI_WP_PIN, 847
CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_SETUP, 847
CONFIG_BROWNOUT_DET, 866, 924, 1005
CONFIG_BROWNOUT_DET_LVL, 866
CONFIG_BROWNOUT_DET_LVL_SEL, 866, 924, 1005
CONFIG_BT_ACL_CONNECTIONS, 862, 893, 902
CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST, 862, 893, 902
CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY, 862, 893, 902
CONFIG_BT_BLE_ENABLED, 860, 889
CONFIG_BT_HCI_UART_BAUDRATE, 860, 890
CONFIG_BT_HCI_UART_NO, 860, 890
CONFIG_BT_RESERVE_DRAM, 862
CONFIG_BT_SPP_ENABLED, 861, 891, 894
CONFIG_BT_STACK_NO_LOG, 861, 892, 895
CONFIG_BTC_INITIAL_TRACE_LEVEL, 862, 893, 901
CONFIG_BTC_TASK_STACK_SIZE, 860, 891, 893
CONFIG_BTDM_CONTROLLER_AUTO_LATENCY, 860, 889
CONFIG_BTDM_CONTROLLER_AUTO_LATENCY_EFF, 860
CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH, 860, 889
CONFIG_BTDM_CONTROLLER_BR_EDR_SCO_DATA_PATH_EFF 860
CONFIG_BTDM_CONTROLLER_HCI_MODE_CHOICE, 860, 889
CONFIG_BTDM_CONTROLLER_MODEM_SLEEP, 860, 890
CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE, 860
CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE_CHOICE, 860, 889
CONFIG_BTDM_LOW_POWER_CLOCK, 860, 890, 891
CONFIG_BTDM_MODEM_SLEEP_MODE, 860, 890
CONFIG_BTIF_INITIAL_TRACE_LEVEL, 862,

892, 893, 901
CONFIG_BTM_INITIAL_TRACE_LEVEL, 861, 892, 895
CONFIG_CLASSIC_BT_ENABLED, 860, 891, 894
CONFIG_COMPATIBLE_PRE_V2_1_BOOTLOADERS, 866, 927
CONFIG_CONSOLE_UART, 865, 921
CONFIG_CONSOLE_UART_BAUDRATE, 866, 922
CONFIG_CONSOLE_UART_NUM, 865, 866, 922
CONFIG_CONSOLE_UART_RX_GPIO, 866, 922
CONFIG_CONSOLE_UART_TX_GPIO, 866, 922
CONFIG_CXX_EXCEPTIONS, 855, 999
CONFIG_CXX_EXCEPTIONS_EMG_POOL_SIZE, 854, 856, 999
CONFIG_DISABLE_BASIC_ROM_CONSOLE, 866, 927
CONFIG_DMA_RX_BUF_NUM, 868, 938
CONFIG_DMA_TX_BUF_NUM, 868, 938
CONFIG_DUPLICATE_SCAN_CACHE_SIZE, 862, 893, 902
CONFIG_EMAC_L2_TO_L3_RX_BUF_MODE, 868, 938
CONFIG_EMAC_TASK_PRIORITY, 868, 939
CONFIG_EMAC_TASK_STACK_SIZE, 868, 939
CONFIG_ENABLE_STATIC_TASK_CLEAN_UP_HOOK, 868, 936
CONFIG_ESP32_APPTRACE_DESTINATION, 864, 913
CONFIG_ESP32_APPTRACE_ENABLE, 864
CONFIG_ESP32_APPTRACE_LOCK_ENABLE, 864
CONFIG_ESP32_APPTRACE_ONPANIC_HOST_FLUSH_TMO, 864, 913, 1076
CONFIG_ESP32_APPTRACE_PENDING_DATA_SIZE_MAX, 864, 913, 1076
CONFIG_ESP32_APPTRACE_POSTMORTEM_FLUSH_TRAX_THRESH, 864, 913, 1076
CONFIG_ESP32_CORE_DUMP_LOG_LEVEL, 865, 919
CONFIG_ESP32_CORE_DUMP_UART_DELAY, 865, 919
CONFIG_ESP32_COREDUMP_TO_FLASH_OR_UART, 865, 919
CONFIG_ESP32_DEBUG_OCDAWARE, 866, 923, 1000, 1071
CONFIG_ESP32_DEBUG_STUBS_ENABLE, 866, 923
CONFIG_ESP32_DEEP_SLEEP_WAKEUP_DELAY, 866, 926
CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ, 819, 865, 916
CONFIG_ESP32_DPORT_DIS_INTERRUPT_LVL, 867, 931
CONFIG_ESP32_DPORT_WORKAROUND, 864
CONFIG_ESP32_ENABLE_COREDUMP, 865
CONFIG_ESP32_GCOV_ENABLE, 864, 915
CONFIG_ESP32_PANIC, 62, 123, 866, 922, 1000
CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE, 867, 931
CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION, 867, 931, 1088
CONFIG_ESP32_PHY_MAX_TX_POWER, 867
CONFIG_ESP32_PHY_MAX_WIFI_TX_POWER, 867, 931
CONFIG_ESP32_PTHREAD_TASK_PRIO_DEFAULT, 858, 879
CONFIG_ESP32_PTHREAD_TASK_STACK_SIZE_DEFAULT, 858, 879
CONFIG_ESP32_REV_MIN, 864, 916
CONFIG_ESP32_RTC_CLK_CAL_CYCLES, 866, 925
CONFIG_ESP32_RTC_CLOCK_SOURCE, 866, 925
CONFIG_ESP32_RTC_EXTERNAL_CRYSTAL_ADDITIONAL_CU 866, 925
CONFIG_ESP32_RTC_XTAL_BOOTSTRAP_CYCLES, 866, 926
CONFIG_ESP32_TIME_SYSCALL, 866, 925
CONFIG_ESP32_TRAX, 865, 919
CONFIG_ESP32_TRAX_TWOBANKS, 865, 919
CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED, 867, 930
CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED, 867, 929
CONFIG_ESP32_WIFI_CSI_ENABLED, 867, 929
CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM, 867, 928
CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM, 867, 929
CONFIG_ESP32_WIFI_IRAM_OPT, 867, 930
CONFIG_ESP32_WIFI_MGMT_SBUF_NUM, 867, 931
CONFIG_ESP32_WIFI_NVS_ENABLED, 867, 930
CONFIG_ESP32_WIFI_RX_BA_WIN, 867, 930
CONFIG_ESP32_WIFI_RX_IRAM_OPT, 867, 931
CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN, 867, 930
CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM, 867, 928
CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM, 867, 929
CONFIG_ESP32_WIFI_TASK_CORE_ID, 867, 930
CONFIG_ESP32_WIFI_TX_BA_WIN, 867, 929
CONFIG_ESP32_WIFI_TX_BUFFER, 867, 928

CONFIG_ESP32_WIFI_TX_BUFFER_TYPE, 867
CONFIG_ESP32_XTAL_FREQ, 866
CONFIG_ESP32_XTAL_FREQ_SEL, 48, 113, 866, 926
CONFIG_ESP_ERR_TO_NAME_LOOKUP, 866, 927, 998
CONFIG_ESP_GRATUITOUS_ARP, 863, 906
CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS, 862, 904
CONFIG_ESP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES, 863, 908, 910
CONFIG_ESP_TIMER_PROFILING, 866, 927
CONFIG_ESPTOOLPY_AFTER, 844, 846
CONFIG_ESPTOOLPY_BAUD, 844
CONFIG_ESPTOOLPY_BAUD_OTHER_VAL, 844, 845
CONFIG_ESPTOOLPY_BEFORE, 844, 846
CONFIG_ESPTOOLPY_COMPRESSED, 844, 845
CONFIG_ESPTOOLPY_FLASHFREQ, 844, 845
CONFIG_ESPTOOLPY_FLASHMODE, 844
CONFIG_ESPTOOLPY_FLASHSIZE, 599, 844, 845
CONFIG_ESPTOOLPY_FLASHSIZE_DETECT, 844, 845
CONFIG_ESPTOOLPY_PORT, 844
CONFIG_FATFS_API_ENCODING, 860, 888
CONFIG_FATFS_CHOOSE_CODEPAGE, 860, 887
CONFIG_FATFS_CODEPAGE, 860
CONFIG_FATFS_FS_LOCK, 860, 888
CONFIG_FATFS_LONG_FILENAMES, 860, 888
CONFIG_FATFS_MAX_LFN, 860, 888
CONFIG_FATFS_PER_FILE_CACHE, 860, 889
CONFIG_FATFS_TIMEOUT_MS, 860, 888
CONFIG_FLASH_ENCRYPTION_DISABLE_PLAINTEXT, 849, 853, 1016
CONFIG_FLASH_ENCRYPTION_ENABLED, 849, 852
CONFIG_FLASH_ENCRYPTION_INSECURE, 849, 852
CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_CACHE, 849, 853
CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_DECRYPT, 849, 853
CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_ENCRYPT, 849, 853
CONFIG_FLASHMODE, 844, 845
CONFIG_FREERTOS_ASSERT, 867, 934
CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION, 867, 933, 1026
CONFIG_FREERTOS_CHECK_STACKOVERFLOW, 867, 933
CONFIG_FREERTOS_CORETIMER, 867, 932
CONFIG_FREERTOS_DEBUG_INTERNALS, 938
CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS, 868, 937
CONFIG_FREERTOS_HZ, 867, 933
CONFIG_FREERTOS_IDLE_TASK_STACKSIZE, 868, 934
CONFIG_FREERTOS_IDLE_TIME_BEFORE_SLEEP, 868, 938
CONFIG_FREERTOS_INTERRUPT_BACKTRACE, 867, 934
CONFIG_FREERTOS_ISR_STACKSIZE, 868, 935
CONFIG_FREERTOS_LEGACY_HOOKS, 769, 868, 935
CONFIG_FREERTOS_LEGACY_IDLE_HOOK, 769, 868, 935
CONFIG_FREERTOS_LEGACY_TICK_HOOK, 769, 868, 935
CONFIG_FREERTOS_MAX_TASK_NAME_LEN, 868, 935
CONFIG_FREERTOS_PORTMUX_DEBUG, 868, 938
CONFIG_FREERTOS_PORTMUX_DEBUG_RECURSIVE, 868, 938
CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE, 868, 936
CONFIG_FREERTOS_RUN_TIME_STATS_CLK, 868, 937
CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS, 867, 934, 1025
CONFIG_FREERTOS_UNICORE, 46, 110, 602, 867, 932, 1025
CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS, 868, 937
CONFIG_FREERTOS_USE_TICKLESS_IDLE, 819, 868, 937
CONFIG_FREERTOS_USE_TRACE_FACILITY, 868, 937
CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK, 867, 933, 1070, 1071
CONFIG_GAP_INITIAL_TRACE_LEVEL, 861, 892, 897
CONFIG_GARP_TMR_INTERVAL, 863, 906
CONFIG_GATT_INITIAL_TRACE_LEVEL, 862, 892, 900
CONFIG_GATTC_CACHE_NVS_FLASH, 861, 891, 895
CONFIG_GATTC_ENABLE, 861, 891, 894
CONFIG_GATTS_ENABLE, 861, 891, 894
CONFIG_HCI_INITIAL_TRACE_LEVEL, 861, 892, 895
CONFIG_HEAP_CORRUPTION_DETECTION, 785, 858, 878
CONFIG_HEAP_TASK_TRACKING, 858, 879
CONFIG_HEAP_TRACING, 786, 858, 878

CONFIG_HEAP_TRACING_STACK_DEPTH, 858, 878
CONFIG_HFP_AUDIO_DATA_PATH, 861, 891, 894
CONFIG_HFP_ENABLE, 861, 891, 894
CONFIG_HFP_ROLE, 861, 891, 894
CONFIG_HID_INITIAL_TRACE_LEVEL, 862, 892, 899
CONFIG_INT_WDT, 866, 923
CONFIG_INT_WDT_CHECK_CPU1, 866, 923
CONFIG_INT_WDT_TIMEOUT_MS, 866, 923
CONFIG_IP_LOST_TIMER_INTERVAL, 860, 887
CONFIG_IPC_TASK_STACK_SIZE, 802, 865, 920
CONFIG_L2_TO_L3_COPY, 862, 904
CONFIG_L2CAP_INITIAL_TRACE_LEVEL, 861, 892, 896
CONFIG_LIBSODIUM_USE_MBEDTLS_SHA, 862, 904
CONFIG_LOG_BOOTLOADER_LEVEL, 847
CONFIG_LOG_COLORS, 859, 880
CONFIG_LOG_DEFAULT_LEVEL, 808, 859, 880
CONFIG_LWIP_AUTOIP, 907
CONFIG_LWIP_AUTOIP_MAX_CONFLICTS, 863, 907
CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL, 863, 907
CONFIG_LWIP_AUTOIP_TRIES, 863, 907
CONFIG_LWIP_BROADCAST_PING, 864, 912
CONFIG_LWIP_DHCP_DOES_ARP_CHECK, 863, 906
CONFIG_LWIP_DHCP_MAX_NTP_SERVERS, 863, 905
CONFIG_LWIP_DHCPS_LEASE_UNIT, 863, 906, 907
CONFIG_LWIP_DHCPS_MAX_STATION_NUM, 863, 907
CONFIG_LWIP_ETHARP_TRUST_IP_MAC, 863, 905
CONFIG_LWIP_IP_FRAG, 863, 905
CONFIG_LWIP_IP_REASSEMBLY, 863, 905
CONFIG_LWIP_IPV6_AUTOCONFIG, 863, 907
CONFIG_LWIP_IRAM_OPTIMIZATION, 863, 904
CONFIG_LWIP_LOOPBACK_MAX_PBUFS, 863, 908
CONFIG_LWIP_MAX_ACTIVE_TCP, 863, 908
CONFIG_LWIP_MAX_LISTENING_TCP, 863, 908
CONFIG_LWIP_MAX_RAW_PCBS, 864, 912
CONFIG_LWIP_MAX_SOCKETS, 863, 904
CONFIG_LWIP_MAX_UDP_PCBS, 864, 911
CONFIG_LWIP_MULTICAST_PING, 864, 912
CONFIG_LWIP_NETIF_LOOPBACK, 908
CONFIG_LWIP_SO_RCVBUF, 863, 905
CONFIG_LWIP_SO_REUSE, 863, 905
CONFIG_LWIP_SO_REUSE_RXTOALL, 863, 905
CONFIG_LWIP_STATS, 863, 905
CONFIG_LWIP_WND_SCALE, 863, 908, 910
CONFIG_MAIN_TASK_STACK_SIZE, 865, 920
CONFIG_MAKE_WARN_UNDEFINED_VARIABLES, 843
CONFIG_MBEDTLS_AES_C, 857, 874
CONFIG_MBEDTLS_BLOWFISH_C, 858, 874, 875
CONFIG_MBEDTLS_CAMELLIA_C, 858, 874
CONFIG_MBEDTLS_CCM_C, 858, 874, 875
CONFIG_MBEDTLS_DEBUG, 857, 870
CONFIG_MBEDTLS_DES_C, 858, 874
CONFIG_MBEDTLS_ECDH_C, 858, 876
CONFIG_MBEDTLS_ECDSA_C, 858, 876
CONFIG_MBEDTLS_ECP_C, 876
CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED, 858, 876, 878
CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED, 858, 876, 878
CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED, 858, 876, 878
CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED, 858, 876, 877
CONFIG_MBEDTLS_ECP_NIST_OPTIM, 858, 876, 878
CONFIG_MBEDTLS_GCM_C, 858, 874, 875
CONFIG_MBEDTLS_HARDWARE_AES, 857, 870
CONFIG_MBEDTLS_HARDWARE_MPI, 857, 870
CONFIG_MBEDTLS_HARDWARE_SHA, 857, 871
CONFIG_MBEDTLS_HAVE_TIME, 857, 871
CONFIG_MBEDTLS_HAVE_TIME_DATE, 857, 871
CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_PSK, 857, 871, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA,

857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA, 857, 872, 873
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_RSA, 857, 872, 873
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA, 857, 872, 873
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_PSK, 857, 871, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA, 857, 872, 873
CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_PSK, 857, 871, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_RSA, 857, 872
CONFIG_MBEDTLS_KEY_EXCHANGE_RSA_PSK, 857, 871, 872
CONFIG_MBEDTLS_MPI_USE_INTERRUPT, 857, 870
CONFIG_MBEDTLS_PEM_PARSE_C, 858, 875
CONFIG_MBEDTLS_PEM_WRITE_C, 858, 875, 876
CONFIG_MBEDTLS_PSK_MODES, 857, 871, 872
CONFIG_MBEDTLS_RC4_MODE, 858, 874
CONFIG_MBEDTLS_RIPEMD160_C, 858, 875
CONFIG_MBEDTLS_SSL_ALPN, 857, 874
CONFIG_MBEDTLS_SSL_MAX_CONTENT_LEN, 857, 870
CONFIG_MBEDTLS_SSL_PROTO_DTLS, 857, 873
CONFIG_MBEDTLS_SSL_PROTO_SSL3, 857, 873
CONFIG_MBEDTLS_SSL_PROTO_TLS1, 857, 873
CONFIG_MBEDTLS_SSL_PROTO_TLS1_1, 857, 873
CONFIG_MBEDTLS_SSL_PROTO_TLS1_2, 857, 873
CONFIG_MBEDTLS_SSL_RENEGOTIATION, 857, 873
CONFIG_MBEDTLS_SSL_SESSION_TICKETS, 857, 874
CONFIG_MBEDTLS_TLS_CLIENT, 857
CONFIG_MBEDTLS_TLS_ENABLED, 857
CONFIG_MBEDTLS_TLS_MODE, 857, 871
CONFIG_MBEDTLS_TLS_SERVER, 857
CONFIG_MBEDTLS_X509_CRL_PARSE_C, 858, 875, 876
CONFIG_MBEDTLS_X509_CSR_PARSE_C, 858, 875, 876
CONFIG_MBEDTLS_XTEA_C, 858, 874, 875
CONFIG_MCA_INITIAL_TRACE_LEVEL, 862, 892, 899
CONFIG_MEMMAP_TRACEMEM, 865
CONFIG_MEMMAP_TRACEMEM_TWOBANKS, 865
CONFIG_MESH_DUPLICATE_SCAN_CACHE_SIZE, 862, 893, 903
CONFIG_MONITOR_BAUD, 844, 846
CONFIG_MONITOR_BAUD_OTHER_VAL, 844, 846
CONFIG_MP_BLOB_SUPPORT, 859, 883
CONFIG_NEWLIB_NANO_FORMAT, 865, 921
CONFIG_NEWLIB_STDIN_LINE_ENDING, 865, 921
CONFIG_NEWLIB_STDOUT_LINE_ENDING, 865, 920
CONFIG_NO_BLOBS, 866, 927
CONFIG_NUMBER_OF_UNIVERSAL_MAC_ADDRESS, 865, 919
CONFIG_NVS_ENCRYPTION, 859, 883
CONFIG_OPENSSL_ASSERT, 857, 869
CONFIG_OPENSSL_DEBUG, 856, 869
CONFIG_OPENSSL_DEBUG_LEVEL, 856, 869
CONFIG_OPENSSL_LOWLEVEL_DEBUG, 857, 869
CONFIG_OPTIMIZATION_ASSERTION_LEVEL, 854, 855
CONFIG_OPTIMIZATION_COMPILER, 854, 855
CONFIG_OSI_INITIAL_TRACE_LEVEL, 862, 893, 901
CONFIG_PAN_INITIAL_TRACE_LEVEL, 861, 892, 897
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME, 854
CONFIG_PARTITION_TABLE_FILENAME, 854
CONFIG_PARTITION_TABLE_MD5, 854, 1089
CONFIG_PARTITION_TABLE_OFFSET, 854
CONFIG_PARTITION_TABLE_TYPE, 854
CONFIG_PM_DFS_INIT_AUTO, 819, 867, 932
CONFIG_PM_ENABLE, 819, 867, 932
CONFIG_PM_PROFILING, 867, 932
CONFIG_PM_TRACE, 867, 932
CONFIG_PM_USE_RTC_TIMER_REF, 867, 932
CONFIG_PPP_CHAP_SUPPORT, 864, 911, 912
CONFIG_PPP_DEBUG_ON, 864, 911, 912
CONFIG_PPP_MPPE_SUPPORT, 864, 911, 912
CONFIG_PPP_MSCHAP_SUPPORT, 864, 911, 912
CONFIG_PPP_PAP_SUPPORT, 864, 911, 912
CONFIG_PPP_SUPPORT, 911
CONFIG_PYTHON, 843
CONFIG_RFCOMM_INITIAL_TRACE_LEVEL, 861, 892, 896
CONFIG_SDP_INITIAL_TRACE_LEVEL, 861, 892, 896
CONFIG_SECURE_BOOT_ALLOW_JTAG, 849, 852

CONFIG_SECURE_BOOT_ALLOW_ROM_BASIC, 849, 852
CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION, 849, 852
CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES, 849, 850
CONFIG_SECURE_BOOT_ENABLED, 849, 850, 1092
CONFIG_SECURE_BOOT_INSECURE, 849, 851
CONFIG_SECURE_BOOT_SIGNING_KEY, 849, 851
CONFIG_SECURE_BOOT_TEST_MODE, 849, 853
CONFIG_SECURE_BOOT_VERIFICATION_KEY, 849, 851
CONFIG_SECURE_BOOTLOADER_KEY_ENCODING, 849, 851, 1092
CONFIG_SECURE_BOOTLOADER_MODE, 849, 850, 1092
CONFIG_SECURE_SIGNED_APPS, 849
CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT, 849
CONFIG_SECURE_SIGNED_ON_BOOT, 849
CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT, 849
CONFIG_SECURE_SIGNED_ON_UPDATE, 849
CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT, 849, 850
CONFIG_SMP_ENABLE, 862, 893
CONFIG_SMP_INITIAL_TRACE_LEVEL, 862, 892, 900
CONFIG_SMP_SLAVE_CON_PARAMS_UPD_ENABLE, 861, 891, 895
CONFIG_SPI_FLASH_ENABLE_COUNTERS, 868, 940
CONFIG_SPI_FLASH_ERASE_YIELD_DURATION_MS, 868, 941
CONFIG_SPI_FLASH_ERASE_YIELD_TICKS, 869, 941
CONFIG_SPI_FLASH_LOG_FAILED_WRITE, 868, 940
CONFIG_SPI_FLASH_ROM_DRIVER_PATCH, 868, 940
CONFIG_SPI_FLASH_VERIFY_WRITE, 868, 939
CONFIG_SPI_FLASH_WARN_SETTING_ZERO_TO_ONE, 868, 940
CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS, 868, 940
CONFIG_SPI_FLASH_YIELD_DURING_ERASE, 868, 941
CONFIG_SPI_MASTER_IN_IRAM, 506, 858, 880
CONFIG_SPI_MASTER_ISR_IN_IRAM, 506, 859, 880
CONFIG_SPIFFS_API_DBG, 859, 885
CONFIG_SPIFFS_CACHE, 859, 883
CONFIG_SPIFFS_CACHE_DBG, 860, 885
CONFIG_SPIFFS_CACHE_STATS, 859, 883, 884
CONFIG_SPIFFS_CACHE_WR, 859, 883, 884
CONFIG_SPIFFS_CHECK_DBG, 860, 885, 886
CONFIG_SPIFFS_DBG, 859, 885
CONFIG_SPIFFS_GC_DBG, 860, 885
CONFIG_SPIFFS_GC_MAX_RUNS, 859, 884
CONFIG_SPIFFS_GC_STATS, 859, 884
CONFIG_SPIFFS_MAX_PARTITIONS, 859, 883
CONFIG_SPIFFS_META_LENGTH, 859, 885
CONFIG_SPIFFS_OBJ_NAME_LEN, 859, 884
CONFIG_SPIFFS_PAGE_CHECK, 859, 884
CONFIG_SPIFFS_PAGE_SIZE, 859, 884
CONFIG_SPIFFS_TEST_VISUALISATION, 860, 885, 886
CONFIG_SPIFFS_USE_MAGIC, 859, 884
CONFIG_SPIFFS_USE_MAGIC_LENGTH, 859, 885
CONFIG_SPIFFS_USE_MTIME, 859, 885
CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY, 865, 917, 918, 1190
CONFIG_SPIRAM_BOOT_INIT, 865, 916, 917
CONFIG_SPIRAM_CACHE_WORKAROUND, 865, 916, 918, 975, 991, 1191
CONFIG_SPIRAM_IGNORE_NOTFOUND, 865, 916, 917
CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL, 865, 916, 918
CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL, 865, 917, 918
CONFIG_SPIRAM_MEMTEST, 865, 916, 918
CONFIG_SPIRAM_SIZE, 865, 916
CONFIG_SPIRAM_SPEED, 865, 916, 917
CONFIG_SPIRAM_SUPPORT, 865, 916
CONFIG_SPIRAM_TYPE, 865, 916, 917
CONFIG_SPIRAM_USE, 865, 916, 917
CONFIG_STACK_CHECK, 854
CONFIG_STACK_CHECK_MODE, 854, 856, 1005
CONFIG_SUPPORT_STATIC_ALLOCATION, 868, 935, 1020, 1025
CONFIG_SUPPRESS_SELECT_DEBUG_OUTPUT, 857, 870
CONFIG_SW_COEXIST_ENABLE, 866, 928
CONFIG_SW_COEXIST_PREFERENCE, 866, 928
CONFIG_SW_COEXIST_PREFERENCE_VALUE, 866
CONFIG_SYSTEM_EVENT_QUEUE_SIZE, 865, 920
CONFIG_SYSTEM_EVENT_TASK_STACK_SIZE, 865, 920
CONFIG_SYSVIEW_ENABLE, 864, 913, 914, 1082
CONFIG_SYSVIEW_EVT_IDLE_ENABLE, 864, 914, 915

CONFIG_SYSVIEW_EVT_ISR_ENTER_ENABLE, 864, 913, 914
CONFIG_SYSVIEW_EVT_ISR_EXIT_ENABLE, 864, 913, 914
CONFIG_SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE, 864, 913, 914
CONFIG_SYSVIEW_EVT_OVERFLOW_ENABLE, 864, 913, 914
CONFIG_SYSVIEW_EVT_TASK_CREATE_ENABLE, 864, 914, 915
CONFIG_SYSVIEW_EVT_TASK_START_EXEC_ENABLE, 864, 913, 915
CONFIG_SYSVIEW_EVT_TASK_START_READY_ENABLE, 864, 914, 915
CONFIG_SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE, 864, 913, 915
CONFIG_SYSVIEW_EVT_TASK_STOP_READY_ENABLE, 864, 914, 915
CONFIG_SYSVIEW_EVT_TASK_TERMINATE_ENABLE, 864, 914, 915
CONFIG_SYSVIEW_EVT_TIMER_ENTER_ENABLE, 864, 914, 915
CONFIG_SYSVIEW_EVT_TIMER_EXIT_ENABLE, 864, 914, 915
CONFIG_SYSVIEW_TS_SOURCE, 864, 913, 914, 1082
CONFIG_TASK_WDT, 799, 866, 923
CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU0, 799, 866, 924
CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU1, 799, 866, 924
CONFIG_TASK_WDT_PANIC, 866, 924, 999
CONFIG_TASK_WDT_TIMEOUT_S, 866, 924
CONFIG_TCP_MAXRTX, 863, 908, 909
CONFIG_TCP_MSL, 863, 908, 909
CONFIG_TCP_MSS, 863, 908, 909
CONFIG_TCP_OVERSIZE, 863, 908, 910
CONFIG_TCP_QUEUE_OOSEQ, 863, 908, 910
CONFIG_TCP_RCV_SCALE, 863, 908, 910
CONFIG_TCP_RECVMBOX_SIZE, 863, 908, 909
CONFIG_TCP_SND_BUF_DEFAULT, 863, 908, 909
CONFIG_TCP_SYNMAXRTX, 863, 908, 909
CONFIG_TCP_WND_DEFAULT, 863, 908, 909
CONFIG_TCPIP_RECVMBOX_SIZE, 863, 906
CONFIG_TCPIP_TASK_STACK_SIZE, 864, 911
CONFIG_TIMER_QUEUE_LENGTH, 868, 936
CONFIG_TIMER_TASK_PRIORITY, 868, 936
CONFIG_TIMER_TASK_STACK_DEPTH, 868, 936
CONFIG_TIMER_TASK_STACK_SIZE, 865, 920
CONFIG_TOOLPREFIX, 843
CONFIG_TRACEMEM_RESERVE_DRAM, 865
CONFIG_UDP_RECVMBOX_SIZE, 864, 911
CONFIG_ULP_COPROC_ENABLED, 866, 922
CONFIG_ULP_COPROC_RESERVE_MEM, 866, 922
CONFIG_USE_ONLY_LWIP_SELECT, 646, 863, 904
CONFIG_WARN_WRITE_STRINGS, 855, 856
CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST, 865, 917, 918
CONFIG_WL_SECTOR_MODE, 860, 886
CONFIG_WL_SECTOR_SIZE, 860, 886
eReady (*C++ enumerator*), 691
eRunning (*C++ enumerator*), 691
eSuspended (*C++ enumerator*), 691
eSetValueWithoutOverwrite (*C++ enumerator*), 691
eSetValueWithOverwrite (*C++ enumerator*), 691
eWakeUpModeStatus (*C++ type*), 692
ESP_A2D_AUDIO_CFG_EVT (*C++ enumerator*), 315
ESP_A2D_AUDIO_STATE_EVT (*C++ enumerator*), 315
ESP_A2D_AUDIO_STATE_REMOTE_SUSPEND (*C++ enumerator*), 314
ESP_A2D_AUDIO_STATE_STARTED (*C++ enumerator*), 314
ESP_A2D_AUDIO_STATE_STOPPED (*C++ enumerator*), 314
esp_a2d_audio_state_t (*C++ type*), 314
esp_a2d_cb_event_t (*C++ type*), 315
esp_a2d_cb_param_t (*C++ type*), 311
esp_a2d_cb_param_t::a2d_audio_cfg_param (*C++ class*), 312
esp_a2d_cb_param_t::a2d_audio_cfg_param::mcc (*C++ member*), 312
esp_a2d_cb_param_t::a2d_audio_cfg_param::remote_bda (*C++ member*), 312
esp_a2d_cb_param_t::a2d_audio_stat_param (*C++ class*), 312
esp_a2d_cb_param_t::a2d_audio_stat_param::remote_bd (*C++ member*), 312
esp_a2d_cb_param_t::a2d_audio_stat_param::state (*C++ member*), 312
esp_a2d_cb_param_t::a2d_conn_stat_param (*C++ class*), 312
esp_a2d_cb_param_t::a2d_conn_stat_param::disc_rsn (*C++ member*), 312
esp_a2d_cb_param_t::a2d_conn_stat_param::remote_bda (*C++ member*), 312
esp_a2d_cb_param_t::a2d_conn_stat_param::state (*C++ member*), 312
esp_a2d_cb_param_t::audio_cfg (*C++ member*), 312
esp_a2d_cb_param_t::audio_stat (*C++ member*), 311
esp_a2d_cb_param_t::conn_stat (*C++ mem-*

ber), 311
esp_a2d_cb_param_t::media_ctrl_stat (C++ member), 312
esp_a2d_cb_param_t::media_ctrl_stat_param (C++ class), 312
esp_a2d_cb_param_t::media_ctrl_stat_param::cmd (C++ member), 312
esp_a2d_cb_param_t::media_ctrl_stat_param::status (C++ member), 312
esp_a2d_cb_t (C++ type), 313
ESP_A2D_CIE_LEN_ATRAC (C macro), 313
ESP_A2D_CIE_LEN_M12 (C macro), 313
ESP_A2D_CIE_LEN_M24 (C macro), 313
ESP_A2D_CIE_LEN_SBC (C macro), 313
ESP_A2D_CONNECTION_STATE_CONNECTED (C++ enumerator), 314
ESP_A2D_CONNECTION_STATE_CONNECTING (C++ enumerator), 314
ESP_A2D_CONNECTION_STATE_DISCONNECTED (C++ enumerator), 314
ESP_A2D_CONNECTION_STATE_DISCONNECTING (C++ enumerator), 314
ESP_A2D_CONNECTION_STATE_EVT (C++ enumerator), 315
esp_a2d_connection_state_t (C++ type), 314
ESP_A2D_DISC_RSN_ABNORMAL (C++ enumerator), 314
ESP_A2D_DISC_RSN_NORMAL (C++ enumerator), 314
esp_a2d_disc_rsn_t (C++ type), 314
esp_a2d_mcc_t (C++ class), 313
esp_a2d_mcc_t::cie (C++ member), 313
esp_a2d_mcc_t::type (C++ member), 313
ESP_A2D_MCT_ATRAC (C macro), 313
ESP_A2D_MCT_M12 (C macro), 313
ESP_A2D_MCT_M24 (C macro), 313
ESP_A2D_MCT_NON_A2DP (C macro), 313
ESP_A2D_MCT_SBC (C macro), 313
esp_a2d_mct_t (C++ type), 313
esp_a2d_media_ctrl (C++ function), 310
ESP_A2D_MEDIA_CTRL_ACK_BUSY (C++ enumerator), 315
ESP_A2D_MEDIA_CTRL_ACK_EVT (C++ enumerator), 315
ESP_A2D_MEDIA_CTRL_ACK_FAILURE (C++ enumerator), 315
ESP_A2D_MEDIA_CTRL_ACK_SUCCESS (C++ enumerator), 315
esp_a2d_media_ctrl_ack_t (C++ type), 314
ESP_A2D_MEDIA_CTRL_CHECK_SRC_RDY (C++ enumerator), 315
ESP_A2D_MEDIA_CTRL_NONE (C++ enumerator), 315
ESP_A2D_MEDIA_CTRL_START (C++ enumerator), 315
ESP_A2D_MEDIA_CTRL_STOP (C++ enumerator), 315
ESP_A2D_MEDIA_CTRL_SUSPEND (C++ enumerator), 315
esp_a2d_media_ctrl_t (C++ type), 315
esp_a2d_register_callback (C++ function), 309
esp_a2d_sink_connect (C++ function), 309
esp_a2d_sink_data_cb_t (C++ type), 313
esp_a2d_sink_deinit (C++ function), 309
esp_a2d_sink_disconnect (C++ function), 310
esp_a2d_sink_init (C++ function), 309
esp_a2d_sink_register_data_callback (C++ function), 309
esp_a2d_source_connect (C++ function), 311
esp_a2d_source_data_cb_t (C++ type), 314
esp_a2d_source_deinit (C++ function), 310
esp_a2d_source_disconnect (C++ function), 311
esp_a2d_source_init (C++ function), 310
esp_a2d_source_register_data_callback (C++ function), 311
esp_adc_cal_characteristics_t (C++ class), 372
esp_adc_cal_characteristics_t::adc_num (C++ member), 373
esp_adc_cal_characteristics_t::atten (C++ member), 373
esp_adc_cal_characteristics_t::bit_width (C++ member), 373
esp_adc_cal_characteristics_t::coeff_a (C++ member), 373
esp_adc_cal_characteristics_t::coeff_b (C++ member), 373
esp_adc_cal_characteristics_t::high_curve (C++ member), 373
esp_adc_cal_characteristics_t::low_curve (C++ member), 373
esp_adc_cal_characteristics_t::vref (C++ member), 373
esp_adc_cal_characterize (C++ function), 371
esp_adc_cal_check_efuse (C++ function), 371
esp_adc_cal_get_voltage (C++ function), 372
esp_adc_cal_raw_to_voltage (C++ function), 372
ESP_ADC_CAL_VAL_DEFAULT_VREF (C++ enumerator), 373
ESP_ADC_CAL_VAL_EFUSE_TP (C++ enumerator), 373
ESP_ADC_CAL_VAL_EFUSE_VREF (C++ enumerator), 373
esp_adc_cal_value_t (C++ type), 373
ESP_APP_ID_MAX (C macro), 210

ESP_APP_ID_MIN (*C macro*), 210
esp_apptrace_buffer_get (*C++ function*), 814
esp_apptrace_buffer_put (*C++ function*), 814
esp_apptrace_dest_t (*C++ type*), 818
ESP_APPTRACE_DEST_TRAX (*C++ enumerator*), 818
ESP_APPTRACE_DEST_UART0 (*C++ enumerator*), 818
esp_apptrace_down_buffer_config (*C++ function*), 814
esp_apptrace_down_buffer_get (*C++ function*), 816
esp_apptrace_down_buffer_put (*C++ function*), 816
esp_apptrace_fclose (*C++ function*), 817
esp_apptrace_flush (*C++ function*), 815
esp_apptrace_flush_nolock (*C++ function*), 815
esp_apptrace_fopen (*C++ function*), 816
esp_apptrace_fread (*C++ function*), 817
esp_apptrace_fseek (*C++ function*), 817
esp_apptrace_fstop (*C++ function*), 818
esp_apptrace_ftell (*C++ function*), 817
esp_apptrace_fwrite (*C++ function*), 817
esp_apptrace_host_is_connected (*C++ function*), 816
esp_apptrace_init (*C++ function*), 814
esp_apptrace_read (*C++ function*), 815
esp_apptrace_vprintf (*C++ function*), 815
esp_apptrace_vprintf_to (*C++ function*), 815
esp_apptrace_write (*C++ function*), 814
esp_attr_control_t (*C++ class*), 242
esp_attr_control_t::auto_rsp (*C++ member*), 242
esp_attr_desc_t (*C++ class*), 242
esp_attr_desc_t::length (*C++ member*), 242
esp_attr_desc_t::max_length (*C++ member*), 242
esp_attr_desc_t::perm (*C++ member*), 242
esp_attr_desc_t::uuid_length (*C++ member*), 242
esp_attr_desc_t::uuid_p (*C++ member*), 242
esp_attr_desc_t::value (*C++ member*), 242
esp_attr_value_t (*C++ class*), 243
esp_attr_value_t::attr_len (*C++ member*), 243
esp_attr_value_t::attr_max_len (*C++ member*), 243
esp_attr_value_t::attr_value (*C++ member*), 243
esp_avrc_ct_cb_event_t (*C++ type*), 320
esp_avrc_ct_cb_param_t (*C++ type*), 318
esp_avrc_ct_cb_param_t::avrc_ct_change_notify_para (*C++ class*), 318
esp_avrc_ct_cb_param_t::avrc_ct_change_notify_para (*C++ member*), 318
esp_avrc_ct_cb_param_t::avrc_ct_change_notify_para (*C++ member*), 318
esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param (*C++ class*), 318
esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param::co (*C++ member*), 318
esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param::re (*C++ member*), 318
esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (*C++ class*), 318
esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param::att (*C++ member*), 319
esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param::att (*C++ member*), 319
esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param::att (*C++ member*), 319
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (*C++ class*), 319
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param::key (*C++ member*), 319
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param::key (*C++ member*), 319
esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param::tl (*C++ member*), 319
esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param (*C++ class*), 319
esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param::fe (*C++ member*), 319
esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param::re (*C++ member*), 319
esp_avrc_ct_cb_param_t::change_ntf (*C++ member*), 318
esp_avrc_ct_cb_param_t::conn_stat (*C++ member*), 318
esp_avrc_ct_cb_param_t::meta_rsp (*C++ member*), 318
esp_avrc_ct_cb_param_t::psth_rsp (*C++ member*), 318
esp_avrc_ct_cb_param_t::rmt_feats (*C++ member*), 318
esp_avrc_ct_cb_t (*C++ type*), 319
ESP_AVRC_CT_CHANGE_NOTIFY_EVT (*C++ enumerator*), 321
ESP_AVRC_CT_CONNECTION_STATE_EVT (*C++ enumerator*), 321
esp_avrc_ct_deinit (*C++ function*), 316
esp_avrc_ct_init (*C++ function*), 316
ESP_AVRC_CT_METADATA_RSP_EVT (*C++ enumerator*), 321
ESP_AVRC_CT_PASSTHROUGH_RSP_EVT (*C++ enumerator*), 321
ESP_AVRC_CT_PLAY_STATUS_RSP_EVT (*C++

enumerator), 321
esp_avrc_ct_register_callback (*C++ function*), 316
ESP_AVRC_CT_REMOTE_FEATURES_EVT (*C++ enumerator*), 321
esp_avrc_ct_send_metadata_cmd (*C++ function*), 317
esp_avrc_ct_send_passthrough_cmd (*C++ function*), 317
esp_avrc_ct_send_register_notification_cmd (*C++ function*), 317
esp_avrc_ct_send_set_player_value_cmd (*C++ function*), 316
ESP_AVRC_FEAT_ADV_CTRL (*C++ enumerator*), 320
ESP_AVRC_FEAT_BROWSE (*C++ enumerator*), 320
ESP_AVRC_FEAT_META_DATA (*C++ enumerator*), 320
ESP_AVRC_FEAT_RCCT (*C++ enumerator*), 320
ESP_AVRC_FEAT_RCTG (*C++ enumerator*), 320
ESP_AVRC_FEAT_VENDOR (*C++ enumerator*), 320
esp_avrc_features_t (*C++ type*), 320
ESP_AVRC_MD_ATTR_ALBUM (*C++ enumerator*), 321
ESP_AVRC_MD_ATTR_ARTIST (*C++ enumerator*), 321
ESP_AVRC_MD_ATTR_GENRE (*C++ enumerator*), 321
esp_avrc_md_attr_mask_t (*C++ type*), 321
ESP_AVRC_MD_ATTR_NUM_TRACKS (*C++ enumerator*), 321
ESP_AVRC_MD_ATTR_PLAYING_TIME (*C++ enumerator*), 321
ESP_AVRC_MD_ATTR_TITLE (*C++ enumerator*), 321
ESP_AVRC_MD_ATTR_TRACK_NUM (*C++ enumerator*), 321
esp_avrc_ps_attr_ids_t (*C++ type*), 322
esp_avrc_ps_eq_value_ids_t (*C++ type*), 322
ESP_AVRC_PS_EQUALIZER (*C++ enumerator*), 322
ESP_AVRC_PS_EQUALIZER_OFF (*C++ enumerator*), 322
ESP_AVRC_PS_EQUALIZER_ON (*C++ enumerator*), 322
ESP_AVRC_PS_MAX_ATTR (*C++ enumerator*), 322
ESP_AVRC_PS_REPEAT_GROUP (*C++ enumerator*), 322
ESP_AVRC_PS_REPEAT_MODE (*C++ enumerator*), 322
ESP_AVRC_PS_REPEAT_OFF (*C++ enumerator*), 322
ESP_AVRC_PS_REPEAT_SINGLE (*C++ enumerator*), 322
esp_avrc_ps_rpt_value_ids_t (*C++ type*), 322
ESP_AVRC_PS_SCAN_ALL (*C++ enumerator*), 323
ESP_AVRC_PS_SCAN_GROUP (*C++ enumerator*), 323
ESP_AVRC_PS_SCAN_MODE (*C++ enumerator*), 322
ESP_AVRC_PS_SCAN_OFF (*C++ enumerator*), 323
esp_avrc_ps_scn_value_ids_t (*C++ type*), 322
esp_avrc_ps_shf_value_ids_t (*C++ type*), 322
ESP_AVRC_PS_SHUFFLE_ALL (*C++ enumerator*), 322
ESP_AVRC_PS_SHUFFLE_GROUP (*C++ enumerator*), 322
ESP_AVRC_PS_SHUFFLE_MODE (*C++ enumerator*), 322
ESP_AVRC_PS_SHUFFLE_OFF (*C++ enumerator*), 322
ESP_AVRC_PT_CMD_BACKWARD (*C++ enumerator*), 320
ESP_AVRC_PT_CMD_FAST_FORWARD (*C++ enumerator*), 320
ESP_AVRC_PT_CMD_FORWARD (*C++ enumerator*), 320
ESP_AVRC_PT_CMD_PAUSE (*C++ enumerator*), 320
ESP_AVRC_PT_CMD_PLAY (*C++ enumerator*), 320
ESP_AVRC_PT_CMD_REWIND (*C++ enumerator*), 320
ESP_AVRC_PT_CMD_STATE_PRESSED (*C++ enumerator*), 320
ESP_AVRC_PT_CMD_STATE_RELEASED (*C++ enumerator*), 320
esp_avrc_pt_cmd_state_t (*C++ type*), 320
ESP_AVRC_PT_CMD_STOP (*C++ enumerator*), 320
esp_avrc_pt_cmd_t (*C++ type*), 320
ESP_AVRC_RN_APP_SETTING_CHANGE (*C++ enumerator*), 322
ESP_AVRC_RN_BATTERY_STATUS_CHANGE (*C++ enumerator*), 321
esp_avrc_rn_event_ids_t (*C++ type*), 321
ESP_AVRC_RN_MAX_EVT (*C++ enumerator*), 322
ESP_AVRC_RN_PLAY_POS_CHANGED (*C++ enumerator*), 321
ESP_AVRC_RN_PLAY_STATUS_CHANGE (*C++ enumerator*), 321
ESP_AVRC_RN_SYSTEM_STATUS_CHANGE (*C++ enumerator*), 322
ESP_AVRC_RN_TRACK_CHANGE (*C++ enumerator*), 321
ESP_AVRC_RN_TRACK_REACHED_END (*C++ enumerator*), 321
ESP_AVRC_RN_TRACK_REACHED_START (*C++ enumerator*), 321
esp_base_mac_addr_set (*C++ function*), 832
ESP_BD_ADDR_HEX (*C macro*), 210
ESP_BD_ADDR_LEN (*C macro*), 210
ESP_BD_ADDR_STR (*C macro*), 210
esp_bd_addr_t (*C++ type*), 210
ESP_BLE_AD_MANUFACTURER_SPECIFIC_TYPE (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_128SERVICE_DATA (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_128SOL_SRV_UUID (*C++ enumerator*), 237

ESP_BLE_AD_TYPE_128SRV_CMPL (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_128SRV_PART (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_16SRV_CMPL (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_16SRV_PART (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_32SERVICE_DATA (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_32SOL_SRV_UUID (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_32SRV_CMPL (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_32SRV_PART (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_ADV_INT (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_APPEARANCE (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_CHAN_MAP_UPDATE (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_DEV_CLASS (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_FLAG (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_INDOOR_POSITION (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_INT_RANGE (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_LE_DEV_ADDR (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_LE_ROLE (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_LE_SECURE_CONFIRM (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_LE_SECURE_RANDOM (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_LE_SUPPORT_FEATURE (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_NAME_CMPL (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_NAME_SHORT (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_PUBLIC_TARGET (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_RANDOM_TARGET (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_SERVICE_DATA (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_SM_OOB_FLAG (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_SM_TK (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_SOL_SRV_UUID (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_SPAIR_C256 (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_SPAIR_R256 (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_TRANS_DISC_DATA (*C++ enumerator*), 238
ESP_BLE_AD_TYPE_TX_PWR (*C++ enumerator*), 237
ESP_BLE_AD_TYPE_URI (*C++ enumerator*), 238
esp_ble_addr_type_t (*C++ type*), 211
esp_ble_adv_channel_t (*C++ type*), 238
ESP_BLE_ADV_DATA_LEN_MAX (*C macro*), 235
esp_ble_adv_data_t (*C++ class*), 227
esp_ble_adv_data_t::appearance (*C++ member*), 228
esp_ble_adv_data_t::flag (*C++ member*), 228
esp_ble_adv_data_t::include_name (*C++ member*), 228
esp_ble_adv_data_t::include_txpower (*C++ member*), 228
esp_ble_adv_data_t::manufacturer_len (*C++ member*), 228
esp_ble_adv_data_t::max_interval (*C++ member*), 228
esp_ble_adv_data_t::min_interval (*C++ member*), 228
esp_ble_adv_data_t::p_manufacturer_data (*C++ member*), 228
esp_ble_adv_data_t::p_service_data (*C++ member*), 228
esp_ble_adv_data_t::p_service_uuid (*C++ member*), 228
esp_ble_adv_data_t::service_data_len (*C++ member*), 228
esp_ble_adv_data_t::service_uuid_len (*C++ member*), 228
esp_ble_adv_data_t::set_scan_rsp (*C++ member*), 228
esp_ble_adv_data_type (*C++ type*), 237
esp_ble_adv_filter_t (*C++ type*), 238
ESP_BLE_ADV_FLAG_BREDR_NOT_SPT (*C macro*), 233
ESP_BLE_ADV_FLAG_DMT_CONTROLLER_SPT (*C macro*), 233
ESP_BLE_ADV_FLAG_DMT_HOST_SPT (*C macro*), 233
ESP_BLE_ADV_FLAG_GEN_DISC (*C macro*), 233
ESP_BLE_ADV_FLAG_LIMIT_DISC (*C macro*), 233
ESP_BLE_ADV_FLAG_NON_LIMIT_DISC (*C macro*), 233
esp_ble_adv_params_t (*C++ class*), 227
esp_ble_adv_params_t::adv_filter_policy (*C++ member*), 227
esp_ble_adv_params_t::adv_int_max (*C++ member*), 227

esp_ble_adv_params_t::adv_int_min (*C++ member*), 227
esp_ble_adv_params_t::adv_type (*C++ member*), 227
esp_ble_adv_params_t::channel_map (*C++ member*), 227
esp_ble_adv_params_t::own_addr_type (*C++ member*), 227
esp_ble_adv_params_t::peer_addr (*C++ member*), 227
esp_ble_adv_params_t::peer_addr_type (*C++ member*), 227
esp_ble_adv_type_t (*C++ type*), 238
ESP_BLE_APPEARANCE_BLOOD_PRESSURE_ARM (*C macro*), 234
ESP_BLE_APPEARANCE_BLOOD_PRESSURE_WRIST (*C macro*), 234
ESP_BLE_APPEARANCE_CYCLING_CADENCE (*C macro*), 234
ESP_BLE_APPEARANCE_CYCLING_COMPUTER (*C macro*), 234
ESP_BLE_APPEARANCE_CYCLING_POWER (*C macro*), 234
ESP_BLE_APPEARANCE_CYCLING_SPEED (*C macro*), 234
ESP_BLE_APPEARANCE_CYCLING_SPEED_CADENCE (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_BARCODE_SCANNER (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_BLOOD_PRESSURE (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_CLOCK (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_COMPUTER (*C macro*), 233
ESP_BLE_APPEARANCE_GENERIC_CONTINUOUS_GLUCOSE_MONITOR (*C macro*), 235
ESP_BLE_APPEARANCE_GENERIC_CYCLING (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_DISPLAY (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_EYEGLASSES (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_GLUCOSE (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_HEART_RATE (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_HID (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_INSULIN_PUMP (*C macro*), 235
ESP_BLE_APPEARANCE_GENERIC_KEYRING (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_MEDIA_PLAYER (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_MEDICATION_DELIVERY (*C macro*), 235
ESP_BLE_APPEARANCE_GENERIC_OUTDOOR_SPORTS (*C macro*), 235
ESP_BLE_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVICE (*C macro*), 235
ESP_BLE_APPEARANCE_GENERIC_PHONE (*C macro*), 233
ESP_BLE_APPEARANCE_GENERIC_PULSE_OXIMETER (*C macro*), 235
ESP_BLE_APPEARANCE_GENERIC_REMOTE (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_TAG (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_THERMOMETER (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_WALKING (*C macro*), 234
ESP_BLE_APPEARANCE_GENERIC_WATCH (*C macro*), 233
ESP_BLE_APPEARANCE_GENERIC_WEIGHT (*C macro*), 235
ESP_BLE_APPEARANCE_HEART_RATE_BELT (*C macro*), 234
ESP_BLE_APPEARANCE_HID_BARCODE_SCANNER (*C macro*), 234
ESP_BLE_APPEARANCE_HID_CARD_READER (*C macro*), 234
ESP_BLE_APPEARANCE_HID_DIGITAL_PEN (*C macro*), 234
ESP_BLE_APPEARANCE_HID_DIGITIZER_TABLET (*C macro*), 234
ESP_BLE_APPEARANCE_HID_GAMEPAD (*C macro*), 234
ESP_BLE_APPEARANCE_HID_JOYSTICK (*C macro*), 234
ESP_BLE_APPEARANCE_HID_KEYBOARD (*C macro*), 234
ESP_BLE_APPEARANCE_HID_MOUSE (*C macro*), 234
ESP_BLE_APPEARANCE_INSULIN_PEN (*C macro*), 235
ESP_BLE_APPEARANCE_INSULIN_PUMP_DURABLE_PUMP (*C macro*), 235
ESP_BLE_APPEARANCE_INSULIN_PUMP_PATCH_PUMP (*C macro*), 235
ESP_BLE_APPEARANCE_MOBILITY_SCOOTER (*C macro*), 235
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION (*C macro*), 235
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_AND_NAV (*C macro*), 235
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD (*C macro*), 235

ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION (C macro), 235
ESP_BLE_APPEARANCE_POWERED_WHEELCHAIR (C macro), 235
ESP_BLE_APPEARANCE_PULSE_OXIMETER_FINGERTIP (C macro), 235
ESP_BLE_APPEARANCE_PULSE_OXIMETER_WRIST (C macro), 235
ESP_BLE_APPEARANCE_SPORTS_WATCH (C macro), 234
ESP_BLE_APPEARANCE_THERMOMETER_EAR (C macro), 234
ESP_BLE_APPEARANCE_UNKNOWN (C macro), 233
ESP_BLE_APPEARANCE_WALKING_IN_SHOE (C macro), 234
ESP_BLE_APPEARANCE_WALKING_ON_HIP (C macro), 234
ESP_BLE_APPEARANCE_WALKING_ON_SHOE (C macro), 234
esp_ble_auth_cmpl_t (C++ class), 232
esp_ble_auth_cmpl_t::addr_type (C++ member), 232
esp_ble_auth_cmpl_t::auth_mode (C++ member), 232
esp_ble_auth_cmpl_t::bd_addr (C++ member), 232
esp_ble_auth_cmpl_t::dev_type (C++ member), 232
esp_ble_auth_cmpl_t::fail_reason (C++ member), 232
esp_ble_auth_cmpl_t::key (C++ member), 232
esp_ble_auth_cmpl_t::key_present (C++ member), 232
esp_ble_auth_cmpl_t::key_type (C++ member), 232
esp_ble_auth_cmpl_t::success (C++ member), 232
esp_ble_auth_req_t (C++ type), 235
esp_ble_bond_dev_t (C++ class), 231
esp_ble_bond_dev_t::bd_addr (C++ member), 231
esp_ble_bond_dev_t::bond_key (C++ member), 231
esp_ble_bond_key_info_t (C++ class), 231
esp_ble_bond_key_info_t::key_mask (C++ member), 231
esp_ble_bond_key_info_t::pcsrk_key (C++ member), 231
esp_ble_bond_key_info_t::penc_key (C++ member), 231
esp_ble_bond_key_info_t::pid_key (C++ member), 231
esp_ble_confirm_reply (C++ function), 220
ESP_BLE_CONN_INT_MAX (C macro), 209
ESP_BLE_CONN_INTERVAL_MIN (C macro), 209
ESP_BLE_CONN_LATENCY_MAX (C macro), 209
ESP_BLE_CONN_PARAM_UNDEF (C macro), 210
ESP_BLE_CONN_SUP_TOUT_MAX (C macro), 210
ESP_BLE_CONN_SUP_TOUT_MIN (C macro), 209
esp_ble_conn_update_params_t (C++ class), 229
esp_ble_conn_update_params_t::bda (C++ member), 229
esp_ble_conn_update_params_t::latency (C++ member), 229
esp_ble_conn_update_params_t::max_int (C++ member), 229
esp_ble_conn_update_params_t::min_int (C++ member), 229
esp_ble_conn_update_params_t::timeout (C++ member), 229
ESP_BLE_CSR_KEY_MASK (C macro), 210
ESP_BLE_ENC_KEY_MASK (C macro), 210
ESP_BLE_EVT_CONN_ADV (C++ enumerator), 240
ESP_BLE_EVT_CONN_DIR_ADV (C++ enumerator), 240
ESP_BLE_EVT_DISC_ADV (C++ enumerator), 240
ESP_BLE_EVT_NON_CONN_ADV (C++ enumerator), 240
ESP_BLE_EVT_SCAN_RSP (C++ enumerator), 241
esp_ble_evt_type_t (C++ type), 240
esp_ble_gap_cb_param_t (C++ type), 222
esp_ble_gap_cb_param_t::adv_data_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::adv_data_raw_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::adv_start_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::adv_stop_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::ble_adv_data_cmpl_evt_para (C++ class), 223
esp_ble_gap_cb_param_t::ble_adv_data_cmpl_evt_para (C++ member), 223
esp_ble_gap_cb_param_t::ble_adv_data_raw_cmpl_evt_p (C++ class), 223
esp_ble_gap_cb_param_t::ble_adv_data_raw_cmpl_evt_p (C++ member), 223
esp_ble_gap_cb_param_t::ble_adv_start_cmpl_evt_para (C++ class), 223
esp_ble_gap_cb_param_t::ble_adv_start_cmpl_evt_para (C++ member), 223
esp_ble_gap_cb_param_t::ble_adv_stop_cmpl_evt_para (C++ class), 223
esp_ble_gap_cb_param_t::ble_adv_stop_cmpl_evt_para (C++ member), 223
esp_ble_gap_cb_param_t::ble_clear_bond_dev_cmpl_evt (C++ class), 223

esp_ble_gap_cb_param_t::ble_clear_bond_dev_cmpl_evt_param (C++ member), 223
esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param (C++ class), 223
esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param::bond_dev (C++ member), 224
esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param::dev_num (C++ member), 224
esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param::status (C++ member), 224
esp_ble_gap_cb_param_t::ble_local_privacy_cmpl_evt_param (C++ class), 224
esp_ble_gap_cb_param_t::ble_local_privacy_cmpl_evt_param::status (C++ member), 224
esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_evt_param (C++ class), 224
esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_evt_param::status (C++ member), 224
esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_evt_param::tx_data_length (C++ member), 224
esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param (C++ class), 224
esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param::rssi (C++ member), 224
esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param::status (C++ member), 224
esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param (C++ class), 224
esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param::bd_addr (C++ member), 224
esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param::status (C++ member), 224
esp_ble_gap_cb_param_t::ble_scan_param_cmpl_evt_param (C++ class), 224
esp_ble_gap_cb_param_t::ble_scan_param_cmpl_evt_param::status (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param (C++ class), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::adv_data_len (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::bda (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::ble_addr_type (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::ble_evt (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::dev_type (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::flag (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::num_dis (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::rssi (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::search_evt (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_result_evt_param::status (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param (C++ class), 225
esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::status (C++ member), 225
esp_ble_gap_cb_param_t::ble_scan_rsp_data_raw_cmpl_evt_param (C++ class), 225
esp_ble_gap_cb_param_t::ble_scan_rsp_data_raw_cmpl_evt_param::data_length (C++ member), 226
esp_ble_gap_cb_param_t::ble_scan_rsp_data_raw_cmpl_evt_param::status (C++ member), 226
esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_param (C++ class), 226
esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_param::status (C++ member), 226
esp_ble_gap_cb_param_t::ble_scan_stop_cmpl_evt_param (C++ class), 226
esp_ble_gap_cb_param_t::ble_scan_stop_cmpl_evt_param::status (C++ member), 226
esp_ble_gap_cb_param_t::ble_security (C++ member), 222
esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param (C++ class), 226
esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param::status (C++ member), 226
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param (C++ class), 226
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::bda (C++ member), 226
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::conn_int (C++ member), 226
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::latency (C++ member), 226
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::status (C++ member), 226
esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::timeout (C++ member), 226
esp_ble_gap_cb_param_t::ble_update_whitelist_cmpl_evt_param (C++ class), 227
esp_ble_gap_cb_param_t::ble_update_whitelist_cmpl_evt_param::operation (C++ member), 227
esp_ble_gap_cb_param_t::ble_update_whitelist_cmpl_evt_param::status (C++ member), 227

esp_ble_gap_cb_param_t::clear_bond_dev_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::get_bond_dev_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::local_privacy_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::pkt_data_lenth_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::read_rssi_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::remove_bond_dev_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::scan_param_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::scan_rsp_data_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::scan_rsp_data_raw_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::scan_rst (C++ member), 222
esp_ble_gap_cb_param_t::scan_start_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::scan_stop_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::set_rand_addr_cmpl (C++ member), 222
esp_ble_gap_cb_param_t::update_conn_params (C++ member), 222
esp_ble_gap_cb_param_t::update_whitelist_cmpl (C++ member), 223
esp_ble_gap_config_adv_data (C++ function), 215
esp_ble_gap_config_adv_data_raw (C++ function), 218
esp_ble_gap_config_local_icon (C++ function), 216
esp_ble_gap_config_local_privacy (C++ function), 216
esp_ble_gap_config_scan_rsp_data_raw (C++ function), 218
esp_ble_gap_disconnect (C++ function), 221
esp_ble_gap_get_local_used_addr (C++ function), 218
esp_ble_gap_get_whitelist_size (C++ function), 217
esp_ble_gap_read_rssi (C++ function), 219
esp_ble_gap_register_callback (C++ function), 214
esp_ble_gap_security_rsp (C++ function), 219
esp_ble_gap_set_device_name (C++ function), 218
esp_ble_gap_set_pkt_data_len (C++ function), 216
esp_ble_gap_set_prefer_conn_params (C++ function), 217
esp_ble_gap_set_rand_addr (C++ function), 216
esp_ble_gap_set_scan_params (C++ function), 215
esp_ble_gap_set_security_param (C++ function), 219
esp_ble_gap_start_advertising (C++ function), 215
esp_ble_gap_start_scanning (C++ function), 215
esp_ble_gap_stop_advertising (C++ function), 216
esp_ble_gap_stop_scanning (C++ function), 215
esp_ble_gap_update_conn_params (C++ function), 216
esp_ble_gap_update_whitelist (C++ function), 217
esp_ble_gattc_app_register (C++ function), 266
esp_ble_gattc_app_unregister (C++ function), 267
esp_ble_gattc_cache_assoc (C++ function), 276
esp_ble_gattc_cache_get_addr_list (C++ function), 276
esp_ble_gattc_cache_refresh (C++ function), 275
esp_ble_gattc_cb_param_t (C++ type), 276
esp_ble_gattc_cb_param_t::cfg_mtu (C++ member), 277
esp_ble_gattc_cb_param_t::close (C++ member), 276
esp_ble_gattc_cb_param_t::congest (C++ member), 277
esp_ble_gattc_cb_param_t::connect (C++ member), 277
esp_ble_gattc_cb_param_t::disconnect (C++ member), 277
esp_ble_gattc_cb_param_t::exec_cmpl (C++ member), 277
esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param (C++ class), 277
esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param: (C++ member), 277
esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param: (C++ member), 277
esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param: (C++ member), 277
esp_ble_gattc_cb_param_t::gattc_close_evt_param (C++ class), 278
esp_ble_gattc_cb_param_t::gattc_close_evt_param::co (C++ member), 278

esp_ble_gattc_cb_param_t::gattc_close_evt_param::reason (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_close_evt_param::remote_bda (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_close_evt_param::status (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_congest_evt_param (C++ class), 278
esp_ble_gattc_cb_param_t::gattc_congest_evt_param::congested (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_congest_evt_param::conn_id (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_connect_evt_param (C++ class), 278
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::conn_id (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::remote_bda (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param (C++ class), 278
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::conn_id (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::reason (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::remote_bda (C++ member), 278
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param (C++ class), 278
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param::conn_id (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param::status (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param (C++ class), 279
esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param::addr_list (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param::num_addr (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param::status (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_notify_evt_param (C++ class), 279
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::conn_id (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::handle (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::is_notify (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::remote_bda (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::value (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_notify_evt_param::value_len (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_open_evt_param (C++ class), 279
esp_ble_gattc_cb_param_t::gattc_open_evt_param::conn_id (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_open_evt_param::mtu (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_open_evt_param::remote_bda (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_open_evt_param::status (C++ member), 279
esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param (C++ class), 280
esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param::conn_id (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param::num_cmd (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param::remote_bda (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param (C++ class), 280
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::conn_id (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::handle (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::value_type (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::value (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_reg_evt_param (C++ class), 280
esp_ble_gattc_cb_param_t::gattc_reg_evt_param::app_id (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_reg_evt_param::status (C++ member), 280
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param (C++ class), 280
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::handle (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::status (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param (C++ class), 281
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::conn_id (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::searched_service_source (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param::status (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param (C++ class), 281
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::conn_id (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::end_handle (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param::start_handle (C++ member), 281

esp_ble_gattc_cb_param_t::gattc_search_res (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_search_res_evt_param (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_set_assoc_addr_cmp (C++ class), 281
esp_ble_gattc_cb_param_t::gattc_set_assoc_addr_cmp_evt_param (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_srvc_chg_evt_param (C++ class), 281
esp_ble_gattc_cb_param_t::gattc_srvc_chg_evt_param::remote_bda (C++ member), 281
esp_ble_gattc_cb_param_t::gattc_unreg_for_notify (C++ class), 281
esp_ble_gattc_cb_param_t::gattc_unreg_for_notify_evt_param::conn_id (C++ member), 282
esp_ble_gattc_cb_param_t::gattc_unreg_for_notify_evt_param::status (C++ member), 282
esp_ble_gattc_cb_param_t::gattc_write_evt (C++ class), 282
esp_ble_gattc_cb_param_t::gattc_write_evt_param::conn_id (C++ member), 282
esp_ble_gattc_cb_param_t::gattc_write_evt_param::handle (C++ member), 282
esp_ble_gattc_cb_param_t::gattc_write_evt_param::offset (C++ member), 282
esp_ble_gattc_cb_param_t::gattc_write_evt_param::status (C++ member), 282
esp_ble_gattc_cb_param_t::get_addr_list (C++ member), 277
esp_ble_gattc_cb_param_t::notify (C++ member), 277
esp_ble_gattc_cb_param_t::open (C++ member), 276
esp_ble_gattc_cb_param_t::queue_full (C++ member), 277
esp_ble_gattc_cb_param_t::read (C++ member), 277
esp_ble_gattc_cb_param_t::reg (C++ member), 276
esp_ble_gattc_cb_param_t::reg_for_notify (C++ member), 277
esp_ble_gattc_cb_param_t::search_cmpl (C++ member), 277
esp_ble_gattc_cb_param_t::search_res (C++ member), 277
esp_ble_gattc_cb_param_t::set_assoc_cmp (C++ member), 277
esp_ble_gattc_cb_param_t::srvc_chg (C++ member), 277
esp_ble_gattc_cb_param_t::unreg_for_notify (C++ member), 277
esp_ble_gattc_cb_param_t::write (C++ member), 277
esp_ble_gattc_cb_param_t::write_evt (C++ function), 267
esp_ble_gattc_execute_write (C++ function), 275
esp_ble_gattc_get_all_char (C++ function), 268
esp_ble_gattc_get_all_descr (C++ function), 269
esp_ble_gattc_get_attr_count (C++ function), 271
esp_ble_gattc_get_char_by_uuid (C++ function), 269
esp_ble_gattc_get_db (C++ function), 272
esp_ble_gattc_get_descr_by_char_handle (C++ function), 270
esp_ble_gattc_get_descr_by_uuid (C++ function), 270
esp_ble_gattc_get_include_service (C++ function), 271
esp_ble_gattc_get_service (C++ function), 268
esp_ble_gattc_open (C++ function), 267
esp_ble_gattc_prepare_write (C++ function), 274
esp_ble_gattc_prepare_write_char_descr (C++ function), 274
esp_ble_gattc_read_char (C++ function), 272
esp_ble_gattc_read_char_descr (C++ function), 273
esp_ble_gattc_read_multiple (C++ function), 272
esp_ble_gattc_register_callback (C++ function), 266
esp_ble_gattc_register_for_notify (C++ function), 275
esp_ble_gattc_search_service (C++ function), 268
esp_ble_gattc_send_mtu_req (C++ function), 267
esp_ble_gattc_unregister_for_notify (C++ function), 275
esp_ble_gattc_write_char (C++ function), 273
esp_ble_gattc_write_char_descr (C++ function), 273
esp_ble_gatts_add_char (C++ function), 254
esp_ble_gatts_add_char_descr (C++ function), 254
esp_ble_gatts_add_included_service (C++ function), 253
esp_ble_gatts_app_register (C++ function), 252
esp_ble_gatts_app_unregister (C++ function), 253

esp_ble_gatts_cb_param_t (*C++ type*), 257
esp_ble_gatts_cb_param_t::add_attr_tab (*C++ member*), 258
esp_ble_gatts_cb_param_t::add_char (*C++ member*), 257
esp_ble_gatts_cb_param_t::add_char_descr (*C++ member*), 257
esp_ble_gatts_cb_param_t::add_incl_srvc (*C++ member*), 257
esp_ble_gatts_cb_param_t::cancel_open (*C++ member*), 258
esp_ble_gatts_cb_param_t::close (*C++ member*), 258
esp_ble_gatts_cb_param_t::conf (*C++ member*), 257
esp_ble_gatts_cb_param_t::congest (*C++ member*), 258
esp_ble_gatts_cb_param_t::connect (*C++ member*), 258
esp_ble_gatts_cb_param_t::create (*C++ member*), 257
esp_ble_gatts_cb_param_t::del (*C++ member*), 258
esp_ble_gatts_cb_param_t::disconnect (*C++ member*), 258
esp_ble_gatts_cb_param_t::exec_write (*C++ member*), 257
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param (*C++ class*), 258
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param::attr_num (*C++ member*), 258
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param::handles (*C++ member*), 258
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param::status (*C++ member*), 258
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param::svc_uuid (*C++ member*), 258
esp_ble_gatts_cb_param_t::gatts_add_char_evt_param (*C++ class*), 258
esp_ble_gatts_cb_param_t::gatts_add_char_evt_param::attr_handle (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_char_evt_param::char_uuid (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_char_evt_param::service_handle (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_char_evt_param::status (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param (*C++ class*), 259
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param::attr_handle (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param::descr_uuid (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param::service_handle (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param::status (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param (*C++ class*), 259
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param::attr_handle (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param::service_handle (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param::status (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param::svc_uuid (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_cancel_open_evt_param (*C++ class*), 259
esp_ble_gatts_cb_param_t::gatts_cancel_open_evt_param::status (*C++ member*), 259
esp_ble_gatts_cb_param_t::gatts_close_evt_param (*C++ class*), 259
esp_ble_gatts_cb_param_t::gatts_close_evt_param::conn_id (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_close_evt_param::status (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_conf_evt_param (*C++ class*), 260
esp_ble_gatts_cb_param_t::gatts_conf_evt_param::conn_id (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_conf_evt_param::length (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_conf_evt_param::status (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_conf_evt_param::value (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_congest_evt_param (*C++ class*), 260
esp_ble_gatts_cb_param_t::gatts_congest_evt_param::congested (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_congest_evt_param::conn_id (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_connect_evt_param (*C++ class*), 260
esp_ble_gatts_cb_param_t::gatts_connect_evt_param::conn_id (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_connect_evt_param::link_role (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_connect_evt_param::remote_bda (*C++ member*), 260
esp_ble_gatts_cb_param_t::gatts_create_evt_param (*C++ class*), 260
esp_ble_gatts_cb_param_t::gatts_create_evt_param::service_id (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_create_evt_param::service_handle (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_delete_evt_param (*C++ class*), 261
esp_ble_gatts_cb_param_t::gatts_delete_evt_param::service_handle (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param (*C++ class*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::conn_id (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::is_cancel (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::status (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param (*C++ class*), 261
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param::conn_id (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param::mtu (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_read_evt_param (*C++ class*), 261
esp_ble_gatts_cb_param_t::gatts_read_evt_param::attr_handle (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_read_evt_param::need_rsp (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_read_evt_param::offset (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_write_evt_param (*C++ class*), 261
esp_ble_gatts_cb_param_t::gatts_write_evt_param::attr_handle

(*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_delete_evt_param::status (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param (*C++ class*), 261
esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param::conn_id (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param::reason (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param::remote_bda (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param (*C++ class*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::bda (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::conn_id (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::exec_write_flag (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::trans_id (*C++ member*), 261
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param (*C++ class*), 261
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param::conn_id (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param::mtu (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_open_evt_param (*C++ class*), 262
esp_ble_gatts_cb_param_t::gatts_open_evt_param::status (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_read_evt_param (*C++ class*), 262
esp_ble_gatts_cb_param_t::gatts_read_evt_param::bda (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_read_evt_param::conn_id (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_read_evt_param::handle (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_read_evt_param::is_long (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_read_evt_param::need_rsp (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_read_evt_param::offset (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_read_evt_param::trans_id (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_reg_evt_param (*C++ class*), 262
esp_ble_gatts_cb_param_t::gatts_reg_evt_param::app_id (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_reg_evt_param::status (*C++ member*), 262
esp_ble_gatts_cb_param_t::gatts_rsp_evt_param (*C++ class*), 262
esp_ble_gatts_cb_param_t::gatts_rsp_evt_param::handle (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_rsp_evt_param::status (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param (*C++ class*), 263
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::attr_id (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::length (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::service_handle (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::status (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_start_evt_param (*C++ class*), 263
esp_ble_gatts_cb_param_t::gatts_start_evt_param::conn_id (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_start_evt_param::service_handle (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_start_evt_param::status (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_stop_evt_param (*C++ class*), 263
esp_ble_gatts_cb_param_t::gatts_stop_evt_param::service_handle (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_stop_evt_param::status (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_write_evt_param (*C++ class*), 263
esp_ble_gatts_cb_param_t::gatts_write_evt_param::bda (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_write_evt_param::conn_id (*C++ member*), 263
esp_ble_gatts_cb_param_t::gatts_write_evt_param::handle (*C++ member*), 264
esp_ble_gatts_cb_param_t::gatts_write_evt_param::is_prep (*C++ member*), 264
esp_ble_gatts_cb_param_t::gatts_write_evt_param::length (*C++ member*), 264
esp_ble_gatts_cb_param_t::gatts_write_evt_param::need_rsp (*C++ member*), 264
esp_ble_gatts_cb_param_t::gatts_write_evt_param::offset (*C++ member*), 264
esp_ble_gatts_cb_param_t::gatts_write_evt_param::value (*C++ member*), 264
esp_ble_gatts_cb_param_t::mtu (*C++ member*), 257
esp_ble_gatts_cb_param_t::open (*C++ member*), 258
esp_ble_gatts_cb_param_t::read (*C++ member*), 257
esp_ble_gatts_cb_param_t::reg (*C++ member*), 257
esp_ble_gatts_cb_param_t::rsp (*C++ mem-*

ber), 258
esp_ble_gatts_cb_param_t::set_attr_val (*C++ member*), 258
esp_ble_gatts_cb_param_t::start (*C++ member*), 258
esp_ble_gatts_cb_param_t::stop (*C++ member*), 258
esp_ble_gatts_cb_param_t::write (*C++ member*), 257
esp_ble_gatts_close (*C++ function*), 257
esp_ble_gatts_create_attr_tab (*C++ function*), 253
esp_ble_gatts_create_service (*C++ function*), 253
esp_ble_gatts_delete_service (*C++ function*), 254
esp_ble_gatts_get_attr_value (*C++ function*), 256
esp_ble_gatts_open (*C++ function*), 256
esp_ble_gatts_register_callback (*C++ function*), 252
esp_ble_gatts_send_indicate (*C++ function*), 255
esp_ble_gatts_send_response (*C++ function*), 256
esp_ble_gatts_set_attr_value (*C++ function*), 256
esp_ble_gatts_start_service (*C++ function*), 255
esp_ble_gatts_stop_service (*C++ function*), 255
esp_ble_get_bond_device_list (*C++ function*), 220
esp_ble_get_bond_device_num (*C++ function*), 220
ESP_BLE_ID_KEY_MASK (*C macro*), 210
esp_ble_io_cap_t (*C++ type*), 235
ESP_BLE_IS_VALID_PARAM (*C macro*), 210
esp_ble_key_mask_t (*C++ type*), 210
esp_ble_key_t (*C++ class*), 231
esp_ble_key_t::bd_addr (*C++ member*), 232
esp_ble_key_t::key_type (*C++ member*), 232
esp_ble_key_t::p_key_value (*C++ member*), 232
esp_ble_key_type_t (*C++ type*), 235
esp_ble_key_value_t (*C++ type*), 221
esp_ble_key_value_t::lcsrk_key (*C++ member*), 221
esp_ble_key_value_t::lenc_key (*C++ member*), 221
esp_ble_key_value_t::pcsrk_key (*C++ member*), 221
esp_ble_key_value_t::penc_key (*C++ member*), 221
esp_ble_key_value_t::pid_key (*C++ member*), 221
esp_ble_lcsrk_keys (*C++ class*), 230
esp_ble_lcsrk_keys::counter (*C++ member*), 230
esp_ble_lcsrk_keys::csrk (*C++ member*), 231
esp_ble_lcsrk_keys::div (*C++ member*), 230
esp_ble_lcsrk_keys::sec_level (*C++ member*), 231
esp_ble_lenc_keys_t (*C++ class*), 230
esp_ble_lenc_keys_t::div (*C++ member*), 230
esp_ble_lenc_keys_t::key_size (*C++ member*), 230
esp_ble_lenc_keys_t::ltk (*C++ member*), 230
esp_ble_lenc_keys_t::sec_level (*C++ member*), 230
ESP_BLE_LINK_KEY_MASK (*C macro*), 210
esp_ble_local_id_keys_t (*C++ class*), 232
esp_ble_local_id_keys_t::dhk (*C++ member*), 232
esp_ble_local_id_keys_t::ir (*C++ member*), 232
esp_ble_local_id_keys_t::irk (*C++ member*), 232
ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_DISABLE (*C macro*), 233
ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_ENABLE (*C macro*), 233
esp_ble_passkey_reply (*C++ function*), 220
esp_ble_pcsrk_keys_t (*C++ class*), 230
esp_ble_pcsrk_keys_t::counter (*C++ member*), 230
esp_ble_pcsrk_keys_t::csrk (*C++ member*), 230
esp_ble_pcsrk_keys_t::sec_level (*C++ member*), 230
esp_ble_penc_keys_t (*C++ class*), 229
esp_ble_penc_keys_t::ediv (*C++ member*), 229
esp_ble_penc_keys_t::key_size (*C++ member*), 229
esp_ble_penc_keys_t::ltk (*C++ member*), 229
esp_ble_penc_keys_t::rand (*C++ member*), 229
esp_ble_penc_keys_t::sec_level (*C++ member*), 229
esp_ble_pid_keys_t (*C++ class*), 230
esp_ble_pid_keys_t::addr_type (*C++ member*), 230
esp_ble_pid_keys_t::irk (*C++ member*), 230
esp_ble_pid_keys_t::static_addr (*C++ member*), 230
esp_ble_pkt_data_length_params_t (*C++ class*), 229

esp_ble_pkt_data_length_params_t::rx_len (C++ member), 229
esp_ble_pkt_data_length_params_t::tx_len (C++ member), 229
esp_ble_power_type_t (C++ type), 207
ESP_BLE_PWR_TYPE_ADV (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL0 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL1 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL2 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL3 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL4 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL5 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL6 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL7 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_CONN_HDL8 (C++ enumerator), 207
ESP_BLE_PWR_TYPE_DEFAULT (C++ enumerator), 207
ESP_BLE_PWR_TYPE_NUM (C++ enumerator), 207
ESP_BLE_PWR_TYPE_SCAN (C++ enumerator), 207
esp_ble_remove_bond_device (C++ function), 220
esp_ble_resolve_adv_data (C++ function), 218
esp_ble_scan_duplicate_t (C++ type), 240
esp_ble_scan_filter_t (C++ type), 239
ESP_BLE_SCAN_PARAM_UNDEF (C macro), 210
esp_ble_scan_params_t (C++ class), 228
esp_ble_scan_params_t::own_addr_type (C++ member), 228
esp_ble_scan_params_t::scan_duplicate (C++ member), 229
esp_ble_scan_params_t::scan_filter_policy (C++ member), 228
esp_ble_scan_params_t::scan_interval (C++ member), 228
esp_ble_scan_params_t::scan_type (C++ member), 228
esp_ble_scan_params_t::scan_window (C++ member), 228
ESP_BLE_SCAN_RSP_DATA_LEN_MAX (C macro), 235
esp_ble_scan_type_t (C++ type), 239
esp_ble_sec_act_t (C++ type), 238
ESP_BLE_SEC_ENCRYPT (C++ enumerator), 238
ESP_BLE_SEC_ENCRYPT_MITM (C++ enumerator), 239
ESP_BLE_SEC_ENCRYPT_NO_MITM (C++ enumerator), 239
esp_ble_sec_key_notif_t (C++ class), 231
esp_ble_sec_key_notif_t::bd_addr (C++ member), 231
esp_ble_sec_key_notif_t::passkey (C++ member), 231
esp_ble_sec_req_t (C++ class), 231
esp_ble_sec_req_t::bd_addr (C++ member), 231
esp_ble_sec_t (C++ type), 221
esp_ble_sec_t::auth_cmpl (C++ member), 221
esp_ble_sec_t::ble_id_keys (C++ member), 221
esp_ble_sec_t::ble_key (C++ member), 221
esp_ble_sec_t::ble_req (C++ member), 221
esp_ble_sec_t::key_notif (C++ member), 221
esp_ble_set_encryption (C++ function), 219
ESP_BLE_SM_AUTHEN_REQ_MODE (C++ enumerator), 239
ESP_BLE_SM_CLEAR_STATIC_PASSKEY (C++ enumerator), 239
ESP_BLE_SM_IOCAP_MODE (C++ enumerator), 239
ESP_BLE_SM_MAX_KEY_SIZE (C++ enumerator), 239
ESP_BLE_SM_MAX_PARAM (C++ enumerator), 239
ESP_BLE_SM_MIN_KEY_SIZE (C++ enumerator), 239
ESP_BLE_SM_ONLY_ACCEPT_SPECIFIED_SEC_AUTH (C++ enumerator), 239
esp_ble_sm_param_t (C++ type), 239
ESP_BLE_SM_PASSKEY (C++ enumerator), 239
ESP_BLE_SM_SET_INIT_KEY (C++ enumerator), 239
ESP_BLE_SM_SET_RSP_KEY (C++ enumerator), 239
ESP_BLE_SM_SET_STATIC_PASSKEY (C++ enumerator), 239
esp_ble_tx_power_get (C++ function), 202
esp_ble_tx_power_set (C++ function), 202
ESP_BLE_WHITELIST_ADD (C++ enumerator), 241
ESP_BLE_WHITELIST_REMOVE (C++ enumerator), 241
esp_ble_wl_opration_t (C++ type), 241
esp_bluedroid_deinit (C++ function), 212
esp_bluedroid_disable (C++ function), 212
esp_bluedroid_enable (C++ function), 212
esp_bluedroid_get_status (C++ function), 212
esp_bluedroid_init (C++ function), 212
ESP_BLUEDROID_STATUS_CHECK (C macro), 209
ESP_BLUEDROID_STATUS_ENABLED (C++ enumerator), 213
ESP_BLUEDROID_STATUS_INITIALIZED (C++ enumerator), 213
esp_bluedroid_status_t (C++ type), 213

ESP_BLUEDROID_STATUS_UNINITIALIZED (*C++ enumerator*), 213
esp_blufi_ap_record_t (*C++ class*), 292
esp_blufi_ap_record_t::rssi (*C++ member*), 293
esp_blufi_ap_record_t::ssid (*C++ member*), 293
esp_blufi_callbacks_t (*C++ class*), 293
esp_blufi_callbacks_t::checksum_func (*C++ member*), 293
esp_blufi_callbacks_t::decrypt_func (*C++ member*), 293
esp_blufi_callbacks_t::encrypt_func (*C++ member*), 293
esp_blufi_callbacks_t::event_cb (*C++ member*), 293
esp_blufi_callbacks_t::negotiate_data_handler (*C++ member*), 293
esp_blufi_cb_event_t (*C++ type*), 294
esp_blufi_cb_param_t (*C++ type*), 287
esp_blufi_cb_param_t::blufi_connect_evt_param (*C++ class*), 288
esp_blufi_cb_param_t::blufi_connect_evt_param::conn_id (*C++ member*), 288
esp_blufi_cb_param_t::blufi_connect_evt_param::remote_bda (*C++ member*), 288
esp_blufi_cb_param_t::blufi_connect_evt_param::service_uuid16 (*C++ member*), 288
esp_blufi_cb_param_t::blufi_deinit_finish_evt_param (*C++ class*), 288
esp_blufi_cb_param_t::blufi_deinit_finish_evt_param::state (*C++ member*), 288
esp_blufi_cb_param_t::blufi_disconnect_evt_param (*C++ class*), 288
esp_blufi_cb_param_t::blufi_disconnect_evt_param::conn_id (*C++ member*), 288
esp_blufi_cb_param_t::blufi_get_error_evt_param (*C++ class*), 288
esp_blufi_cb_param_t::blufi_get_error_evt_param::state (*C++ member*), 288
esp_blufi_cb_param_t::blufi_init_finish_evt_param (*C++ class*), 288
esp_blufi_cb_param_t::blufi_init_finish_evt_param::state (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_ca_evt_param (*C++ class*), 289
esp_blufi_cb_param_t::blufi_recv_ca_evt_param::cert (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_ca_evt_param::cert_blen (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_client_cert_evt_param (*C++ class*), 289
esp_blufi_cb_param_t::blufi_recv_client_cert_evt_param::cert (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_client_cert_evt_param::cert_blen (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_client_pkey_evt_param (*C++ class*), 289
esp_blufi_cb_param_t::blufi_recv_client_pkey_evt_param::pkey (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_client_pkey_evt_param::pkey_blen (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_custom_data_evt_param (*C++ class*), 289
esp_blufi_cb_param_t::blufi_recv_custom_data_evt_param::data (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_custom_data_evt_param::data_blen (*C++ member*), 289
esp_blufi_cb_param_t::blufi_recv_server_cert_evt_param (*C++ class*), 289
esp_blufi_cb_param_t::blufi_recv_server_cert_evt_param::cert (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_server_cert_evt_param::cert_blen (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_server_pkey_evt_param (*C++ class*), 290
esp_blufi_cb_param_t::blufi_recv_server_pkey_evt_param::pkey (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_server_pkey_evt_param::pkey_blen (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_softap_auth_mode_evt_param (*C++ class*), 290
esp_blufi_cb_param_t::blufi_recv_softap_auth_mode_evt_param::auth_mode (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_softap_channel_evt_param (*C++ class*), 290
esp_blufi_cb_param_t::blufi_recv_softap_channel_evt_param::channel (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_softap_max_conn_num_evt_param (*C++ class*), 290
esp_blufi_cb_param_t::blufi_recv_softap_max_conn_num_evt_param::max_conn_num (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_softap_passwd_evt_param (*C++ class*), 290
esp_blufi_cb_param_t::blufi_recv_softap_passwd_evt_param::passwd (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_softap_passwd_evt_param::passwd_blen (*C++ member*), 290
esp_blufi_cb_param_t::blufi_recv_softap_ssid_evt_param (*C++ class*), 290
esp_blufi_cb_param_t::blufi_recv_softap_ssid_evt_param::ssid (*C++ member*), 291
esp_blufi_cb_param_t::blufi_recv_softap_ssid_evt_param::ssid_blen (*C++ member*), 291
esp_blufi_cb_param_t::blufi_recv_sta_bssid_evt_param (*C++ class*), 291
esp_blufi_cb_param_t::blufi_recv_sta_bssid_evt_param::bssid (*C++ member*), 291

esp_blufi_cb_param_t::blufi_recv_sta_passwd_evt_param (C++ class), 291
esp_blufi_cb_param_t::blufi_recv_sta_passwd_evt_param::length (C++ member), 291
esp_blufi_cb_param_t::blufi_recv_sta_passwd_evt_param::passwd (C++ member), 291
esp_blufi_cb_param_t::blufi_recv_sta_ssid_evt_param (C++ class), 291
esp_blufi_cb_param_t::blufi_recv_sta_ssid_evt_param::length (C++ member), 291
esp_blufi_cb_param_t::blufi_recv_sta_ssid_evt_param::ssid (C++ member), 291
esp_blufi_cb_param_t::blufi_recv_username_evt_param (C++ class), 291
esp_blufi_cb_param_t::blufi_recv_username_evt_param::length (C++ member), 291
esp_blufi_cb_param_t::blufi_recv_username_evt_param::name (C++ member), 291
esp_blufi_cb_param_t::blufi_set_wifi_mode_evt_param (C++ class), 291
esp_blufi_cb_param_t::blufi_set_wifi_mode_evt_param::opmode (C++ member), 292
esp_blufi_cb_param_t::ca (C++ member), 287
esp_blufi_cb_param_t::client_cert (C++ member), 287
esp_blufi_cb_param_t::client_pkey (C++ member), 287
esp_blufi_cb_param_t::connect (C++ member), 287
esp_blufi_cb_param_t::custom_data (C++ member), 288
esp_blufi_cb_param_t::deinit_finish (C++ member), 287
esp_blufi_cb_param_t::disconnect (C++ member), 287
esp_blufi_cb_param_t::init_finish (C++ member), 287
esp_blufi_cb_param_t::report_error (C++ member), 288
esp_blufi_cb_param_t::server_cert (C++ member), 287
esp_blufi_cb_param_t::server_pkey (C++ member), 287
esp_blufi_cb_param_t::softap_auth_mode (C++ member), 287
esp_blufi_cb_param_t::softap_channel (C++ member), 287
esp_blufi_cb_param_t::softap_max_conn_num (C++ member), 287
esp_blufi_cb_param_t::softap_passwd (C++ member), 287
esp_blufi_cb_param_t::softap_ssid (C++ member), 287
esp_blufi_cb_param_t::sta_bssid (C++ member), 287
esp_blufi_cb_param_t::sta_passwd (C++ member), 287
esp_blufi_cb_param_t::sta_ssid (C++ member), 287
esp_blufi_cb_param_t::passwd_len (C++ member), 287
esp_blufi_cb_param_t::username (C++ member), 287
esp_blufi_cb_param_t::wifi_mode (C++ member), 287
esp_blufi_cb_param_t::ssid_len
esp_blufi_checksum_func_t (C++ type), 294
esp_blufi_close (C++ function), 286
ESP_BLUFI_DECRYPT_ERROR (C++ enumerator), 295
esp_blufi_decrypt_func_t (C++ type), 294
ESP_BLUFI_DEINIT_FAILED (C++ enumerator), 295
ESP_BLUFI_DEINIT_OK (C++ enumerator), 295
esp_blufi_deinit_state_t (C++ type), 295
ESP_BLUFI_DH_MALLOC_ERROR (C++ enumerator), 296
ESP_BLUFI_DH_PARAM_ERROR (C++ enumerator), 296
ESP_BLUFI_ENCRYPT_ERROR (C++ enumerator), 295
esp_blufi_encrypt_func_t (C++ type), 293
esp_blufi_error_state_t (C++ type), 295
ESP_BLUFI_EVENT_BLE_CONNECT (C++ enumerator), 294
ESP_BLUFI_EVENT_BLE_DISCONNECT (C++ enumerator), 294
esp_blufi_event_cb_t (C++ type), 293
ESP_BLUFI_EVENT_DEAUTHENTICATE_STA (C++ enumerator), 294
ESP_BLUFI_EVENT_DEINIT_FINISH (C++ enumerator), 294
ESP_BLUFI_EVENT_GET_WIFI_LIST (C++ enumerator), 295
ESP_BLUFI_EVENT_GET_WIFI_STATUS (C++ enumerator), 294
ESP_BLUFI_EVENT_INIT_FINISH (C++ enumerator), 294
ESP_BLUFI_EVENT_RECV_CA_CERT (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_CLIENT_CERT (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_CUSTOM_DATA (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_SERVER_CERT (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY

(C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_SOFTAP_SSID (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_STA_BSSID (C++ enumerator), 294
ESP_BLUFI_EVENT_RECV_STA_PASSWD (C++ enumerator), 295
ESP_BLUFI_EVENT_RECV_STA_SSID (C++ enumerator), 294
ESP_BLUFI_EVENT_RECV_USERNAME (C++ enumerator), 295
ESP_BLUFI_EVENT_REPORT_ERROR (C++ enumerator), 295
ESP_BLUFI_EVENT_REQ_CONNECT_TO_AP (C++ enumerator), 294
ESP_BLUFI_EVENT_REQ_DISCONNECT_FROM_AP (C++ enumerator), 294
ESP_BLUFI_EVENT_SET_WIFI_OPMODE (C++ enumerator), 294
esp_blufi_extra_info_t (C++ class), 292
esp_blufi_extra_info_t::softap_authmode (C++ member), 292
esp_blufi_extra_info_t::softap_authmode_set (C++ member), 292
esp_blufi_extra_info_t::softap_channel (C++ member), 292
esp_blufi_extra_info_t::softap_channel_set (C++ member), 292
esp_blufi_extra_info_t::softap_max_conn_num (C++ member), 292
esp_blufi_extra_info_t::softap_max_conn_num_set (C++ member), 292
esp_blufi_extra_info_t::softap_passwd (C++ member), 292
esp_blufi_extra_info_t::softap_passwd_len (C++ member), 292
esp_blufi_extra_info_t::softap_ssid (C++ member), 292
esp_blufi_extra_info_t::softap_ssid_len (C++ member), 292
esp_blufi_extra_info_t::sta_bssid (C++ member), 292
esp_blufi_extra_info_t::sta_bssid_set (C++ member), 292
esp_blufi_extra_info_t::sta_passwd (C++ member), 292
esp_blufi_extra_info_t::sta_passwd_len (C++ member), 292
esp_blufi_extra_info_t::sta_ssid (C++ member), 292
esp_blufi_extra_info_t::sta_ssid_len (C++ member), 292
esp_blufi_get_version (C++ function), 286
ESP_BLUFI_INIT_FAILED (C++ enumerator), 295
ESP_BLUFI_INIT_OK (C++ enumerator), 295
ESP_BLUFI_INIT_SECURITY_ERROR (C++ enumerator), 296
esp_blufi_init_state_t (C++ type), 295
ESP_BLUFI_MAKE_PUBLIC_ERROR (C++ enumerator), 296
esp_blufi_negotiate_data_handler_t (C++ type), 293
esp_blufi_profile_deinit (C++ function), 285
esp_blufi_profile_init (C++ function), 285
ESP_BLUFI_READ_PARAM_ERROR (C++ enumerator), 296
esp_blufi_register_callbacks (C++ function), 285
esp_blufi_send_custom_data (C++ function), 286
esp_blufi_send_error_info (C++ function), 286
esp_blufi_send_wifi_conn_report (C++ function), 285
esp_blufi_send_wifi_list (C++ function), 286
ESP_BLUFI_SEQUENCE_ERROR (C++ enumerator), 295
ESP_BLUFI_STA_CONN_FAIL (C++ enumerator), 295
esp_blufi_sta_conn_state_t (C++ type), 295
ESP_BLUFI_STA_CONN_SUCCESS (C++ enumerator), 295
esp_bredr_sco_datapath_set (C++ function), 202
esp_bredr_tx_power_get (C++ function), 202
esp_bredr_tx_power_set (C++ function), 202
ESP_BT_CLR_COD_SERVICE_CLASS (C++ enumerator), 305
ESP_BT_COD_FORMAT_TYPE_1 (C macro), 304
ESP_BT_COD_FORMAT_TYPE_BIT_MASK (C macro), 304
ESP_BT_COD_FORMAT_TYPE_BIT_OFFSET (C macro), 304
ESP_BT_COD_MAJOR_DEV_AV (C++ enumerator), 307
ESP_BT_COD_MAJOR_DEV_BIT_MASK (C macro), 304
ESP_BT_COD_MAJOR_DEV_BIT_OFFSET (C

macro), 304
ESP_BT_COD_MAJOR_DEV_COMPUTER (*C++ enumerator*), 307
ESP_BT_COD_MAJOR_DEV_HEALTH (*C++ enumerator*), 307
ESP_BT_COD_MAJOR_DEV_IMAGING (*C++ enumerator*), 307
ESP_BT_COD_MAJOR_DEV_LAN_NAP (*C++ enumerator*), 307
ESP_BT_COD_MAJOR_DEV_MISC (*C++ enumerator*), 307
ESP_BT_COD_MAJOR_DEV_PERIPHERAL (*C++ enumerator*), 307
ESP_BT_COD_MAJOR_DEV_PHONE (*C++ enumerator*), 307
esp_bt_cod_major_dev_t (*C++ type*), 307
ESP_BT_COD_MAJOR_DEV_TOY (*C++ enumerator*), 307
ESP_BT_COD_MAJOR_DEV_UNCATEGORIZED (*C++ enumerator*), 307
ESP_BT_COD_MAJOR_DEV_WEARABLE (*C++ enumerator*), 307
ESP_BT_COD_MINOR_DEV_BIT_MASK (*C macro*), 304
ESP_BT_COD_MINOR_DEV_BIT_OFFSET (*C macro*), 304
esp_bt_cod_mode_t (*C++ type*), 305
ESP_BT_COD_SRVC_AUDIO (*C++ enumerator*), 306
ESP_BT_COD_SRVC_BIT_MASK (*C macro*), 304
ESP_BT_COD_SRVC_BIT_OFFSET (*C macro*), 304
ESP_BT_COD_SRVC_CAPTURING (*C++ enumerator*), 306
ESP_BT_COD_SRVC_INFORMATION (*C++ enumerator*), 306
ESP_BT_COD_SRVC_LMTD_DISCOVER (*C++ enumerator*), 306
ESP_BT_COD_SRVC_NETWORKING (*C++ enumerator*), 306
ESP_BT_COD_SRVC_NONE (*C++ enumerator*), 306
ESP_BT_COD_SRVC_OBJ_TRANSFER (*C++ enumerator*), 306
ESP_BT_COD_SRVC_POSITIONING (*C++ enumerator*), 306
ESP_BT_COD_SRVC_RENDERING (*C++ enumerator*), 306
esp_bt_cod_srvc_t (*C++ type*), 306
ESP_BT_COD_SRVC_TELEPHONY (*C++ enumerator*), 306
esp_bt_cod_t (*C++ class*), 303
esp_bt_cod_t::major (*C++ member*), 303
esp_bt_cod_t::minor (*C++ member*), 303
esp_bt_cod_t::reserved_2 (*C++ member*), 303
esp_bt_cod_t::reserved_8 (*C++ member*), 303
esp_bt_cod_t::service (*C++ member*), 303
esp_bt_controller_config_t (*C++ class*), 205
esp_bt_controller_config_t::auto_latency (*C++ member*), 206
esp_bt_controller_config_t::bt_sco_datapath (*C++ member*), 206
esp_bt_controller_config_t::controller_debug_flag (*C++ member*), 206
esp_bt_controller_config_t::controller_task_prio (*C++ member*), 205
esp_bt_controller_config_t::controller_task_stack_s (*C++ member*), 205
esp_bt_controller_config_t::hci_uart_baudrate (*C++ member*), 205
esp_bt_controller_config_t::hci_uart_no (*C++ member*), 205
esp_bt_controller_config_t::mesh_adv_size (*C++ member*), 206
esp_bt_controller_config_t::normal_adv_size (*C++ member*), 205
esp_bt_controller_config_t::scan_duplicate_mode (*C++ member*), 205
esp_bt_controller_config_t::send_adv_reserved_size (*C++ member*), 206
esp_bt_controller_deinit (*C++ function*), 203
esp_bt_controller_disable (*C++ function*), 203
esp_bt_controller_enable (*C++ function*), 203
esp_bt_controller_get_status (*C++ function*), 203
esp_bt_controller_init (*C++ function*), 203
esp_bt_controller_is_sleeping (*C++ function*), 205
esp_bt_controller_mem_release (*C++ function*), 204
ESP_BT_CONTROLLER_STATUS_ENABLED (*C++ enumerator*), 207
ESP_BT_CONTROLLER_STATUS_IDLE (*C++ enumerator*), 207
ESP_BT_CONTROLLER_STATUS_INITED (*C++ enumerator*), 207
ESP_BT_CONTROLLER_STATUS_NUM (*C++ enumerator*), 207
esp_bt_controller_status_t (*C++ type*), 206
esp_bt_controller_wakeup_request (*C++ function*), 205
esp_bt_dev_get_address (*C++ function*), 213
esp_bt_dev_set_device_name (*C++ function*), 213
esp_bt_dev_type_t (*C++ type*), 211
ESP_BT_DEVICE_TYPE_BLE (*C++ enumerator*), 211
ESP_BT_DEVICE_TYPE_BREDR (*C++ enumerator*), 211
ESP_BT_DEVICE_TYPE_DUMO (*C++ enumerator*), 211

ESP_BT_EIR_TYPE_CMPL_128BITS_UUID (*C++ enumerator*), 306
ESP_BT_EIR_TYPE_CMPL_16BITS_UUID (*C++ enumerator*), 306
ESP_BT_EIR_TYPE_CMPL_32BITS_UUID (*C++ enumerator*), 306
ESP_BT_EIR_TYPE_CMPL_LOCAL_NAME (*C++ enumerator*), 306
ESP_BT_EIR_TYPE_FLAGS (*C++ enumerator*), 305
ESP_BT_EIR_TYPE_INCMPL_128BITS_UUID (*C++ enumerator*), 306
ESP_BT_EIR_TYPE_INCMPL_16BITS_UUID (*C++ enumerator*), 306
ESP_BT_EIR_TYPE_INCMPL_32BITS_UUID (*C++ enumerator*), 306
ESP_BT_EIR_TYPE_MANU_SPECIFIC (*C++ enumerator*), 306
ESP_BT_EIR_TYPE_SHORT_LOCAL_NAME (*C++ enumerator*), 306
esp_bt_eir_type_t (*C++ type*), 305
ESP_BT_EIR_TYPE_TX_POWER_LEVEL (*C++ enumerator*), 306
ESP_BT_GAP_AUTH_CMPL_EVT (*C++ enumerator*), 308
esp_bt_gap_cancel_discovery (*C++ function*), 298
esp_bt_gap_cb_event_t (*C++ type*), 307
esp_bt_gap_cb_param_t (*C++ type*), 301
esp_bt_gap_cb_param_t::auth_cmpl (*C++ member*), 301
esp_bt_gap_cb_param_t::auth_cmpl_param (*C++ class*), 301
esp_bt_gap_cb_param_t::auth_cmpl_param::bda (*C++ member*), 301
esp_bt_gap_cb_param_t::auth_cmpl_param::device_name (*C++ member*), 301
esp_bt_gap_cb_param_t::auth_cmpl_param::stat (*C++ member*), 301
esp_bt_gap_cb_param_t::disc_res (*C++ member*), 301
esp_bt_gap_cb_param_t::disc_res_param (*C++ class*), 301
esp_bt_gap_cb_param_t::disc_res_param::bda (*C++ member*), 301
esp_bt_gap_cb_param_t::disc_res_param::num_prop (*C++ member*), 301
esp_bt_gap_cb_param_t::disc_res_param::prop (*C++ member*), 301
esp_bt_gap_cb_param_t::disc_st_chg (*C++ member*), 301
esp_bt_gap_cb_param_t::disc_state_changed_param (*C++ class*), 302
esp_bt_gap_cb_param_t::disc_state_changed (*C++ member*), 302
esp_bt_gap_cb_param_t::pin_req (*C++ member*), 301
esp_bt_gap_cb_param_t::pin_req_param (*C++ class*), 302
esp_bt_gap_cb_param_t::pin_req_param::bda (*C++ member*), 302
esp_bt_gap_cb_param_t::pin_req_param::min_16_digit (*C++ member*), 302
esp_bt_gap_cb_param_t::read_rssi_delta (*C++ member*), 301
esp_bt_gap_cb_param_t::read_rssi_delta_param (*C++ class*), 302
esp_bt_gap_cb_param_t::read_rssi_delta_param::bda (*C++ member*), 302
esp_bt_gap_cb_param_t::read_rssi_delta_param::rssi (*C++ member*), 302
esp_bt_gap_cb_param_t::read_rssi_delta_param::stat (*C++ member*), 302
esp_bt_gap_cb_param_t::rmt_srvc_rec (*C++ member*), 301
esp_bt_gap_cb_param_t::rmt_srvc_rec_param (*C++ class*), 302
esp_bt_gap_cb_param_t::rmt_srvc_rec_param::bda (*C++ member*), 302
esp_bt_gap_cb_param_t::rmt_srvc_rec_param::stat (*C++ member*), 302
esp_bt_gap_cb_param_t::rmt_srvcs (*C++ member*), 301
esp_bt_gap_cb_param_t::rmt_srvcs_param (*C++ class*), 302
esp_bt_gap_cb_param_t::rmt_srvcs_param::bda (*C++ member*), 303
esp_bt_gap_cb_param_t::rmt_srvcs_param::num_uuids (*C++ member*), 303
esp_bt_gap_cb_param_t::rmt_srvcs_param::stat (*C++ member*), 303
esp_bt_gap_cb_param_t::rmt_srvcs_param::uuid_list (*C++ member*), 303
esp_bt_gap_cb_t (*C++ type*), 304
ESP_BT_GAP_DEV_PROP_BDNAME (*C++ enumerator*), 305
ESP_BT_GAP_DEV_PROP_COD (*C++ enumerator*), 305
ESP_BT_GAP_DEV_PROP_EIR (*C++ enumerator*), 305
ESP_BT_GAP_DEV_PROP_RSSI (*C++ enumerator*), 305
esp_bt_gap_dev_prop_t (*C++ class*), 303
esp_bt_gap_dev_prop_t::len (*C++ member*), 303
esp_bt_gap_dev_prop_t::type (*C++ member*), 303
esp_bt_gap_dev_prop_t::val (*C++ member*), 303

esp_bt_gap_dev_prop_type_t (*C++ type*), 305
ESP_BT_GAP_DISC_RES_EVT (*C++ enumerator*), 307
ESP_BT_GAP_DISC_STATE_CHANGED_EVT (*C++ enumerator*), 308
ESP_BT_GAP_DISCOVERY_STARTED (*C++ enumerator*), 307
esp_bt_gap_discovery_state_t (*C++ type*), 307
ESP_BT_GAP_DISCOVERY_STOPPED (*C++ enumerator*), 307
ESP_BT_GAP_EIR_DATA_LEN (*C macro*), 304
ESP_BT_GAP_EVT_MAX (*C++ enumerator*), 308
esp_bt_gap_get_bond_device_list (*C++ function*), 300
esp_bt_gap_get_bond_device_num (*C++ function*), 299
esp_bt_gap_get_cod (*C++ function*), 299
esp_bt_gap_get_cod_format_type (*C++ function*), 297
esp_bt_gap_get_cod_major_dev (*C++ function*), 296
esp_bt_gap_get_cod_minor_dev (*C++ function*), 296
esp_bt_gap_get_cod_srvc (*C++ function*), 296
esp_bt_gap_get_remote_service_record (*C++ function*), 298
esp_bt_gap_get_remote_services (*C++ function*), 298
esp_bt_gap_is_valid_cod (*C++ function*), 297
ESP_BT_GAP_MAX_BDNAME_LEN (*C macro*), 304
ESP_BT_GAP_MAX_INQ_LEN (*C macro*), 304
ESP_BT_GAP_MIN_INQ_LEN (*C macro*), 304
esp_bt_gap_pin_reply (*C++ function*), 300
ESP_BT_GAP_PIN_REQ_EVT (*C++ enumerator*), 308
esp_bt_gap_read_rssi_delta (*C++ function*), 299
ESP_BT_GAP_READ_RSSI_DELTA_EVT (*C++ enumerator*), 308
esp_bt_gap_register_callback (*C++ function*), 297
esp_bt_gap_remove_bond_device (*C++ function*), 299
esp_bt_gap_resolve_eir_data (*C++ function*), 298
ESP_BT_GAP_RMT_SRVC_REC_EVT (*C++ enumerator*), 308
ESP_BT_GAP_RMT_SRVCS_EVT (*C++ enumerator*), 308
ESP_BT_GAP_RSSI_HIGH_THRLD (*C macro*), 303
ESP_BT_GAP_RSSI_LOW_THRLD (*C macro*), 303
esp_bt_gap_set_cod (*C++ function*), 299
esp_bt_gap_set_pin (*C++ function*), 300
esp_bt_gap_set_scan_mode (*C++ function*), 297
esp_bt_gap_start_discovery (*C++ function*), 297
ESP_BT_HF_CLIENT_NUMBER_LEN (*C macro*), 347
ESP_BT_HF_CLIENT_OPERATOR_NAME_LEN (*C macro*), 347
ESP_BT_INIT_COD (*C++ enumerator*), 305
ESP_BT_INQ_MODE_GENERAL_INQUIRY (*C++ enumerator*), 308
ESP_BT_INQ_MODE_LIMITED_INQUIRY (*C++ enumerator*), 308
esp_bt_inq_mode_t (*C++ type*), 308
ESP_BT_MODE_BLE (*C++ enumerator*), 206
ESP_BT_MODE_BTDM (*C++ enumerator*), 206
ESP_BT_MODE_CLASSIC_BT (*C++ enumerator*), 206
ESP_BT_MODE_IDLE (*C++ enumerator*), 206
esp_bt_mode_t (*C++ type*), 206
ESP_BT_OCTET16_LEN (*C macro*), 209
esp_bt_octet16_t (*C++ type*), 210
ESP_BT_OCTET8_LEN (*C macro*), 209
esp_bt_octet8_t (*C++ type*), 210
ESP_BT_PIN_CODE_LEN (*C macro*), 304
esp_bt_pin_code_t (*C++ type*), 304
ESP_BT_PIN_TYPE_FIXED (*C++ enumerator*), 307
esp_bt_pin_type_t (*C++ type*), 307
ESP_BT_PIN_TYPE_VARIABLE (*C++ enumerator*), 307
ESP_BT_SCAN_MODE_CONNECTABLE (*C++ enumerator*), 305
ESP_BT_SCAN_MODE_CONNECTABLE_DISCOVERABLE (*C++ enumerator*), 305
ESP_BT_SCAN_MODE_NONE (*C++ enumerator*), 305
esp_bt_scan_mode_t (*C++ type*), 305
ESP_BT_SET_COD_ALL (*C++ enumerator*), 305
ESP_BT_SET_COD_MAJOR_MINOR (*C++ enumerator*), 305
ESP_BT_SET_COD_SERVICE_CLASS (*C++ enumerator*), 305
esp_bt_sleep_disable (*C++ function*), 204
esp_bt_sleep_enable (*C++ function*), 204
ESP_BT_STATUS_AUTH_FAILURE (*C++ enumerator*), 211
ESP_BT_STATUS_AUTH_REJECTED (*C++ enumerator*), 211
ESP_BT_STATUS_BUSY (*C++ enumerator*), 211
ESP_BT_STATUS_CONTROL_LE_DATA_LEN_UNSUPPORTED (*C++ enumerator*), 211
ESP_BT_STATUS_DONE (*C++ enumerator*), 211
ESP_BT_STATUS_ERR_ILLEGAL_PARAMETER_FMT (*C++ enumerator*), 211
ESP_BT_STATUS_FAIL (*C++ enumerator*), 211
ESP_BT_STATUS_INVALID_STATIC_RAND_ADDR (*C++ enumerator*), 211
ESP_BT_STATUS_MEMORY_FULL (*C++ enumerator*), 211

ESP_BT_STATUS_NOMEM (*C++ enumerator*), 211
ESP_BT_STATUS_NOT_READY (*C++ enumerator*), 211
ESP_BT_STATUS_PARAM_OUT_OF_RANGE (*C++ enumerator*), 211
ESP_BT_STATUS_PARM_INVALID (*C++ enumerator*), 211
ESP_BT_STATUS_PEER_LE_DATA_LEN_UNSUPPORTED (*C++ enumerator*), 211
ESP_BT_STATUS_PENDING (*C++ enumerator*), 211
ESP_BT_STATUS_RMT_DEV_DOWN (*C++ enumerator*), 211
ESP_BT_STATUS_SUCCESS (*C++ enumerator*), 211
esp_bt_status_t (*C++ type*), 210
ESP_BT_STATUS_TIMEOUT (*C++ enumerator*), 211
ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL (*C++ enumerator*), 211
ESP_BT_STATUS_UNHANDLED (*C++ enumerator*), 211
ESP_BT_STATUS_UNSUPPORTED (*C++ enumerator*), 211
esp_bt_uuid_t (*C++ class*), 209
esp_bt_uuid_t::len (*C++ member*), 209
esp_bt_uuid_t::uuid (*C++ member*), 209
esp_deep_sleep_start (*C++ function*), 829
ESP_DEFAULT_GATT_IF (*C macro*), 209
esp_deregister_freertos_idle_hook (*C++ function*), 771
esp_deregister_freertos_idle_hook_for_cpu (*C++ function*), 771
esp_deregister_freertos_tick_hook (*C++ function*), 772
esp_deregister_freertos_tick_hook_for_cpu (*C++ function*), 771
ESP_EARLY_LOGD (*C macro*), 812
ESP_EARLY_LOGE (*C macro*), 812
ESP_EARLY_LOGI (*C macro*), 812
ESP_EARLY_LOGV (*C macro*), 812
ESP_EARLY_LOGW (*C macro*), 812
esp_efuse_mac_get_custom (*C++ function*), 833
ESP_ERR_ESPNOW_ARG (*C macro*), 168
ESP_ERR_ESPNOW_BASE (*C macro*), 168
ESP_ERR_ESPNOW_EXIST (*C macro*), 168
ESP_ERR_ESPNOW_FULL (*C macro*), 168
ESP_ERR_ESPNOW_IF (*C macro*), 168
ESP_ERR_ESPNOW_INTERNAL (*C macro*), 168
ESP_ERR_ESPNOW_NO_MEM (*C macro*), 168
ESP_ERR_ESPNOW_NOT_FOUND (*C macro*), 168
ESP_ERR_ESPNOW_NOT_INIT (*C macro*), 168
ESP_ERR_FLASH_BASE (*C macro*), 607
ESP_ERR_FLASH_OP_FAIL (*C macro*), 607
ESP_ERR_FLASH_OP_TIMEOUT (*C macro*), 607
ESP_ERR_HTTP_BASE (*C macro*), 597
ESP_ERR_HTTP_CONNECT (*C macro*), 597
ESP_ERR_HTTP_FETCH_HEADER (*C macro*), 597
ESP_ERR_HTTP_INVALID_TRANSPORT (*C macro*), 597
ESP_ERR_HTTP_MAX_REDIRECT (*C macro*), 597
ESP_ERR_HTTP_WRITE_DATA (*C macro*), 597
ESP_ERR_INVALID_ARG (*C macro*), 841
ESP_ERR_INVALID_CRC (*C macro*), 842
ESP_ERR_INVALID_MAC (*C macro*), 842
ESP_ERR_INVALID_RESPONSE (*C macro*), 842
ESP_ERR_INVALID_SIZE (*C macro*), 841
ESP_ERR_INVALID_STATE (*C macro*), 841
ESP_ERR_INVALID_VERSION (*C macro*), 842
ESP_ERR_MESH_ARGUMENT (*C macro*), 196
ESP_ERR_MESH_BASE (*C macro*), 842
ESP_ERR_MESH_DISCARD (*C macro*), 197
ESP_ERR_MESH_DISCARD_DUPLICATE (*C macro*), 197
ESP_ERR_MESH_DISCONNECTED (*C macro*), 197
ESP_ERR_MESH_EXCEED_MTU (*C macro*), 196
ESP_ERR_MESH_INTERFACE (*C macro*), 197
ESP_ERR_MESH_NO_MEMORY (*C macro*), 196
ESP_ERR_MESH_NO_PARENT_FOUND (*C macro*), 197
ESP_ERR_MESH_NO_ROUTE_FOUND (*C macro*), 197
ESP_ERR_MESH_NOT_ALLOWED (*C macro*), 196
ESP_ERR_MESH_NOT_CONFIG (*C macro*), 196
ESP_ERR_MESH_NOT_INIT (*C macro*), 196
ESP_ERR_MESH_NOT_START (*C macro*), 196
ESP_ERR_MESH_NOT_SUPPORT (*C macro*), 196
ESP_ERR_MESH_OPTION_NULL (*C macro*), 197
ESP_ERR_MESH_OPTION_UNKNOWN (*C macro*), 197
ESP_ERR_MESH_QUEUE_FAIL (*C macro*), 197
ESP_ERR_MESH_QUEUE_FULL (*C macro*), 197
ESP_ERR_MESH_TIMEOUT (*C macro*), 197
ESP_ERR_MESH_VOTING (*C macro*), 197
ESP_ERR_MESH_WIFI_NOT_START (*C macro*), 196
ESP_ERR_MESH_XON_NO_WINDOW (*C macro*), 197
ESP_ERR_NO_MEM (*C macro*), 841
ESP_ERR_NOT_FOUND (*C macro*), 841
ESP_ERR_NOT_SUPPORTED (*C macro*), 841
ESP_ERR_NVS_BASE (*C macro*), 640
ESP_ERR_NVS_CORRUPT_KEY_PART (*C macro*), 642
ESP_ERR_NVS_ENCR_NOT_SUPPORTED (*C macro*), 641
ESP_ERR_NVS_INVALID_HANDLE (*C macro*), 641
ESP_ERR_NVS_INVALID_LENGTH (*C macro*), 641
ESP_ERR_NVS_INVALID_NAME (*C macro*), 641
ESP_ERR_NVS_INVALID_STATE (*C macro*), 641
ESP_ERR_NVS_KEY_TOO_LONG (*C macro*), 641
ESP_ERR_NVS_KEYS_NOT_INITIALIZED (*C macro*), 641
ESP_ERR_NVS_NEW_VERSION_FOUND (*C macro*), 641
ESP_ERR_NVS_NO_FREE_PAGES (*C macro*), 641
ESP_ERR_NVS_NOT_ENOUGH_SPACE (*C macro*), 641

ESP_ERR_NVS_NOT_FOUND (*C macro*), 641
ESP_ERR_NVS_NOT_INITIALIZED (*C macro*), 640
ESP_ERR_NVS_PAGE_FULL (*C macro*), 641
ESP_ERR_NVS_PART_NOT_FOUND (*C macro*), 641
ESP_ERR_NVS_READ_ONLY (*C macro*), 641
ESP_ERR_NVS_REMOVE_FAILED (*C macro*), 641
ESP_ERR_NVS_TYPE_MISMATCH (*C macro*), 641
ESP_ERR_NVS_VALUE_TOO_LONG (*C macro*), 641
ESP_ERR_NVS_XTS_CFG_FAILED (*C macro*), 641
ESP_ERR_NVS_XTS_CFG_NOT_FOUND (*C macro*), 641
ESP_ERR_NVS_XTS_DECR_FAILED (*C macro*), 641
ESP_ERR_NVS_XTS_ENCR_FAILED (*C macro*), 641
ESP_ERR_OTA_BASE (*C macro*), 837
ESP_ERR_OTA_PARTITION_CONFLICT (*C macro*), 837
ESP_ERR_OTA_SELECT_INFO_INVALID (*C macro*), 837
ESP_ERR_OTA_VALIDATE_FAILED (*C macro*), 837
esp_err_t (*C++ type*), 842
ESP_ERR_TIMEOUT (*C macro*), 841
esp_err_to_name (*C++ function*), 841
esp_err_to_name_r (*C++ function*), 841
ESP_ERR_ULP_BASE (*C macro*), 1112
ESP_ERR_ULP_BRANCH_OUT_OF_RANGE (*C macro*), 1113
ESP_ERR_ULP_DUPLICATE_LABEL (*C macro*), 1112
ESP_ERR_ULP_INVALID_LOAD_ADDR (*C macro*), 1112
ESP_ERR_ULP_SIZE_TOO_BIG (*C macro*), 1112
ESP_ERR_ULP_UNDEFINED_LABEL (*C macro*), 1112
ESP_ERR_WIFI_BASE (*C macro*), 842
ESP_ERR_WIFI_CONN (*C macro*), 143
ESP_ERR_WIFI_IF (*C macro*), 143
ESP_ERR_WIFI_INIT_STATE (*C macro*), 143
ESP_ERR_WIFI_MAC (*C macro*), 143
ESP_ERR_WIFI_MODE (*C macro*), 143
ESP_ERR_WIFI_NOT_CONNECT (*C macro*), 143
ESP_ERR_WIFI_NOT_INIT (*C macro*), 143
ESP_ERR_WIFI_NOT_STARTED (*C macro*), 143
ESP_ERR_WIFI_NOT_STOPPED (*C macro*), 143
ESP_ERR_WIFI_NVS (*C macro*), 143
ESP_ERR_WIFI_PASSWORD (*C macro*), 143
ESP_ERR_WIFI_POST (*C macro*), 143
ESP_ERR_WIFI_SSID (*C macro*), 143
ESP_ERR_WIFI_STATE (*C macro*), 143
ESP_ERR_WIFI_STOP_STATE (*C macro*), 143
ESP_ERR_WIFI_TIMEOUT (*C macro*), 143
ESP_ERR_WIFI_WAKE_FAIL (*C macro*), 143
ESP_ERR_WIFI_WOULD_BLOCK (*C macro*), 143
ESP_ERROR_CHECK (*C macro*), 842
esp_esptouch_set_timeout (*C++ function*), 160
esp_eth_disable (*C++ function*), 352
esp_eth_enable (*C++ function*), 352
esp_eth_free_rx_buf (*C++ function*), 353
esp_eth_get_mac (*C++ function*), 352
esp_eth_init (*C++ function*), 351
esp_eth_init_internal (*C++ function*), 351
esp_eth_set_mac (*C++ function*), 353
esp_eth_smi_read (*C++ function*), 353
esp_eth_smi_wait_set (*C++ function*), 353
esp_eth_smi_wait_value (*C++ function*), 353
esp_eth_smi_write (*C++ function*), 352
esp_eth_tx (*C++ function*), 352
ESP_EXT1_WAKEUP_ALL_LOW (*C++ enumerator*), 827
ESP_EXT1_WAKEUP_ANY_HIGH (*C++ enumerator*), 827
ESP_FAIL (*C macro*), 841
esp_flash_encrypt_check_and_update (*C++ function*), 614
esp_flash_encrypt_region (*C++ function*), 614
esp_flash_encryption_enabled (*C++ function*), 614
esp_flash_write_protect_crypt_cnt (*C++ function*), 614
esp_freertos_idle_cb_t (*C++ type*), 772
esp_freertos_tick_cb_t (*C++ type*), 772
ESP_GAP_BLE_ADD_WHITELIST_COMPLETE_EVT (*C macro*), 235
ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_ADV_START_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_AUTH_CMPL_EVT (*C++ enumerator*), 236
esp_gap_ble_cb_event_t (*C++ type*), 236
esp_gap_ble_cb_t (*C++ type*), 235
ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT (*C++ enumerator*), 237
ESP_GAP_BLE_EVT_MAX (*C++ enumerator*), 237
ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT (*C++ enumerator*), 237
ESP_GAP_BLE_KEY_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_LOCAL_ER_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_LOCAL_IR_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_NC_REQ_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_OOB_REQ_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_PASSKEY_NOTIF_EVT (*C++ enumerator*), 236

ESP_GAP_BLE_PASSKEY_REQ_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT (*C++ enumerator*), 237
ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT (*C++ enumerator*), 237
ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SCAN_RESULT_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SCAN_START_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SEC_REQ_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT (*C++ enumerator*), 236
ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT (*C++ enumerator*), 237
ESP_GAP_SEARCH_DI_DISC_CMPL_EVT (*C++ enumerator*), 240
ESP_GAP_SEARCH_DISC_BLE_RES_EVT (*C++ enumerator*), 240
ESP_GAP_SEARCH_DISC_CMPL_EVT (*C++ enumerator*), 240
ESP_GAP_SEARCH_DISC_RES_EVT (*C++ enumerator*), 240
esp_gap_search_evt_t (*C++ type*), 240
ESP_GAP_SEARCH_INQ_CMPL_EVT (*C++ enumerator*), 240
ESP_GAP_SEARCH_INQ_DISCARD_NUM_EVT (*C++ enumerator*), 240
ESP_GAP_SEARCH_INQ_RES_EVT (*C++ enumerator*), 240
ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT (*C++ enumerator*), 240
ESP_GATT_ALREADY_OPEN (*C++ enumerator*), 250
ESP_GATT_APP_RSP (*C++ enumerator*), 250
ESP_GATT_ATTR_HANDLE_MAX (*C macro*), 248
ESP_GATT_AUTH_FAIL (*C++ enumerator*), 250
ESP_GATT_AUTH_REQ_MITM (*C++ enumerator*), 251
ESP_GATT_AUTH_REQ_NO_MITM (*C++ enumerator*), 251
ESP_GATT_AUTH_REQ_NONE (*C++ enumerator*), 251
ESP_GATT_AUTH_REQ_SIGNED_MITM (*C++ enumerator*), 251
ESP_GATT_AUTH_REQ_SIGNED_NO_MITM (*C++ enumerator*), 251
esp_gatt_auth_req_t (*C++ type*), 251
ESP_GATT_AUTO_RSP (*C macro*), 249
ESP_GATT_BODY_SENSOR_LOCATION (*C macro*), 248
ESP_GATT_BUSY (*C++ enumerator*), 250
ESP_GATT_CANCEL (*C++ enumerator*), 250
ESP_GATT_CCC_CFG_ERR (*C++ enumerator*), 250
ESP_GATT_CHAR_PROP_BIT_AUTH (*C macro*), 248
ESP_GATT_CHAR_PROP_BIT_BROADCAST (*C macro*), 248
ESP_GATT_CHAR_PROP_BIT_EXT_PROP (*C macro*), 248
ESP_GATT_CHAR_PROP_BIT_INDICATE (*C macro*), 248
ESP_GATT_CHAR_PROP_BIT_NOTIFY (*C macro*), 248
ESP_GATT_CHAR_PROP_BIT_READ (*C macro*), 248
ESP_GATT_CHAR_PROP_BIT_WRITE (*C macro*), 248
ESP_GATT_CHAR_PROP_BIT_WRITE_NR (*C macro*), 248
esp_gatt_char_prop_t (*C++ type*), 249
ESP_GATT_CMD_STARTED (*C++ enumerator*), 250
ESP_GATT_CONGESTED (*C++ enumerator*), 250
ESP_GATT_CONN_CONN_CANCEL (*C++ enumerator*), 251
ESP_GATT_CONN_FAIL_ESTABLISH (*C++ enumerator*), 251
ESP_GATT_CONN_L2C_FAILURE (*C++ enumerator*), 250
ESP_GATT_CONN_LMP_TIMEOUT (*C++ enumerator*), 251
ESP_GATT_CONN_NONE (*C++ enumerator*), 251
esp_gatt_conn_reason_t (*C++ type*), 250
ESP_GATT_CONN_TERMINATE_LOCAL_HOST (*C++ enumerator*), 251
ESP_GATT_CONN_TERMINATE_PEER_USER (*C++ enumerator*), 251
ESP_GATT_CONN_TIMEOUT (*C++ enumerator*), 251
ESP_GATT_CONN_UNKNOWN (*C++ enumerator*), 250
ESP_GATT_DB_ALL (*C++ enumerator*), 252
esp_gatt_db_attr_type_t (*C++ type*), 251
ESP_GATT_DB_CHARACTERISTIC (*C++ enumerator*), 251
ESP_GATT_DB_DESCRIPTOR (*C++ enumerator*), 251
ESP_GATT_DB_FULL (*C++ enumerator*), 250
ESP_GATT_DB_INCLUDED_SERVICE (*C++ enumerator*), 251
ESP_GATT_DB_PRIMARY_SERVICE (*C++ enumerator*), 251

ESP_GATT_DB_SECONDARY_SERVICE (*C++ enumerator*), 251
ESP_GATT_DUP_REG (*C++ enumerator*), 250
ESP_GATT_ENCRYPED_MITM (*C++ enumerator*), 250
ESP_GATT_ENCRYPED_NO_MITM (*C++ enumerator*), 250
ESP_GATT_ERR_UNLIKELY (*C++ enumerator*), 250
ESP_GATT_ERROR (*C++ enumerator*), 250
ESP_GATT_HEART_RATE_CNTL_POINT (*C macro*), 248
ESP_GATT_HEART_RATE_MEAS (*C macro*), 247
esp_gatt_id_t (*C++ class*), 241
esp_gatt_id_t::inst_id (*C++ member*), 242
esp_gatt_id_t::uuid (*C++ member*), 242
ESP_GATT_IF_NONE (*C macro*), 249
esp_gatt_if_t (*C++ type*), 249
ESP_GATT_ILLEGAL_HANDLE (*C macro*), 248
ESP_GATT_ILLEGAL_PARAMETER (*C++ enumerator*), 250
ESP_GATT_ILLEGAL_UUID (*C macro*), 248
ESP_GATT_INSUF_AUTHENTICATION (*C++ enumerator*), 249
ESP_GATT_INSUF_AUTHORIZATION (*C++ enumerator*), 249
ESP_GATT_INSUF_ENCRYPTION (*C++ enumerator*), 250
ESP_GATT_INSUF_KEY_SIZE (*C++ enumerator*), 249
ESP_GATT_INSUF_RESOURCE (*C++ enumerator*), 250
ESP_GATT_INTERNAL_ERROR (*C++ enumerator*), 250
ESP_GATT_INVALID_ATTR_LEN (*C++ enumerator*), 249
ESP_GATT_INVALID_CFG (*C++ enumerator*), 250
ESP_GATT_INVALID_HANDLE (*C++ enumerator*), 249
ESP_GATT_INVALID_OFFSET (*C++ enumerator*), 249
ESP_GATT_INVALID_PDU (*C++ enumerator*), 249
ESP_GATT_MAX_ATTR_LEN (*C macro*), 249
ESP_GATT_MAX_READ_MULTI_HANDLES (*C macro*), 248
ESP_GATT_MORE (*C++ enumerator*), 250
ESP_GATT_NO_RESOURCES (*C++ enumerator*), 250
ESP_GATT_NOT_ENCRYPTED (*C++ enumerator*), 250
ESP_GATT_NOT_FOUND (*C++ enumerator*), 249
ESP_GATT_NOT_LONG (*C++ enumerator*), 249
ESP_GATT_OK (*C++ enumerator*), 249
ESP_GATT_OUT_OF_RANGE (*C++ enumerator*), 250
ESP_GATT_PENDING (*C++ enumerator*), 250
ESP_GATT_PERM_READ (*C macro*), 248
ESP_GATT_PERM_READ_ENC_MITM (*C macro*), 248
ESP_GATT_PERM_READ_ENCRYPTED (*C macro*), 248
esp_gatt_perm_t (*C++ type*), 249
ESP_GATT_PERM_WRITE (*C macro*), 248
ESP_GATT_PERM_WRITE_ENC_MITM (*C macro*), 248
ESP_GATT_PERM_WRITE_ENCRYPTED (*C macro*), 248
ESP_GATT_PERM_WRITE_SIGNED (*C macro*), 248
ESP_GATT_PERM_WRITE_SIGNED_MITM (*C macro*), 248
ESP_GATT_PRC_IN_PROGRESS (*C++ enumerator*), 250
ESP_GATT_PREP_WRITE_CANCEL (*C macro*), 264
ESP_GATT_PREP_WRITE_CANCEL (*C++ enumerator*), 249
ESP_GATT_PREP_WRITE_EXEC (*C macro*), 264
ESP_GATT_PREP_WRITE_EXEC (*C++ enumerator*), 249
esp_gatt_prep_write_type (*C++ type*), 249
ESP_GATT_PREPARE_Q_FULL (*C++ enumerator*), 249
ESP_GATT_READ_NOT_PERMIT (*C++ enumerator*), 249
ESP_GATT_REQ_NOT_SUPPORTED (*C++ enumerator*), 249
ESP_GATT_RSP_BY_APP (*C macro*), 249
esp_gatt_rsp_t (*C++ type*), 241
esp_gatt_rsp_t::attr_value (*C++ member*), 241
esp_gatt_rsp_t::handle (*C++ member*), 241
ESP_GATT_SERVICE_STARTED (*C++ enumerator*), 250
esp_gatt_srvc_id_t (*C++ class*), 242
esp_gatt_srvc_id_t::id (*C++ member*), 242
esp_gatt_srvc_id_t::is_primary (*C++ member*), 242
ESP_GATT_STACK_RSP (*C++ enumerator*), 250
esp_gatt_status_t (*C++ type*), 249
ESP_GATT_UNKNOWN_ERROR (*C++ enumerator*), 250
ESP_GATT_UNSUPPORT_GRP_TYPE (*C++ enumerator*), 250
ESP_GATT_UUID_ALERT_LEVEL (*C macro*), 247
ESP_GATT_UUID_ALERT_NTF_SVC (*C macro*), 246
ESP_GATT_UUID_ALERT_STATUS (*C macro*), 247
ESP_GATT_UUID_BATTERY_LEVEL (*C macro*), 248
ESP_GATT_UUID_BATTERY_SERVICE_SVC (*C macro*), 246
ESP_GATT_UUID_BLOOD_PRESSURE_SVC (*C macro*), 246
ESP_GATT_UUID_CHAR_AGG_FORMAT (*C macro*), 246
ESP_GATT_UUID_CHAR_CLIENT_CONFIG (*C macro*), 246
ESP_GATT_UUID_CHAR_DECLARE (*C macro*), 246
ESP_GATT_UUID_CHAR_DESCRIPTION (*C macro*), 246

ESP_GATT_UUID_CHAR_EXT_PROP (*C macro*), 246
ESP_GATT_UUID_CHAR_PRESENT_FORMAT (*C macro*), 246
ESP_GATT_UUID_CHAR_SRVR_CONFIG (*C macro*), 246
ESP_GATT_UUID_CHAR_VALID_RANGE (*C macro*), 246
ESP_GATT_UUID_CSC_FEATURE (*C macro*), 248
ESP_GATT_UUID_CSC_MEASUREMENT (*C macro*), 248
ESP_GATT_UUID_CURRENT_TIME (*C macro*), 247
ESP_GATT_UUID_CURRENT_TIME_SVC (*C macro*), 245
ESP_GATT_UUID_CYCLING_POWER_SVC (*C macro*), 246
ESP_GATT_UUID_CYCLING_SPEED_CADENCE_SVC (*C macro*), 246
ESP_GATT_UUID_DEVICE_INFO_SVC (*C macro*), 246
ESP_GATT_UUID_ENV_SENSING_CONFIG_DESCR (*C macro*), 246
ESP_GATT_UUID_ENV_SENSING_MEASUREMENT_DESCR (*C macro*), 246
ESP_GATT_UUID_ENV_SENSING_TRIGGER_DESCR (*C macro*), 246
ESP_GATT_UUID_EXT_RPT_REF_DESCR (*C macro*), 246
ESP_GATT_UUID_FW_VERSION_STR (*C macro*), 247
ESP_GATT_UUID_GAP_CENTRAL_ADDR_RESOL (*C macro*), 247
ESP_GATT_UUID_GAP_DEVICE_NAME (*C macro*), 246
ESP_GATT_UUID_GAP_ICON (*C macro*), 247
ESP_GATT_UUID_GAP_PREF_CONN_PARAM (*C macro*), 247
ESP_GATT_UUID_GATT_SRV_CHGD (*C macro*), 247
ESP_GATT_UUID_GLUCOSE_SVC (*C macro*), 246
ESP_GATT_UUID_GM_CONTEXT (*C macro*), 247
ESP_GATT_UUID_GM_CONTROL_POINT (*C macro*), 247
ESP_GATT_UUID_GM_FEATURE (*C macro*), 247
ESP_GATT_UUID_GM_MEASUREMENT (*C macro*), 247
ESP_GATT_UUID_HEALTH_THERMOM_SVC (*C macro*), 246
ESP_GATT_UUID_HEART_RATE_SVC (*C macro*), 246
ESP_GATT_UUID_HID_BT_KB_INPUT (*C macro*), 247
ESP_GATT_UUID_HID_BT_KB_OUTPUT (*C macro*), 247
ESP_GATT_UUID_HID_BT_MOUSE_INPUT (*C macro*), 247
ESP_GATT_UUID_HID_CONTROL_POINT (*C macro*), 247
ESP_GATT_UUID_HID_INFORMATION (*C macro*), 247
ESP_GATT_UUID_HID_PROTO_MODE (*C macro*), 247
ESP_GATT_UUID_HID_REPORT (*C macro*), 247
ESP_GATT_UUID_HID_REPORT_MAP (*C macro*), 247
ESP_GATT_UUID_HID_SVC (*C macro*), 246
ESP_GATT_UUID_HW_VERSION_STR (*C macro*), 247
ESP_GATT_UUID_IEEE_DATA (*C macro*), 247
ESP_GATT_UUID_IMMEDIATE_ALERT_SVC (*C macro*), 245
ESP_GATT_UUID_INCLUDE_SERVICE (*C macro*), 246
ESP_GATT_UUID_LINK_LOSS_SVC (*C macro*), 245
ESP_GATT_UUID_LOCAL_TIME_INFO (*C macro*), 247
ESP_GATT_UUID_LOCATION_AND_NAVIGATION_SVC (*C macro*), 246
ESP_GATT_UUID_MANU_NAME (*C macro*), 247
ESP_GATT_UUID_MODEL_NUMBER_STR (*C macro*), 247
ESP_GATT_UUID_NEXT_DST_CHANGE_SVC (*C macro*), 245
ESP_GATT_UUID_NUM_DIGITALS_DESCR (*C macro*), 246
ESP_GATT_UUID_NW_STATUS (*C macro*), 247
ESP_GATT_UUID_NW_TRIGGER (*C macro*), 247
ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC (*C macro*), 246
ESP_GATT_UUID_PNP_ID (*C macro*), 247
ESP_GATT_UUID_PRI_SERVICE (*C macro*), 246
ESP_GATT_UUID_REF_TIME_INFO (*C macro*), 247
ESP_GATT_UUID_REF_TIME_UPDATE_SVC (*C macro*), 245
ESP_GATT_UUID_RINGER_CP (*C macro*), 247
ESP_GATT_UUID_RINGER_SETTING (*C macro*), 247
ESP_GATT_UUID_RPT_REF_DESCR (*C macro*), 246
ESP_GATT_UUID_RSC_FEATURE (*C macro*), 248
ESP_GATT_UUID_RSC_MEASUREMENT (*C macro*), 248
ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC (*C macro*), 246
ESP_GATT_UUID_SC_CONTROL_POINT (*C macro*), 248
ESP_GATT_UUID_SCAN_INT_WINDOW (*C macro*), 248
ESP_GATT_UUID_SCAN_PARAMETERS_SVC (*C macro*), 246
ESP_GATT_UUID_SCAN_REFRESH (*C macro*), 248
ESP_GATT_UUID_SEC_SERVICE (*C macro*), 246
ESP_GATT_UUID_SENSOR_LOCATION (*C macro*), 248
ESP_GATT_UUID_SERIAL_NUMBER_STR (*C macro*), 247
ESP_GATT_UUID_SW_VERSION_STR (*C macro*), 247
ESP_GATT_UUID_SYSTEM_ID (*C macro*), 247

ESP_GATT_UUID_TIME_TRIGGER_DESCR (*C macro*), 246
ESP_GATT_UUID_TX_POWER_LEVEL (*C macro*), 247
ESP_GATT_UUID_TX_POWER_SVC (*C macro*), 245
ESP_GATT_UUID_USER_DATA_SVC (*C macro*), 246
ESP_GATT_UUID_VALUE_TRIGGER_DESCR (*C macro*), 246
ESP_GATT_UUID_WEIGHT_SCALE_SVC (*C macro*), 246
esp_gatt_value_t (*C++ class*), 243
esp_gatt_value_t::auth_req (*C++ member*), 244
esp_gatt_value_t::handle (*C++ member*), 243
esp_gatt_value_t::len (*C++ member*), 244
esp_gatt_value_t::offset (*C++ member*), 243
esp_gatt_value_t::value (*C++ member*), 243
ESP_GATT_WRITE_NOT_PERMIT (*C++ enumerator*), 249
ESP_GATT_WRITE_TYPE_NO_RSP (*C++ enumerator*), 251
ESP_GATT_WRITE_TYPE_RSP (*C++ enumerator*), 251
esp_gatt_write_type_t (*C++ type*), 251
ESP_GATT_WRONG_STATE (*C++ enumerator*), 250
ESP_GATTC_ACL_EVT (*C++ enumerator*), 283
ESP_GATTC_ADV_DATA_EVT (*C++ enumerator*), 283
ESP_GATTC_ADV_VSC_EVT (*C++ enumerator*), 284
ESP_GATTC_BTH_SCAN_CFG_EVT (*C++ enumerator*), 284
ESP_GATTC_BTH_SCAN_DIS_EVT (*C++ enumerator*), 284
ESP_GATTC_BTH_SCAN_ENB_EVT (*C++ enumerator*), 284
ESP_GATTC_BTH_SCAN_PARAM_EVT (*C++ enumerator*), 284
ESP_GATTC_BTH_SCAN_RD_EVT (*C++ enumerator*), 284
ESP_GATTC_BTH_SCAN_THR_EVT (*C++ enumerator*), 284
ESP_GATTC_CANCEL_OPEN_EVT (*C++ enumerator*), 283
esp_gattc_cb_event_t (*C++ type*), 282
esp_gattc_cb_t (*C++ type*), 282
ESP_GATTC_CFG_MTU_EVT (*C++ enumerator*), 283
esp_gattc_char_elem_t (*C++ class*), 245
esp_gattc_char_elem_t::char_handle (*C++ member*), 245
esp_gattc_char_elem_t::properties (*C++ member*), 245
esp_gattc_char_elem_t::uuid (*C++ member*), 245
ESP_GATTC_CLOSE_EVT (*C++ enumerator*), 283
ESP_GATTC_CONGEST_EVT (*C++ enumerator*), 283
ESP_GATTC_CONNECT_EVT (*C++ enumerator*), 284
esp_gattc_db_elem_t (*C++ class*), 244
esp_gattc_db_elem_t::attribute_handle (*C++ member*), 244
esp_gattc_db_elem_t::end_handle (*C++ member*), 244
esp_gattc_db_elem_t::properties (*C++ member*), 244
esp_gattc_db_elem_t::start_handle (*C++ member*), 244
esp_gattc_db_elem_t::type (*C++ member*), 244
esp_gattc_db_elem_t::uuid (*C++ member*), 244
esp_gattc_descr_elem_t (*C++ class*), 245
esp_gattc_descr_elem_t::handle (*C++ member*), 245
esp_gattc_descr_elem_t::uuid (*C++ member*), 245
ESP_GATTC_DISCONNECT_EVT (*C++ enumerator*), 284
ESP_GATTC_ENC_CMPL_CB_EVT (*C++ enumerator*), 283
ESP_GATTC_EXEC_EVT (*C++ enumerator*), 283
ESP_GATTC_GET_ADDR_LIST_EVT (*C++ enumerator*), 284
esp_gattc_incl_svc_elem_t (*C++ class*), 245
esp_gattc_incl_svc_elem_t::handle (*C++ member*), 245
esp_gattc_incl_svc_elem_t::incl_srvc_e_handle (*C++ member*), 245
esp_gattc_incl_svc_elem_t::incl_srvc_s_handle (*C++ member*), 245
esp_gattc_incl_svc_elem_t::uuid (*C++ member*), 245
ESP_GATTC_MULT_ADV_DATA_EVT (*C++ enumerator*), 283
ESP_GATTC_MULT_ADV_DIS_EVT (*C++ enumerator*), 283
ESP_GATTC_MULT_ADV_ENB_EVT (*C++ enumerator*), 283
ESP_GATTC_MULT_ADV_UPD_EVT (*C++ enumerator*), 283
esp_gattc_multi_t (*C++ class*), 244
esp_gattc_multi_t::handles (*C++ member*), 244
esp_gattc_multi_t::num_attr (*C++ member*), 244
ESP_GATTC_NOTIFY_EVT (*C++ enumerator*), 283
ESP_GATTC_OPEN_EVT (*C++ enumerator*), 282
ESP_GATTC_PREP_WRITE_EVT (*C++ enumerator*), 283
ESP_GATTC_QUEUE_FULL_EVT (*C++ enumerator*), 284
ESP_GATTC_READ_CHAR_EVT (*C++ enumerator*),

283
ESP_GATTC_READ_DESCR_EVT (*C++ enumerator*), 283
ESP_GATTC_READ_MULTIPLE_EVT (*C++ enumerator*), 284
ESP_GATTC_REG_EVT (*C++ enumerator*), 282
ESP_GATTC_REG_FOR_NOTIFY_EVT (*C++ enumerator*), 284
ESP_GATTC_SCAN_FLT_CFG_EVT (*C++ enumerator*), 284
ESP_GATTC_SCAN_FLT_PARAM_EVT (*C++ enumerator*), 284
ESP_GATTC_SCAN_FLT_STATUS_EVT (*C++ enumerator*), 284
ESP_GATTC_SEARCH_CMPL_EVT (*C++ enumerator*), 283
ESP_GATTC_SEARCH_RES_EVT (*C++ enumerator*), 283
esp_gattc_service_elem_t (*C++ class*), 244
esp_gattc_service_elem_t::end_handle (*C++ member*), 244
esp_gattc_service_elem_t::is_primary (*C++ member*), 244
esp_gattc_service_elem_t::start_handle (*C++ member*), 244
esp_gattc_service_elem_t::uuid (*C++ member*), 244
ESP_GATTC_SET_ASSOC_EVT (*C++ enumerator*), 284
ESP_GATTC_SRVC_CHG_EVT (*C++ enumerator*), 283
ESP_GATTC_UNREG_EVT (*C++ enumerator*), 282
ESP_GATTC_UNREG_FOR_NOTIFY_EVT (*C++ enumerator*), 284
ESP_GATTC_WRITE_CHAR_EVT (*C++ enumerator*), 283
ESP_GATTC_WRITE_DESCR_EVT (*C++ enumerator*), 283
ESP_GATTS_ADD_CHAR_DESCR_EVT (*C++ enumerator*), 265
ESP_GATTS_ADD_CHAR_EVT (*C++ enumerator*), 265
ESP_GATTS_ADD_INCL_SRVC_EVT (*C++ enumerator*), 265
esp_gatts_attr_db_t (*C++ class*), 242
esp_gatts_attr_db_t::att_desc (*C++ member*), 243
esp_gatts_attr_db_t::attr_control (*C++ member*), 243
ESP_GATTS_CANCEL_OPEN_EVT (*C++ enumerator*), 265
esp_gatts_cb_event_t (*C++ type*), 264
esp_gatts_cb_t (*C++ type*), 264
ESP_GATTS_CLOSE_EVT (*C++ enumerator*), 265
ESP_GATTS_CONF_EVT (*C++ enumerator*), 265
ESP_GATTS_CONGEST_EVT (*C++ enumerator*), 265
ESP_GATTS_CONNECT_EVT (*C++ enumerator*), 265
ESP_GATTS_CREAT_ATTR_TAB_EVT (*C++ enumerator*), 265
ESP_GATTS_CREATE_EVT (*C++ enumerator*), 265
ESP_GATTS_DELETE_EVT (*C++ enumerator*), 265
ESP_GATTS_DISCONNECT_EVT (*C++ enumerator*), 265
ESP_GATTS_EXEC_WRITE_EVT (*C++ enumerator*), 264
esp_gatts_incl128_svc_desc_t (*C++ class*), 243
esp_gatts_incl128_svc_desc_t::end_hdl (*C++ member*), 243
esp_gatts_incl128_svc_desc_t::start_hdl (*C++ member*), 243
esp_gatts_incl_svc_desc_t (*C++ class*), 243
esp_gatts_incl_svc_desc_t::end_hdl (*C++ member*), 243
esp_gatts_incl_svc_desc_t::start_hdl (*C++ member*), 243
esp_gatts_incl_svc_desc_t::uuid (*C++ member*), 243
ESP_GATTS_LISTEN_EVT (*C++ enumerator*), 265
ESP_GATTS_MTU_EVT (*C++ enumerator*), 265
ESP_GATTS_OPEN_EVT (*C++ enumerator*), 265
ESP_GATTS_READ_EVT (*C++ enumerator*), 264
ESP_GATTS_REG_EVT (*C++ enumerator*), 264
ESP_GATTS_RESPONSE_EVT (*C++ enumerator*), 265
ESP_GATTS_SET_ATTR_VAL_EVT (*C++ enumerator*), 265
ESP_GATTS_START_EVT (*C++ enumerator*), 265
ESP_GATTS_STOP_EVT (*C++ enumerator*), 265
ESP_GATTS_UNREG_EVT (*C++ enumerator*), 265
ESP_GATTS_WRITE_EVT (*C++ enumerator*), 264
esp_gcov_dump (*C++ function*), 818
ESP_HF_AT_RESPONSE_CODE_BLACKLISTED (*C++ enumerator*), 334
ESP_HF_AT_RESPONSE_CODE_BUSY (*C++ enumerator*), 334
ESP_HF_AT_RESPONSE_CODE_CME (*C++ enumerator*), 334
ESP_HF_AT_RESPONSE_CODE_DELAYED (*C++ enumerator*), 334
ESP_HF_AT_RESPONSE_CODE_ERR (*C++ enumerator*), 334
ESP_HF_AT_RESPONSE_CODE_NO_ANSWER (*C++ enumerator*), 334
ESP_HF_AT_RESPONSE_CODE_NO_CARRIER (*C++ enumerator*), 334
ESP_HF_AT_RESPONSE_CODE_OK (*C++ enumerator*), 334
esp_hf_at_response_code_t (*C++ type*), 334
ESP_HF_BTRH_CMD_ACCEPT (*C++ enumerator*), 334
ESP_HF_BTRH_CMD_HOLD (*C++ enumerator*), 334

ESP_HF_BTRH_CMD_REJECT (*C++ enumerator*), 334
esp_hf_btrh_cmd_t (*C++ type*), 334
ESP_HF_BTRH_STATUS_ACCEPTED (*C++ enumerator*), 334
ESP_HF_BTRH_STATUS_HELD (*C++ enumerator*), 333
ESP_HF_BTRH_STATUS_REJECTED (*C++ enumerator*), 334
esp_hf_btrh_status_t (*C++ type*), 333
ESP_HF_CALL_ADDR_TYPE_INTERNATIONAL (*C++ enumerator*), 333
esp_hf_call_addr_type_t (*C++ type*), 333
ESP_HF_CALL_ADDR_TYPE_UNKNOWN (*C++ enumerator*), 333
ESP_HF_CALL_HELD_STATUS_HELD (*C++ enumerator*), 332
ESP_HF_CALL_HELD_STATUS_HELD_AND_ACTIVE (*C++ enumerator*), 332
ESP_HF_CALL_HELD_STATUS_NONE (*C++ enumerator*), 332
esp_hf_call_held_status_t (*C++ type*), 332
ESP_HF_CALL_SETUP_STATUS_INCOMING (*C++ enumerator*), 332
ESP_HF_CALL_SETUP_STATUS_NONE (*C++ enumerator*), 332
ESP_HF_CALL_SETUP_STATUS_OUTGOING_ALERTING (*C++ enumerator*), 332
ESP_HF_CALL_SETUP_STATUS_OUTGOING_DIALING (*C++ enumerator*), 332
esp_hf_call_setup_status_t (*C++ type*), 331
ESP_HF_CALL_STATUS_CALL_IN_PROGRESS (*C++ enumerator*), 331
ESP_HF_CALL_STATUS_NO_CALLS (*C++ enumerator*), 331
esp_hf_call_status_t (*C++ type*), 331
ESP_HF_CHLD_TYPE_HOLD_ACC (*C++ enumerator*), 335
ESP_HF_CHLD_TYPE_MERGE (*C++ enumerator*), 335
ESP_HF_CHLD_TYPE_MERGE_DETACH (*C++ enumerator*), 335
ESP_HF_CHLD_TYPE_PRIV_X (*C++ enumerator*), 335
ESP_HF_CHLD_TYPE_REL (*C++ enumerator*), 335
ESP_HF_CHLD_TYPE_REL_ACC (*C++ enumerator*), 335
ESP_HF_CHLD_TYPE_REL_X (*C++ enumerator*), 335
esp_hf_chld_type_t (*C++ type*), 334
esp_hf_client_answer_call (*C++ function*), 340
ESP_HF_CLIENT_AT_RESPONSE_EVT (*C++ enumerator*), 350
ESP_HF_CLIENT_AUDIO_STATE_CONNECTED (*C++ enumerator*), 349
ESP_HF_CLIENT_AUDIO_STATE_CONNECTED_MSBC (*C++ enumerator*), 349
ESP_HF_CLIENT_AUDIO_STATE_CONNECTING (*C++ enumerator*), 348
ESP_HF_CLIENT_AUDIO_STATE_DISCONNECTED (*C++ enumerator*), 348
ESP_HF_CLIENT_AUDIO_STATE_EVT (*C++ enumerator*), 349
esp_hf_client_audio_state_t (*C++ type*), 348
ESP_HF_CLIENT_BINP_EVT (*C++ enumerator*), 350
ESP_HF_CLIENT_BSIR_EVT (*C++ enumerator*), 350
ESP_HF_CLIENT_BTRH_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_BVRA_EVT (*C++ enumerator*), 349
esp_hf_client_cb_event_t (*C++ type*), 349
esp_hf_client_cb_param_t (*C++ type*), 342
esp_hf_client_cb_param_t::at_response (*C++ member*), 343
esp_hf_client_cb_param_t::audio_stat (*C++ member*), 342
esp_hf_client_cb_param_t::battery_level (*C++ member*), 342
esp_hf_client_cb_param_t::binp (*C++ member*), 343
esp_hf_client_cb_param_t::bsir (*C++ member*), 343
esp_hf_client_cb_param_t::btrh (*C++ member*), 343
esp_hf_client_cb_param_t::bvra (*C++ member*), 342
esp_hf_client_cb_param_t::call (*C++ member*), 342
esp_hf_client_cb_param_t::call_held (*C++ member*), 343
esp_hf_client_cb_param_t::call_setup (*C++ member*), 342
esp_hf_client_cb_param_t::ccwa (*C++ member*), 343
esp_hf_client_cb_param_t::clcc (*C++ member*), 343
esp_hf_client_cb_param_t::clip (*C++ member*), 343
esp_hf_client_cb_param_t::cnum (*C++ member*), 343
esp_hf_client_cb_param_t::conn_stat (*C++ member*), 342
esp_hf_client_cb_param_t::cops (*C++ member*), 342
esp_hf_client_cb_param_t::hf_client_at_response_par (*C++ class*), 343
esp_hf_client_cb_param_t::hf_client_at_response_par (*C++ member*), 343
esp_hf_client_cb_param_t::hf_client_at_response_par (*C++ member*), 343
esp_hf_client_cb_param_t::hf_client_audio_stat_para (*C++ class*), 343

esp_hf_client_cb_param_t::hf_client_audio (C++ member), 343
esp_hf_client_cb_param_t::hf_client_audio (C++ member), 343
esp_hf_client_cb_param_t::hf_client_battery (C++ class), 343
esp_hf_client_cb_param_t::hf_client_battery (C++ member), 344
esp_hf_client_cb_param_t::hf_client_binp (C++ class), 344
esp_hf_client_cb_param_t::hf_client_binp (C++ member), 344
esp_hf_client_cb_param_t::hf_client_bsir (C++ class), 344
esp_hf_client_cb_param_t::hf_client_bsir (C++ member), 344
esp_hf_client_cb_param_t::hf_client_btrh (C++ class), 344
esp_hf_client_cb_param_t::hf_client_btrh (C++ member), 344
esp_hf_client_cb_param_t::hf_client_bvra (C++ class), 344
esp_hf_client_cb_param_t::hf_client_bvra (C++ member), 344
esp_hf_client_cb_param_t::hf_client_call_held (C++ class), 344
esp_hf_client_cb_param_t::hf_client_call_held (C++ member), 344
esp_hf_client_cb_param_t::hf_client_call_setup (C++ class), 344
esp_hf_client_cb_param_t::hf_client_call_setup (C++ member), 345
esp_hf_client_cb_param_t::hf_client_call (C++ class), 345
esp_hf_client_cb_param_t::hf_client_call (C++ member), 345
esp_hf_client_cb_param_t::hf_client_ccwa (C++ class), 345
esp_hf_client_cb_param_t::hf_client_ccwa (C++ member), 345
esp_hf_client_cb_param_t::hf_client_clcc (C++ class), 345
esp_hf_client_cb_param_t::hf_client_clcc (C++ member), 345
esp_hf_client_cb_param_t::hf_client_clcc (C++ member), 345
esp_hf_client_cb_param_t::hf_client_clcc (C++ member), 345
esp_hf_client_cb_param_t::hf_client_clip (C++ class), 345
esp_hf_client_cb_param_t::hf_client_clip (C++ member), 345
esp_hf_client_cb_param_t::hf_client_cmd (C++ class), 345
esp_hf_client_cb_param_t::hf_client_cmd (C++ class), 345
esp_hf_client_cb_param_t::hf_client_cnum (C++ member), 346
esp_hf_client_cb_param_t::hf_client_cnum (C++ member), 346
esp_hf_client_cb_param_t::hf_client_conn_stat (C++ class), 346
esp_hf_client_cb_param_t::hf_client_conn_stat (C++ member), 346
esp_hf_client_cb_param_t::hf_client_conn_stat (C++ member), 346
esp_hf_client_cb_param_t::hf_client_conn_stat (C++ member), 346
esp_hf_client_cb_param_t::hf_client_current_operator (C++ class), 346
esp_hf_client_cb_param_t::hf_client_current_operator (C++ member), 346
esp_hf_client_cb_param_t::hf_client_network_roaming (C++ class), 346
esp_hf_client_cb_param_t::hf_client_network_roaming (C++ member), 346
esp_hf_client_cb_param_t::hf_client_service_availability (C++ class), 346
esp_hf_client_cb_param_t::hf_client_service_availability (C++ member), 346
esp_hf_client_cb_param_t::hf_client_signal_strength (C++ class), 346
esp_hf_client_cb_param_t::hf_client_signal_strength (C++ member), 347
esp_hf_client_cb_param_t::hf_client_volume_control (C++ class), 347
esp_hf_client_cb_param_t::hf_client_volume_control (C++ member), 347
esp_hf_client_cb_param_t::hf_client_volume_control (C++ member), 347
esp_hf_client_cb_param_t::roaming (C++ member), 342
esp_hf_client_cb_param_t::service_availability (C++ member), 342
esp_hf_client_cb_param_t::signal_strength (C++ member), 342
esp_hf_client_cb_param_t::volume_control (C++ member), 343
esp_hf_client_cb_t (C++ type), 348
ESP_HF_CLIENT_CCWA_EVT (C++ enumerator), 349
ESP_HF_CLIENT_CHLD_FEAT_HOLD_ACC (C macro), 347
ESP_HF_CLIENT_CHLD_FEAT_MERGE (C macro), 347

ESP_HF_CLIENT_CHLD_FEAT_MERGE_DETACH (*C macro*), 347
ESP_HF_CLIENT_CHLD_FEAT_PRIV_X (*C macro*), 347
ESP_HF_CLIENT_CHLD_FEAT_REL (*C macro*), 347
ESP_HF_CLIENT_CHLD_FEAT_REL_ACC (*C macro*), 347
ESP_HF_CLIENT_CHLD_FEAT_REL_X (*C macro*), 347
ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CIND_CALL_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CIND_CALL_HELD_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CIND_CALL_SETUP_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CLCC_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CLIP_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CNUM_EVT (*C++ enumerator*), 350
esp_hf_client_connect (*C++ function*), 337
esp_hf_client_connect_audio (*C++ function*), 337
ESP_HF_CLIENT_CONNECTION_STATE_CONNECTED (*C++ enumerator*), 348
ESP_HF_CLIENT_CONNECTION_STATE_CONNECTING (*C++ enumerator*), 348
ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTED (*C++ enumerator*), 348
ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTING (*C++ enumerator*), 348
ESP_HF_CLIENT_CONNECTION_STATE_EVT (*C++ enumerator*), 349
ESP_HF_CLIENT_CONNECTION_STATE_SLC_CONNECTED (*C++ enumerator*), 348
esp_hf_client_connection_state_t (*C++ type*), 348
ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT (*C++ enumerator*), 349
esp_hf_client_deinit (*C++ function*), 337
esp_hf_client_dial (*C++ function*), 339
esp_hf_client_dial_memory (*C++ function*), 339
esp_hf_client_disconnect (*C++ function*), 337
esp_hf_client_disconnect_audio (*C++ function*), 338
esp_hf_client_in_band_ring_state_t (*C++ type*), 349
ESP_HF_CLIENT_IN_BAND_RINGTONE_NOT_PROVIDED (*C++ enumerator*), 349
ESP_HF_CLIENT_IN_BAND_RINGTONE_PROVIDED (*C++ enumerator*), 349
esp_hf_client_incoming_data_cb_t (*C++ type*), 347
esp_hf_client_init (*C++ function*), 337
esp_hf_client_outgoing_data_cb_t (*C++ type*), 348
esp_hf_client_outgoing_data_ready (*C++ function*), 341
esp_hf_client_pcm_resample (*C++ function*), 342
esp_hf_client_pcm_resample_init (*C++ function*), 341
ESP_HF_CLIENT_PEER_FEAT_3WAY (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_CODEC (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_ECC (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_ECNR (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_ECS (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_EXTERR (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_INBAND (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_REJECT (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_VREC (*C macro*), 347
ESP_HF_CLIENT_PEER_FEAT_VTAG (*C macro*), 347
esp_hf_client_query_current_calls (*C++ function*), 340
esp_hf_client_query_current_operator_name (*C++ function*), 340
esp_hf_client_register_callback (*C++ function*), 336
esp_hf_client_register_data_callback (*C++ function*), 341
esp_hf_client_reject_call (*C++ function*), 340
esp_hf_client_request_last_voice_tag_number (*C++ function*), 341
esp_hf_client_retrieve_subscriber_info (*C++ function*), 340
ESP_HF_CLIENT_RING_IND_EVT (*C++ enumerator*), 350
esp_hf_client_send_btrh_cmd (*C++ function*), 339
esp_hf_client_send_chld_cmd (*C++ function*), 339
esp_hf_client_send_dtmf (*C++ function*), 341
esp_hf_client_start_voice_recognition (*C++ function*), 338
esp_hf_client_stop_voice_recognition (*C++ function*), 338

ESP_HF_CLIENT_VOLUME_CONTROL_EVT (*C++ enumerator*), 350
esp_hf_client_volume_update (*C++ function*), 338
ESP_HF_CME_AG_FAILURE (*C++ enumerator*), 335
ESP_HF_CME_DIAL_STRING_TOO_LONG (*C++ enumerator*), 336
esp_hf_cme_err_t (*C++ type*), 335
ESP_HF_CME_INCORRECT_PASSWORD (*C++ enumerator*), 335
ESP_HF_CME_INVALID_CHARACTERS_IN_DIAL_STRING (*C++ enumerator*), 336
ESP_HF_CME_INVALID_CHARACTERS_IN_TEXT_STRING (*C++ enumerator*), 336
ESP_HF_CME_INVALID_INDEX (*C++ enumerator*), 336
ESP_HF_CME_MEMEORY_FAILURE (*C++ enumerator*), 336
ESP_HF_CME_MEMEORY_FULL (*C++ enumerator*), 336
ESP_HF_CME_NETWORK_NOT_ALLOWED (*C++ enumerator*), 336
ESP_HF_CME_NETWORK_TIMEOUT (*C++ enumerator*), 336
ESP_HF_CME_NO_CONNECTION_TO_PHONE (*C++ enumerator*), 335
ESP_HF_CME_NO_NETWORK_SERVICE (*C++ enumerator*), 336
ESP_HF_CME_OPERATION_NOT_ALLOWED (*C++ enumerator*), 335
ESP_HF_CME_OPERATION_NOT_SUPPORTED (*C++ enumerator*), 335
ESP_HF_CME_PH_SIM_PIN_REQUIRED (*C++ enumerator*), 335
ESP_HF_CME_SIM_BUSY (*C++ enumerator*), 335
ESP_HF_CME_SIM_FAILURE (*C++ enumerator*), 335
ESP_HF_CME_SIM_NOT_INSERTED (*C++ enumerator*), 335
ESP_HF_CME_SIM_PIN2_REQUIRED (*C++ enumerator*), 335
ESP_HF_CME_SIM_PIN_REQUIRED (*C++ enumerator*), 335
ESP_HF_CME_SIM_PUK2_REQUIRED (*C++ enumerator*), 336
ESP_HF_CME_SIM_PUK_REQUIRED (*C++ enumerator*), 335
ESP_HF_CME_TEXT_STRING_TOO_LONG (*C++ enumerator*), 336
ESP_HF_CURRENT_CALL_DIRECTION_INCOMING (*C++ enumerator*), 333
ESP_HF_CURRENT_CALL_DIRECTION_OUTGOING (*C++ enumerator*), 333
esp_hf_current_call_direction_t (*C++ type*), 333
ESP_HF_CURRENT_CALL_MODE_DATA (*C++ enumerator*), 333
ESP_HF_CURRENT_CALL_MODE_FAX (*C++ enumerator*), 333
esp_hf_current_call_mode_t (*C++ type*), 333
ESP_HF_CURRENT_CALL_MODE_VOICE (*C++ enumerator*), 333
ESP_HF_CURRENT_CALL_MPTY_TYPE_MULTI (*C++ enumerator*), 333
ESP_HF_CURRENT_CALL_MPTY_TYPE_SINGLE (*C++ enumerator*), 333
esp_hf_current_call_mpty_type_t (*C++ type*), 333
ESP_HF_CURRENT_CALL_STATUS_ACTIVE (*C++ enumerator*), 332
ESP_HF_CURRENT_CALL_STATUS_ALERTING (*C++ enumerator*), 332
ESP_HF_CURRENT_CALL_STATUS_DIALING (*C++ enumerator*), 332
ESP_HF_CURRENT_CALL_STATUS_HELD (*C++ enumerator*), 332
ESP_HF_CURRENT_CALL_STATUS_HELD_BY_RESP_HOLD (*C++ enumerator*), 332
ESP_HF_CURRENT_CALL_STATUS_INCOMING (*C++ enumerator*), 332
esp_hf_current_call_status_t (*C++ type*), 332
ESP_HF_CURRENT_CALL_STATUS_WAITING (*C++ enumerator*), 332
ESP_HF_ROAMING_STATUS_ACTIVE (*C++ enumerator*), 331
ESP_HF_ROAMING_STATUS_INACTIVE (*C++ enumerator*), 331
esp_hf_roaming_status_t (*C++ type*), 331
ESP_HF_SERVICE_AVAILABILITY_STATUS_AVAILABLE (*C++ enumerator*), 332
esp_hf_service_availability_status_t (*C++ type*), 332
ESP_HF_SERVICE_AVAILABILITY_STATUS_UNAVAILABLE (*C++ enumerator*), 332
ESP_HF_SUBSCRIBER_SERVICE_TYPE_FAX (*C++ enumerator*), 333
esp_hf_subscriber_service_type_t (*C++ type*), 333
ESP_HF_SUBSCRIBER_SERVICE_TYPE_UNKNOWN (*C++ enumerator*), 333
ESP_HF_SUBSCRIBER_SERVICE_TYPE_VOICE (*C++ enumerator*), 333
ESP_HF_VOLUME_CONTROL_TARGET_MIC (*C++ enumerator*), 331
ESP_HF_VOLUME_CONTROL_TARGET_SPK (*C++ enumerator*), 331
esp_hf_volume_control_target_t (*C++ type*), 331

ESP_HF_VR_STATE_DISABLED (*C++ enumerator*), 334
ESP_HF_VR_STATE_ENABLED (*C++ enumerator*), 334
esp_hf_vr_state_t (*C++ type*), 334
esp_http_client_auth_type_t (*C++ type*), 598
esp_http_client_cleanup (*C++ function*), 595
esp_http_client_close (*C++ function*), 594
esp_http_client_config_t (*C++ class*), 596
esp_http_client_config_t::auth_type (*C++ member*), 596
esp_http_client_config_t::buffer_size (*C++ member*), 596
esp_http_client_config_t::cert_pem (*C++ member*), 596
esp_http_client_config_t::disable_auto_redirection (*C++ member*), 596
esp_http_client_config_t::event_handler (*C++ member*), 596
esp_http_client_config_t::host (*C++ member*), 596
esp_http_client_config_t::max_redirection_count (*C++ member*), 596
esp_http_client_config_t::method (*C++ member*), 596
esp_http_client_config_t::password (*C++ member*), 596
esp_http_client_config_t::path (*C++ member*), 596
esp_http_client_config_t::port (*C++ member*), 596
esp_http_client_config_t::query (*C++ member*), 596
esp_http_client_config_t::timeout_ms (*C++ member*), 596
esp_http_client_config_t::transport_type (*C++ member*), 596
esp_http_client_config_t::url (*C++ member*), 596
esp_http_client_config_t::user_data (*C++ member*), 596
esp_http_client_config_t::username (*C++ member*), 596
esp_http_client_delete_header (*C++ function*), 593
esp_http_client_event (*C++ class*), 595
esp_http_client_event::client (*C++ member*), 595
esp_http_client_event::data (*C++ member*), 595
esp_http_client_event::data_len (*C++ member*), 595
esp_http_client_event::event_id (*C++ member*), 595
esp_http_client_event::header_key (*C++ member*), 596
esp_http_client_event::header_value (*C++ member*), 596
esp_http_client_event::user_data (*C++ member*), 595
esp_http_client_event_handle_t (*C++ type*), 597
esp_http_client_event_id_t (*C++ type*), 597
esp_http_client_event_t (*C++ type*), 597
esp_http_client_fetch_headers (*C++ function*), 593
esp_http_client_get_content_length (*C++ function*), 594
esp_http_client_get_post_field (*C++ function*), 592
esp_http_client_get_status_code (*C++ function*), 594
esp_http_client_get_transport_type (*C++ function*), 595
esp_http_client_handle_t (*C++ type*), 597
esp_http_client_init (*C++ function*), 591
esp_http_client_is_chunked_response (*C++ function*), 594
esp_http_client_method_t (*C++ type*), 598
esp_http_client_open (*C++ function*), 593
esp_http_client_perform (*C++ function*), 591
esp_http_client_read (*C++ function*), 594
esp_http_client_set_header (*C++ function*), 592
esp_http_client_set_method (*C++ function*), 592
esp_http_client_set_post_field (*C++ function*), 592
esp_http_client_set_url (*C++ function*), 592
esp_http_client_transport_t (*C++ type*), 598
esp_http_client_write (*C++ function*), 593
esp_https_ota (*C++ function*), 838
esp_int_wdt_init (*C++ function*), 799
esp_intr_alloc (*C++ function*), 794
esp_intr_alloc_intrstatus (*C++ function*), 794
esp_intr_disable (*C++ function*), 796
esp_intr_enable (*C++ function*), 796
ESP_INTR_FLAG_EDGE (*C macro*), 797
ESP_INTR_FLAG_HIGH (*C macro*), 797
ESP_INTR_FLAG_INTRDISABLED (*C macro*), 797
ESP_INTR_FLAG_IRAM (*C macro*), 797
ESP_INTR_FLAG_LEVEL1 (*C macro*), 797
ESP_INTR_FLAG_LEVEL2 (*C macro*), 797
ESP_INTR_FLAG_LEVEL3 (*C macro*), 797
ESP_INTR_FLAG_LEVEL4 (*C macro*), 797
ESP_INTR_FLAG_LEVEL5 (*C macro*), 797
ESP_INTR_FLAG_LEVEL6 (*C macro*), 797

ESP_INTR_FLAG_LEVELMASK (*C macro*), 797
ESP_INTR_FLAG_LOWMED (*C macro*), 797
ESP_INTR_FLAG_NMI (*C macro*), 797
ESP_INTR_FLAG_SHARED (*C macro*), 797
esp_intr_free (*C++ function*), 795
esp_intr_get_cpu (*C++ function*), 795
esp_intr_get_intno (*C++ function*), 796
esp_intr_mark_shared (*C++ function*), 793
esp_intr_noniram_disable (*C++ function*), 796
esp_intr_noniram_enable (*C++ function*), 796
esp_intr_reserve (*C++ function*), 794
esp_intr_set_in_iram (*C++ function*), 796
ESP_IO_CAP_IN (*C macro*), 233
ESP_IO_CAP_IO (*C macro*), 233
ESP_IO_CAP_KBDISP (*C macro*), 233
ESP_IO_CAP_NONE (*C macro*), 233
ESP_IO_CAP_OUT (*C macro*), 233
esp_ipc_call (*C++ function*), 802
esp_ipc_call_blocking (*C++ function*), 803
ESP_LE_AUTH_BOND (*C macro*), 233
ESP_LE_AUTH_NO_BOND (*C macro*), 233
ESP_LE_AUTH_REQ_BOND_MITM (*C macro*), 233
ESP_LE_AUTH_REQ_MITM (*C macro*), 233
ESP_LE_AUTH_REQ_SC_BOND (*C macro*), 233
ESP_LE_AUTH_REQ_SC_MITM (*C macro*), 233
ESP_LE_AUTH_REQ_SC_MITM_BOND (*C macro*), 233
ESP_LE_AUTH_REQ_SC_ONLY (*C macro*), 233
ESP_LE_KEY_LCSRK (*C macro*), 233
ESP_LE_KEY_LENC (*C macro*), 233
ESP_LE_KEY_LID (*C macro*), 233
ESP_LE_KEY_LLK (*C macro*), 233
ESP_LE_KEY_NONE (*C macro*), 233
ESP_LE_KEY_PCSRK (*C macro*), 233
ESP_LE_KEY_PENC (*C macro*), 233
ESP_LE_KEY_PID (*C macro*), 233
ESP_LE_KEY_PLK (*C macro*), 233
esp_light_sleep_start (*C++ function*), 829
ESP_LINE_ENDINGS_CR (*C++ enumerator*), 653
ESP_LINE_ENDINGS_CRLF (*C++ enumerator*), 653
ESP_LINE_ENDINGS_LF (*C++ enumerator*), 653
esp_line_endings_t (*C++ type*), 653
esp_link_key (*C++ type*), 210
ESP_LOG_BUFFER_CHAR (*C macro*), 811
ESP_LOG_BUFFER_CHAR_LEVEL (*C macro*), 810
ESP_LOG_BUFFER_HEX (*C macro*), 811
ESP_LOG_BUFFER_HEX_LEVEL (*C macro*), 810
ESP_LOG_BUFFER_HEXDUMP (*C macro*), 811
ESP_LOG_DEBUG (*C++ enumerator*), 813
ESP_LOG_EARLY_IMPL (*C macro*), 812
esp_log_early_timestamp (*C++ function*), 810
ESP_LOG_ERROR (*C++ enumerator*), 813
ESP_LOG_INFO (*C++ enumerator*), 813
ESP_LOG_LEVEL (*C macro*), 812
ESP_LOG_LEVEL_LOCAL (*C macro*), 812
esp_log_level_set (*C++ function*), 809
esp_log_level_t (*C++ type*), 813
ESP_LOG_NONE (*C++ enumerator*), 813
esp_log_set_vprintf (*C++ function*), 810
esp_log_timestamp (*C++ function*), 810
ESP_LOG_VERBOSE (*C++ enumerator*), 813
ESP_LOG_WARN (*C++ enumerator*), 813
esp_log_write (*C++ function*), 810
ESP_LOGD (*C macro*), 812
ESP_LOGE (*C macro*), 812
ESP_LOGI (*C macro*), 812
ESP_LOGV (*C macro*), 812
ESP_LOGW (*C macro*), 812
esp_mesh_allow_root_conflicts (*C++ function*), 185
esp_mesh_available_txupQ_num (*C++ function*), 184
esp_mesh_deinit (*C++ function*), 174
esp_mesh_delete_group_id (*C++ function*), 185
esp_mesh_fix_root (*C++ function*), 188
esp_mesh_flush_upstream_packets (*C++ function*), 189
esp_mesh_get_ap_assoc_expire (*C++ function*), 183
esp_mesh_get_ap_authmode (*C++ function*), 180
esp_mesh_get_ap_connections (*C++ function*), 181
esp_mesh_get_capacity_num (*C++ function*), 186
esp_mesh_get_config (*C++ function*), 178
esp_mesh_get_group_list (*C++ function*), 186
esp_mesh_get_group_num (*C++ function*), 186
esp_mesh_get_id (*C++ function*), 179
esp_mesh_get_ie_crypto_key (*C++ function*), 187
esp_mesh_get_layer (*C++ function*), 181
esp_mesh_get_max_layer (*C++ function*), 180
esp_mesh_get_parent_bssid (*C++ function*), 181
esp_mesh_get_root_healing_delay (*C++ function*), 187
esp_mesh_get_router (*C++ function*), 178
esp_mesh_get_routing_table (*C++ function*), 184
esp_mesh_get_routing_table_size (*C++ function*), 183
esp_mesh_get_rx_pending (*C++ function*), 184
esp_mesh_get_self_organized (*C++ function*), 182
esp_mesh_get_subnet_nodes_list (*C++ function*), 190
esp_mesh_get_subnet_nodes_num (*C++ function*), 189

esp_mesh_get_total_node_num (*C++ function*), 183
esp_mesh_get_tx_pending (*C++ function*), 184
esp_mesh_get_type (*C++ function*), 179
esp_mesh_get_vote_percentage (*C++ function*), 183
esp_mesh_get_xon_qsize (*C++ function*), 185
esp_mesh_init (*C++ function*), 173
esp_mesh_is_my_group (*C++ function*), 186
esp_mesh_is_root (*C++ function*), 181
esp_mesh_is_root_conflicts_allowed (*C++ function*), 185
esp_mesh_is_root_fixed (*C++ function*), 188
esp_mesh_post_toDS_state (*C++ function*), 184
esp_mesh_recv (*C++ function*), 176
esp_mesh_recv_toDS (*C++ function*), 177
esp_mesh_scan_get_ap_ie_len (*C++ function*), 189
esp_mesh_scan_get_ap_record (*C++ function*), 189
esp_mesh_send (*C++ function*), 174
esp_mesh_set_ap_assoc_expire (*C++ function*), 183
esp_mesh_set_ap_authmode (*C++ function*), 180
esp_mesh_set_ap_connections (*C++ function*), 181
esp_mesh_set_ap_password (*C++ function*), 180
esp_mesh_set_capacity_num (*C++ function*), 186
esp_mesh_set_config (*C++ function*), 177
esp_mesh_set_event_cb (*C++ function*), 187
esp_mesh_set_group_id (*C++ function*), 185
esp_mesh_set_id (*C++ function*), 179
esp_mesh_set_ie_crypto_funcs (*C++ function*), 186
esp_mesh_set_ie_crypto_key (*C++ function*), 187
esp_mesh_set_max_layer (*C++ function*), 180
esp_mesh_set_parent (*C++ function*), 188
esp_mesh_set_root_healing_delay (*C++ function*), 187
esp_mesh_set_router (*C++ function*), 178
esp_mesh_set_self_organized (*C++ function*), 181
esp_mesh_set_type (*C++ function*), 179
esp_mesh_set_vote_percentage (*C++ function*), 183
esp_mesh_set_xon_qsize (*C++ function*), 185
esp_mesh_start (*C++ function*), 174
esp_mesh_stop (*C++ function*), 174
esp_mesh_waive_root (*C++ function*), 182
esp_now_add_peer (*C++ function*), 165
esp_now_deinit (*C++ function*), 163
esp_now_del_peer (*C++ function*), 165
ESP_NOW_ETH_ALEN (*C macro*), 168
esp_now_fetch_peer (*C++ function*), 166
esp_now_get_peer (*C++ function*), 166
esp_now_get_peer_num (*C++ function*), 167
esp_now_get_version (*C++ function*), 164
esp_now_init (*C++ function*), 163
esp_now_is_peer_exist (*C++ function*), 166
ESP_NOW_KEY_LEN (*C macro*), 168
ESP_NOW_MAX_DATA_LEN (*C macro*), 168
ESP_NOW_MAX_ENCRYPT_PEER_NUM (*C macro*), 168
ESP_NOW_MAX_TOTAL_PEER_NUM (*C macro*), 168
esp_now_mod_peer (*C++ function*), 166
esp_now_peer_info (*C++ class*), 167
esp_now_peer_info::channel (*C++ member*), 167
esp_now_peer_info::encrypt (*C++ member*), 167
esp_now_peer_info::ifidx (*C++ member*), 167
esp_now_peer_info::lmk (*C++ member*), 167
esp_now_peer_info::peer_addr (*C++ member*), 167
esp_now_peer_info::priv (*C++ member*), 168
esp_now_peer_info_t (*C++ type*), 169
esp_now_peer_num (*C++ class*), 168
esp_now_peer_num::encrypt_num (*C++ member*), 168
esp_now_peer_num::total_num (*C++ member*), 168
esp_now_peer_num_t (*C++ type*), 169
esp_now_recv_cb_t (*C++ type*), 169
esp_now_register_recv_cb (*C++ function*), 164
esp_now_register_send_cb (*C++ function*), 164
esp_now_send (*C++ function*), 164
esp_now_send_cb_t (*C++ type*), 169
ESP_NOW_SEND_FAIL (*C++ enumerator*), 169
esp_now_send_status_t (*C++ type*), 169
ESP_NOW_SEND_SUCCESS (*C++ enumerator*), 169
esp_now_set_pmk (*C++ function*), 167
esp_now_unregister_recv_cb (*C++ function*), 164
esp_now_unregister_send_cb (*C++ function*), 164
ESP_OK (*C macro*), 841
esp_ota_begin (*C++ function*), 834
esp_ota_end (*C++ function*), 835
esp_ota_get_boot_partition (*C++ function*), 836
esp_ota_get_next_update_partition (*C++ function*), 836
esp_ota_get_running_partition (*C++ function*), 836
esp_ota_handle_t (*C++ type*), 837
esp_ota_set_boot_partition (*C++ function*), 835

esp_ota_write (*C++ function*), 835
esp_partition_erase_range (*C++ function*), 610
esp_partition_find (*C++ function*), 608
esp_partition_find_first (*C++ function*), 609
esp_partition_get (*C++ function*), 609
esp_partition_iterator_release (*C++ function*), 609
esp_partition_iterator_t (*C++ type*), 612
esp_partition_mmap (*C++ function*), 611
esp_partition_next (*C++ function*), 609
esp_partition_read (*C++ function*), 610
ESP_PARTITION_SUBTYPE_ANY (*C++ enumerator*), 614
ESP_PARTITION_SUBTYPE_APP_FACTORY (*C++ enumerator*), 612
ESP_PARTITION_SUBTYPE_APP_OTA_0 (*C++ enumerator*), 612
ESP_PARTITION_SUBTYPE_APP_OTA_1 (*C++ enumerator*), 612
ESP_PARTITION_SUBTYPE_APP_OTA_10 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_11 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_12 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_13 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_14 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_15 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_2 (*C++ enumerator*), 612
ESP_PARTITION_SUBTYPE_APP_OTA_3 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_4 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_5 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_6 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_7 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_8 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_9 (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_MAX (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_APP_OTA_MIN (*C++ enumerator*), 612
ESP_PARTITION_SUBTYPE_APP_TEST (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_DATA_COREDUMP (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_DATA_FAT (*C++ enumerator*), 614
ESP_PARTITION_SUBTYPE_DATA_NVS (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_DATA_NVS_KEYS (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_DATA_OTA (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_DATA_PHY (*C++ enumerator*), 613
ESP_PARTITION_SUBTYPE_DATA_SPIFFS (*C++ enumerator*), 614
ESP_PARTITION_SUBTYPE_OTA (*C macro*), 612
esp_partition_subtype_t (*C++ type*), 612
esp_partition_t (*C++ class*), 611
esp_partition_t::address (*C++ member*), 611
esp_partition_t::encrypted (*C++ member*), 612
esp_partition_t::label (*C++ member*), 612
esp_partition_t::size (*C++ member*), 612
esp_partition_t::subtype (*C++ member*), 611
esp_partition_t::type (*C++ member*), 611
ESP_PARTITION_TYPE_APP (*C++ enumerator*), 612
ESP_PARTITION_TYPE_DATA (*C++ enumerator*), 612
esp_partition_type_t (*C++ type*), 612
esp_partition_verify (*C++ function*), 609
esp_partition_write (*C++ function*), 610
ESP_PD_DOMAIN_MAX (*C++ enumerator*), 829
ESP_PD_DOMAIN_RTC_FAST_MEM (*C++ enumerator*), 828
ESP_PD_DOMAIN_RTC_PERIPH (*C++ enumerator*), 828
ESP_PD_DOMAIN_RTC_SLOW_MEM (*C++ enumerator*), 828
ESP_PD_DOMAIN_XTAL (*C++ enumerator*), 828
ESP_PD_OPTION_AUTO (*C++ enumerator*), 829
ESP_PD_OPTION_OFF (*C++ enumerator*), 829
ESP_PD_OPTION_ON (*C++ enumerator*), 829
ESP_PM_APB_FREQ_MAX (*C++ enumerator*), 823
esp_pm_config_esp32_t (*C++ class*), 824
esp_pm_config_esp32_t::light_sleep_enable (*C++ member*), 824
esp_pm_config_esp32_t::max_cpu_freq (*C++ member*), 824
esp_pm_config_esp32_t::min_cpu_freq (*C++ member*), 824
esp_pm_configure (*C++ function*), 821
ESP_PM_CPU_FREQ_MAX (*C++ enumerator*), 823
esp_pm_dump_locks (*C++ function*), 823

esp_pm_lock_acquire (*C++ function*), 822
esp_pm_lock_create (*C++ function*), 821
esp_pm_lock_delete (*C++ function*), 822
esp_pm_lock_handle_t (*C++ type*), 823
esp_pm_lock_release (*C++ function*), 822
esp_pm_lock_type_t (*C++ type*), 823
ESP_PM_NO_LIGHT_SLEEP (*C++ enumerator*), 823
esp_power_level_t (*C++ type*), 207
esp_pthread_cfg_t (*C++ class*), 840
esp_pthread_cfg_t::inherit_cfg (*C++ member*), 840
esp_pthread_cfg_t::prio (*C++ member*), 840
esp_pthread_cfg_t::stack_size (*C++ member*), 840
esp_pthread_get_cfg (*C++ function*), 840
esp_pthread_set_cfg (*C++ function*), 839
ESP_PWR_LVL_N0 (*C++ enumerator*), 208
ESP_PWR_LVL_N11 (*C++ enumerator*), 208
ESP_PWR_LVL_N12 (*C++ enumerator*), 208
ESP_PWR_LVL_N14 (*C++ enumerator*), 208
ESP_PWR_LVL_N2 (*C++ enumerator*), 208
ESP_PWR_LVL_N3 (*C++ enumerator*), 208
ESP_PWR_LVL_N5 (*C++ enumerator*), 208
ESP_PWR_LVL_N6 (*C++ enumerator*), 208
ESP_PWR_LVL_N8 (*C++ enumerator*), 208
ESP_PWR_LVL_N9 (*C++ enumerator*), 208
ESP_PWR_LVL_P1 (*C++ enumerator*), 208
ESP_PWR_LVL_P3 (*C++ enumerator*), 208
ESP_PWR_LVL_P4 (*C++ enumerator*), 208
ESP_PWR_LVL_P6 (*C++ enumerator*), 208
ESP_PWR_LVL_P7 (*C++ enumerator*), 208
ESP_PWR_LVL_P9 (*C++ enumerator*), 208
esp_register_freertos_idle_hook (*C++ function*), 770
esp_register_freertos_idle_hook_for_cpu (*C++ function*), 770
esp_register_freertos_tick_hook (*C++ function*), 771
esp_register_freertos_tick_hook_for_cpu (*C++ function*), 771
ESP_SCO_DATA_PATH_HCI (*C++ enumerator*), 208
ESP_SCO_DATA_PATH_PCM (*C++ enumerator*), 208
esp_sco_data_path_t (*C++ type*), 208
esp_sleep_disable_wakeup_source (*C++ function*), 831
esp_sleep_enable_ext0_wakeup (*C++ function*), 826
esp_sleep_enable_ext1_wakeup (*C++ function*), 827
esp_sleep_enable_timer_wakeup (*C++ function*), 825
esp_sleep_enable_touchpad_wakeup (*C++ function*), 825
esp_sleep_enable_ulp_wakeup (*C++ function*), 828
esp_sleep_ext1_wakeup_mode_t (*C++ type*), 827
esp_sleep_get_ext1_wakeup_status (*C++ function*), 830
esp_sleep_get_touchpad_wakeup_status (*C++ function*), 830
esp_sleep_get_wakeup_cause (*C++ function*), 830
esp_sleep_pd_config (*C++ function*), 828
esp_sleep_pd_domain_t (*C++ type*), 828
esp_sleep_pd_option_t (*C++ type*), 829
esp_sleep_source_t (*C++ type*), 830
esp_sleep_wakeup_cause_t (*C++ type*), 830
ESP_SLEEP_WAKEUP_EXT0 (*C++ enumerator*), 830
ESP_SLEEP_WAKEUP_EXT1 (*C++ enumerator*), 830
ESP_SLEEP_WAKEUP_TIMER (*C++ enumerator*), 830
ESP_SLEEP_WAKEUP_TOUCHPAD (*C++ enumerator*), 831
ESP_SLEEP_WAKEUP_ULP (*C++ enumerator*), 831
ESP_SLEEP_WAKEUP_UNDEFINED (*C++ enumerator*), 830
esp_smartconfig_fast_mode (*C++ function*), 160
esp_smartconfig_get_version (*C++ function*), 159
esp_smartconfig_set_type (*C++ function*), 160
esp_smartconfig_start (*C++ function*), 159
esp_smartconfig_stop (*C++ function*), 159
esp_spiffs_format (*C++ function*), 665
esp_spiffs_info (*C++ function*), 665
esp_spiffs_mounted (*C++ function*), 665
ESP_SPP_BUSY (*C++ enumerator*), 330
esp_spp_cb_event_t (*C++ type*), 330
esp_spp_cb_param_t (*C++ type*), 325
esp_spp_cb_param_t::cl_init (*C++ member*), 326
esp_spp_cb_param_t::close (*C++ member*), 326
esp_spp_cb_param_t::cong (*C++ member*), 326
esp_spp_cb_param_t::data_ind (*C++ member*), 326
esp_spp_cb_param_t::disc_comp (*C++ member*), 325
esp_spp_cb_param_t::init (*C++ member*), 325
esp_spp_cb_param_t::open (*C++ member*), 326
esp_spp_cb_param_t::spp_cl_init_evt_param (*C++ class*), 326
esp_spp_cb_param_t::spp_cl_init_evt_param::handle (*C++ member*), 326
esp_spp_cb_param_t::spp_cl_init_evt_param::sec_id (*C++ member*), 326
esp_spp_cb_param_t::spp_cl_init_evt_param::status

(C++ member), 326
esp_spp_cb_param_t::spp_cl_init_evt_param (C++ class), 326
esp_spp_cb_param_t::spp_close_evt_param (C++ member), 326
esp_spp_cb_param_t::spp_close_evt_param::cap_up (C++ member), 326
esp_spp_cb_param_t::spp_close_evt_param::handle (C++ member), 326
esp_spp_cb_param_t::spp_close_evt_param::port_status (C++ member), 326
esp_spp_cb_param_t::spp_close_evt_param::status (C++ member), 326
esp_spp_cb_param_t::spp_cong_evt_param (C++ class), 326
esp_spp_cb_param_t::spp_cong_evt_param::cong (C++ member), 327
esp_spp_cb_param_t::spp_cong_evt_param::handle (C++ member), 327
esp_spp_cb_param_t::spp_cong_evt_param::status (C++ member), 327
esp_spp_cb_param_t::spp_data_ind_evt_param (C++ class), 327
esp_spp_cb_param_t::spp_data_ind_evt_param::data (C++ member), 327
esp_spp_cb_param_t::spp_data_ind_evt_param::handle (C++ member), 327
esp_spp_cb_param_t::spp_data_ind_evt_param::len (C++ member), 327
esp_spp_cb_param_t::spp_data_ind_evt_param::status (C++ member), 327
esp_spp_cb_param_t::spp_discovery_comp_evt_param (C++ class), 327
esp_spp_cb_param_t::spp_discovery_comp_evt_param::scn (C++ member), 327
esp_spp_cb_param_t::spp_discovery_comp_evt_param::service_name (C++ member), 327
esp_spp_cb_param_t::spp_init_evt_param (C++ class), 327
esp_spp_cb_param_t::spp_init_evt_param::status (C++ member), 327
esp_spp_cb_param_t::spp_open_evt_param (C++ class), 327
esp_spp_cb_param_t::spp_open_evt_param::fd (C++ member), 328
esp_spp_cb_param_t::spp_open_evt_param::handle (C++ member), 328
esp_spp_cb_param_t::spp_open_evt_param::rem_bda (C++ member), 328
esp_spp_cb_param_t::spp_open_evt_param::status (C++ member), 328
esp_spp_cb_param_t::spp_srv_open_evt_param (C++ class), 328
esp_spp_cb_param_t::spp_srv_open_evt_param::fd (C++ member), 328
esp_spp_cb_param_t::spp_srv_open_evt_param::handle (C++ member), 328
esp_spp_cb_param_t::spp_srv_open_evt_param::new_listen_handle (C++ member), 328
esp_spp_cb_param_t::spp_srv_open_evt_param::rem_bda (C++ member), 328
esp_spp_cb_param_t::spp_srv_open_evt_param::status (C++ member), 328
esp_spp_cb_param_t::spp_start_evt_param (C++ class), 328
esp_spp_cb_param_t::spp_start_evt_param::handle (C++ member), 328
esp_spp_cb_param_t::spp_start_evt_param::sec_id (C++ member), 328
esp_spp_cb_param_t::spp_start_evt_param::status (C++ member), 328
esp_spp_cb_param_t::spp_start_evt_param::use_co (C++ member), 328
esp_spp_cb_param_t::spp_write_evt_param (C++ class), 328
esp_spp_cb_param_t::spp_write_evt_param::cong (C++ member), 329
esp_spp_cb_param_t::spp_write_evt_param::handle (C++ member), 329
esp_spp_cb_param_t::spp_write_evt_param::len (C++ member), 329
esp_spp_cb_param_t::spp_write_evt_param::status (C++ member), 329
esp_spp_cb_param_t::srv_open (C++ member), 326
esp_spp_cb_param_t::start (C++ member), 326
esp_spp_cb_param_t::write (C++ member), 326
ESP_SPP_CL_INIT_EVT (C++ enumerator), 330
ESP_SPP_CLOSE_EVT (C++ enumerator), 330
ESP_SPP_CONG_EVT (C++ enumerator), 330
esp_spp_connect (C++ function), 324
ESP_SPP_DATA_IND_EVT (C++ enumerator), 330
esp_spp_deinit (C++ function), 324
esp_spp_disconnect (C++ function), 324
ESP_SPP_DISCOVERY_COMP_EVT (C++ enumerator), 330
ESP_SPP_FAILURE (C++ enumerator), 330
esp_spp_init (C++ function), 323
ESP_SPP_INIT_EVT (C++ enumerator), 330
ESP_SPP_MAX_MTU (C macro), 329
ESP_SPP_MAX_SCN (C macro), 329
ESP_SPP_MODE_CB (C++ enumerator), 330
esp_spp_mode_t (C++ type), 330
ESP_SPP_MODE_VFS (C++ enumerator), 330

ESP_SPP_NO_DATA (*C++ enumerator*), 330
ESP_SPP_NO_RESOURCE (*C++ enumerator*), 330
ESP_SPP_OPEN_EVT (*C++ enumerator*), 330
esp_spp_register_callback (*C++ function*), 323
ESP_SPP_ROLE_MASTER (*C++ enumerator*), 330
ESP_SPP_ROLE_SLAVE (*C++ enumerator*), 330
esp_spp_role_t (*C++ type*), 330
ESP_SPP_SEC_AUTHENTICATE (*C macro*), 329
ESP_SPP_SEC_AUTHORIZE (*C macro*), 329
ESP_SPP_SEC_ENCRYPT (*C macro*), 329
ESP_SPP_SEC_IN_16_DIGITS (*C macro*), 329
ESP_SPP_SEC_MITM (*C macro*), 329
ESP_SPP_SEC_MODE4_LEVEL4 (*C macro*), 329
ESP_SPP_SEC_NONE (*C macro*), 329
esp_spp_sec_t (*C++ type*), 329
ESP_SPP_SRV_OPEN_EVT (*C++ enumerator*), 331
esp_spp_start_discovery (*C++ function*), 324
ESP_SPP_START_EVT (*C++ enumerator*), 330
esp_spp_start_srv (*C++ function*), 325
esp_spp_status_t (*C++ type*), 330
ESP_SPP_SUCCESS (*C++ enumerator*), 330
esp_spp_vfs_register (*C++ function*), 325
esp_spp_write (*C++ function*), 325
ESP_SPP_WRITE_EVT (*C++ enumerator*), 331
esp_task_wdt_add (*C++ function*), 800
esp_task_wdt_deinit (*C++ function*), 800
esp_task_wdt_delete (*C++ function*), 801
esp_task_wdt_feed (*C++ function*), 801
esp_task_wdt_init (*C++ function*), 799
esp_task_wdt_reset (*C++ function*), 800
esp_task_wdt_status (*C++ function*), 801
esp_timer_cb_t (*C++ type*), 807
esp_timer_create (*C++ function*), 805
esp_timer_create_args_t (*C++ class*), 807
esp_timer_create_args_t::arg (*C++ member*), 807
esp_timer_create_args_t::callback (*C++ member*), 807
esp_timer_create_args_t::dispatch_method (*C++ member*), 807
esp_timer_create_args_t::name (*C++ member*), 807
esp_timer_deinit (*C++ function*), 805
esp_timer_delete (*C++ function*), 806
esp_timer_dispatch_t (*C++ type*), 808
esp_timer_dump (*C++ function*), 806
esp_timer_get_next_alarm (*C++ function*), 806
esp_timer_get_time (*C++ function*), 806
esp_timer_handle_t (*C++ type*), 807
esp_timer_init (*C++ function*), 804
esp_timer_start_once (*C++ function*), 805
esp_timer_start_periodic (*C++ function*), 805
esp_timer_stop (*C++ function*), 806
ESP_TIMER_TASK (*C++ enumerator*), 808
esp_tls (*C++ class*), 586
esp_tls::cacert (*C++ member*), 587
esp_tls::conf (*C++ member*), 587
esp_tls::ctr_drbg (*C++ member*), 587
esp_tls::entropy (*C++ member*), 587
esp_tls::read (*C++ member*), 587
esp_tls::server_fd (*C++ member*), 587
esp_tls::sockfd (*C++ member*), 587
esp_tls::ssl (*C++ member*), 587
esp_tls::write (*C++ member*), 587
esp_tls_cfg (*C++ class*), 586
esp_tls_cfg::alpn_protos (*C++ member*), 586
esp_tls_cfg::cacert_pem_buf (*C++ member*), 586
esp_tls_cfg::cacert_pem_bytes (*C++ member*), 586
esp_tls_cfg::non_block (*C++ member*), 586
esp_tls_cfg_t (*C++ type*), 587
esp_tls_conn_delete (*C++ function*), 586
esp_tls_conn_http_new (*C++ function*), 585
esp_tls_conn_new (*C++ function*), 585
esp_tls_conn_read (*C++ function*), 586
esp_tls_conn_write (*C++ function*), 585
esp_tls_t (*C++ type*), 587
ESP_UUID_LEN_128 (*C macro*), 210
ESP_UUID_LEN_16 (*C macro*), 210
ESP_UUID_LEN_32 (*C macro*), 210
esp_vendor_ie_cb_t (*C++ type*), 144
esp_vfs_close (*C++ function*), 649
esp_vfs_dev_uart_register (*C++ function*), 652
esp_vfs_dev_uart_set_rx_line_endings (*C++ function*), 652
esp_vfs_dev_uart_set_tx_line_endings (*C++ function*), 653
esp_vfs_dev_uart_use_driver (*C++ function*), 653
esp_vfs_dev_uart_use_nonblocking (*C++ function*), 653
esp_vfs_fat_mount_config_t (*C++ class*), 656, 660
esp_vfs_fat_mount_config_t::allocation_unit_size (*C++ member*), 656, 660
esp_vfs_fat_mount_config_t::format_if_mount_failed (*C++ member*), 656, 660
esp_vfs_fat_mount_config_t::max_files (*C++ member*), 656, 660
esp_vfs_fat_rawflash_mount (*C++ function*), 657
esp_vfs_fat_rawflash_unmount (*C++ function*), 657
esp_vfs_fat_register (*C++ function*), 654
esp_vfs_fat_sdmmc_mount (*C++ function*), 655

esp_vfs_fat_sdmmc_unmount (*C++ function*), 656
esp_vfs_fat_spiflash_mount (*C++ function*), 660
esp_vfs_fat_spiflash_unmount (*C++ function*), 661
esp_vfs_fat_unregister_path (*C++ function*), 655
ESP_VFS_FLAG_CONTEXT_PTR (*C macro*), 652
ESP_VFS_FLAG_DEFAULT (*C macro*), 652
esp_vfs_fstat (*C++ function*), 649
esp_vfs_id_t (*C++ type*), 652
esp_vfs_link (*C++ function*), 649
esp_vfs_lseek (*C++ function*), 648
esp_vfs_open (*C++ function*), 649
ESP_VFS_PATH_MAX (*C macro*), 652
esp_vfs_read (*C++ function*), 648
esp_vfs_register (*C++ function*), 649
esp_vfs_register_fd (*C++ function*), 650
esp_vfs_register_fd_range (*C++ function*), 649
esp_vfs_register_with_id (*C++ function*), 649
esp_vfs_rename (*C++ function*), 649
esp_vfs_select (*C++ function*), 650
esp_vfs_select_triggered (*C++ function*), 651
esp_vfs_select_triggered_isr (*C++ function*), 651
esp_vfs_spiffs_conf_t (*C++ class*), 666
esp_vfs_spiffs_conf_t::base_path (*C++ member*), 666
esp_vfs_spiffs_conf_t::format_if_mount_failed (*C++ member*), 666
esp_vfs_spiffs_conf_t::max_files (*C++ member*), 666
esp_vfs_spiffs_conf_t::partition_label (*C++ member*), 666
esp_vfs_spiffs_register (*C++ function*), 664
esp_vfs_spiffs_unregister (*C++ function*), 665
esp_vfs_stat (*C++ function*), 649
esp_vfs_t (*C++ class*), 651
esp_vfs_t::end_select (*C++ member*), 652
esp_vfs_t::flags (*C++ member*), 651
esp_vfs_t::get_socket_select_semaphore (*C++ member*), 652
esp_vfs_t::socket_select (*C++ member*), 651
esp_vfs_t::start_select (*C++ member*), 651
esp_vfs_t::stop_socket_select (*C++ member*), 651
esp_vfs_t::stop_socket_select_isr (*C++ member*), 652
esp_vfs_unlink (*C++ function*), 649
esp_vfs_unregister (*C++ function*), 650
esp_vfs_unregister_fd (*C++ function*), 650
esp_vfs_write (*C++ function*), 648
esp_vhci_host_callback (*C++ class*), 206
esp_vhci_host_callback::notify_host_recv (*C++ member*), 206
esp_vhci_host_callback::notify_host_send_available (*C++ member*), 206
esp_vhci_host_callback_t (*C++ type*), 206
esp_vhci_host_check_send_available (*C++ function*), 203
esp_vhci_host_register_callback (*C++ function*), 204
esp_vhci_host_send_packet (*C++ function*), 203
esp_wifi_80211_tx (*C++ function*), 139
esp_wifi_ap_get_sta_list (*C++ function*), 136
esp_wifi_clear_fast_connect (*C++ function*), 128
esp_wifi_connect (*C++ function*), 127
esp_wifi_deauth_sta (*C++ function*), 128
esp_wifi_deinit (*C++ function*), 126
esp_wifi_disconnect (*C++ function*), 128
esp_wifi_get_ant (*C++ function*), 141
esp_wifi_get_ant_gpio (*C++ function*), 141
esp_wifi_get_auto_connect (*C++ function*), 137
esp_wifi_get_bandwidth (*C++ function*), 131
esp_wifi_get_channel (*C++ function*), 132
esp_wifi_get_config (*C++ function*), 136
esp_wifi_get_country (*C++ function*), 133
esp_wifi_get_event_mask (*C++ function*), 139
esp_wifi_get_mac (*C++ function*), 134
esp_wifi_get_max_tx_power (*C++ function*), 138
esp_wifi_get_mode (*C++ function*), 126
esp_wifi_get_promiscuous (*C++ function*), 134
esp_wifi_get_promiscuous_ctrl_filter (*C++ function*), 135
esp_wifi_get_promiscuous_filter (*C++ function*), 135
esp_wifi_get_protocol (*C++ function*), 131
esp_wifi_get_ps (*C++ function*), 130
esp_wifi_init (*C++ function*), 126
ESP_WIFI_MAX_CONN_NUM (*C macro*), 153
esp_wifi_restore (*C++ function*), 127
esp_wifi_scan_get_ap_num (*C++ function*), 129
esp_wifi_scan_get_ap_records (*C++ function*), 129
esp_wifi_scan_start (*C++ function*), 129
esp_wifi_scan_stop (*C++ function*), 129
esp_wifi_set_ant (*C++ function*), 141
esp_wifi_set_ant_gpio (*C++ function*), 141
esp_wifi_set_auto_connect (*C++ function*), 137
esp_wifi_set_bandwidth (*C++ function*), 131

esp_wifi_set_channel (*C++ function*), 132
esp_wifi_set_config (*C++ function*), 136
esp_wifi_set_country (*C++ function*), 132
esp_wifi_set_csi (*C++ function*), 140
esp_wifi_set_csi_config (*C++ function*), 140
esp_wifi_set_csi_rx_cb (*C++ function*), 140
esp_wifi_set_event_mask (*C++ function*), 139
esp_wifi_set_mac (*C++ function*), 133
esp_wifi_set_max_tx_power (*C++ function*), 138
esp_wifi_set_mode (*C++ function*), 126
esp_wifi_set_promiscuous (*C++ function*), 134
esp_wifi_set_promiscuous_ctrl_filter (*C++ function*), 135
esp_wifi_set_promiscuous_filter (*C++ function*), 135
esp_wifi_set_promiscuous_rx_cb (*C++ function*), 134
esp_wifi_set_protocol (*C++ function*), 130
esp_wifi_set_ps (*C++ function*), 130
esp_wifi_set_storage (*C++ function*), 137
esp_wifi_set_vendor_ie (*C++ function*), 138
esp_wifi_set_vendor_ie_cb (*C++ function*), 138
esp_wifi_sta_get_ap_info (*C++ function*), 130
esp_wifi_start (*C++ function*), 127
esp_wifi_stop (*C++ function*), 127
eStandardSleep (*C++ enumerator*), 692
eSuspended (*C++ enumerator*), 691
eTaskGetState (*C++ function*), 673
eTaskState (*C++ type*), 691
ETH_CLOCK_GPIO0_IN (*C++ enumerator*), 355
ETH_CLOCK_GPIO0_OUT (*C++ enumerator*), 355
ETH_CLOCK_GPIO16_OUT (*C++ enumerator*), 355
ETH_CLOCK_GPIO17_OUT (*C++ enumerator*), 355
eth_clock_mode_t (*C++ type*), 355
eth_config_t (*C++ class*), 354
eth_config_t::clock_mode (*C++ member*), 354
eth_config_t::flow_ctrl_enable (*C++ member*), 354
eth_config_t::gpio_config (*C++ member*), 354
eth_config_t::mac_mode (*C++ member*), 354
eth_config_t::phy_addr (*C++ member*), 354
eth_config_t::phy_check_init (*C++ member*), 354
eth_config_t::phy_check_link (*C++ member*), 354
eth_config_t::phy_get_duplex_mode (*C++ member*), 354
eth_config_t::phy_get_partner_pause_enable (*C++ member*), 354
eth_config_t::phy_get_speed_mode (*C++ member*), 354
eth_config_t::phy_init (*C++ member*), 354
eth_config_t::phy_power_enable (*C++ member*), 354
eth_config_t::tcpip_input (*C++ member*), 354
eth_duplex_mode_t (*C++ type*), 355
eth_gpio_config_func (*C++ type*), 355
ETH_MODE_FULLDUPLEX (*C++ enumerator*), 355
ETH_MODE_HALFDUPLEX (*C++ enumerator*), 355
ETH_MODE_MII (*C++ enumerator*), 355
ETH_MODE_RMII (*C++ enumerator*), 355
eth_mode_t (*C++ type*), 355
eth_phy_base_t (*C++ type*), 355
eth_phy_check_init_func (*C++ type*), 355
eth_phy_check_link_func (*C++ type*), 355
eth_phy_func (*C++ type*), 355
eth_phy_get_duplex_mode_func (*C++ type*), 355
eth_phy_get_partner_pause_enable_func (*C++ type*), 355
eth_phy_get_speed_mode_func (*C++ type*), 355
eth_phy_power_enable_func (*C++ type*), 355
ETH_SPEED_MODE_100M (*C++ enumerator*), 355
ETH_SPEED_MODE_10M (*C++ enumerator*), 355
eth_speed_mode_t (*C++ type*), 355
eth_tcpip_input_func (*C++ type*), 355
ETS_INTERNAL_INTR_SOURCE_OFF (*C macro*), 798
ETS_INTERNAL_PROFILING_INTR_SOURCE (*C macro*), 798
ETS_INTERNAL_SW0_INTR_SOURCE (*C macro*), 797
ETS_INTERNAL_SW1_INTR_SOURCE (*C macro*), 797
ETS_INTERNAL_TIMER0_INTR_SOURCE (*C macro*), 797
ETS_INTERNAL_TIMER1_INTR_SOURCE (*C macro*), 797
ETS_INTERNAL_TIMER2_INTR_SOURCE (*C macro*), 797
EventBits_t (*C++ type*), 756
EventGroupHandle_t (*C++ type*), 755

F

ff_diskio_impl_t (*C++ class*), 658
ff_diskio_impl_t::init (*C++ member*), 658
ff_diskio_impl_t::ioctl (*C++ member*), 658
ff_diskio_impl_t::read (*C++ member*), 658
ff_diskio_impl_t::status (*C++ member*), 658
ff_diskio_impl_t::write (*C++ member*), 658
ff_diskio_register (*C++ function*), 657
ff_diskio_register_sdmmc (*C++ function*), 658
filter_cb_t (*C++ type*), 550

G

gpio_config (*C++ function*), 377
gpio_config_t (*C++ class*), 384

gpio_config_t::intr_type (*C++ member*), 384
gpio_config_t::mode (*C++ member*), 384
gpio_config_t::pin_bit_mask (*C++ member*), 384
gpio_config_t::pull_down_en (*C++ member*), 384
gpio_config_t::pull_up_en (*C++ member*), 384
gpio_deep_sleep_hold_dis (*C++ function*), 384
gpio_deep_sleep_hold_en (*C++ function*), 383
GPIO_DRIVE_CAP_0 (*C++ enumerator*), 386
GPIO_DRIVE_CAP_1 (*C++ enumerator*), 387
GPIO_DRIVE_CAP_2 (*C++ enumerator*), 387
GPIO_DRIVE_CAP_3 (*C++ enumerator*), 387
GPIO_DRIVE_CAP_DEFAULT (*C++ enumerator*), 387
GPIO_DRIVE_CAP_MAX (*C++ enumerator*), 387
gpio_drive_cap_t (*C++ type*), 386
GPIO_FLOATING (*C++ enumerator*), 386
gpio_get_drive_capability (*C++ function*), 383
gpio_get_level (*C++ function*), 379
gpio_hold_dis (*C++ function*), 383
gpio_hold_en (*C++ function*), 383
gpio_install_isr_service (*C++ function*), 381
gpio_int_type_t (*C++ type*), 385
GPIO_INTR_ANYEDGE (*C++ enumerator*), 385
GPIO_INTR_DISABLE (*C++ enumerator*), 385
gpio_intr_disable (*C++ function*), 378
gpio_intr_enable (*C++ function*), 378
GPIO_INTR_HIGH_LEVEL (*C++ enumerator*), 386
GPIO_INTR_LOW_LEVEL (*C++ enumerator*), 385
GPIO_INTR_MAX (*C++ enumerator*), 386
GPIO_INTR_NEGEDGE (*C++ enumerator*), 385
GPIO_INTR_POSEDGE (*C++ enumerator*), 385
gpio_iomux_in (*C++ function*), 384
gpio_iomux_out (*C++ function*), 384
GPIO_IS_VALID_GPIO (*C macro*), 385
GPIO_IS_VALID_OUTPUT_GPIO (*C macro*), 385
gpio_isr_handle_t (*C++ type*), 385
gpio_isr_handler_add (*C++ function*), 382
gpio_isr_handler_remove (*C++ function*), 382
gpio_isr_register (*C++ function*), 380
gpio_isr_t (*C++ type*), 385
GPIO_MODE_DISABLE (*C++ enumerator*), 386
GPIO_MODE_INPUT (*C++ enumerator*), 386
GPIO_MODE_INPUT_OUTPUT (*C++ enumerator*), 386
GPIO_MODE_INPUT_OUTPUT_OD (*C++ enumerator*), 386
GPIO_MODE_OUTPUT (*C++ enumerator*), 386
GPIO_MODE_OUTPUT_OD (*C++ enumerator*), 386
gpio_mode_t (*C++ type*), 386
GPIO_NUM_0 (*C++ enumerator*), 385
GPIO_NUM_1 (*C++ enumerator*), 385
GPIO_NUM_2 (*C++ enumerator*), 385
gpio_num_t (*C++ type*), 385
gpio_pull_mode_t (*C++ type*), 386
gpio_pulldown_dis (*C++ function*), 381
GPIO_PULLDOWN_DISABLE (*C++ enumerator*), 386
gpio_pulldown_en (*C++ function*), 381
GPIO_PULLDOWN_ENABLE (*C++ enumerator*), 386
GPIO_PULLDOWN_ONLY (*C++ enumerator*), 386
gpio_pulldown_t (*C++ type*), 386
gpio_pullup_dis (*C++ function*), 381
GPIO_PULLUP_DISABLE (*C++ enumerator*), 386
gpio_pullup_en (*C++ function*), 380
GPIO_PULLUP_ENABLE (*C++ enumerator*), 386
GPIO_PULLUP_ONLY (*C++ enumerator*), 386
GPIO_PULLUP_PULLDOWN (*C++ enumerator*), 386
gpio_pullup_t (*C++ type*), 386
gpio_reset_pin (*C++ function*), 378
GPIO_SEL_0 (*C macro*), 385
GPIO_SEL_1 (*C macro*), 385
GPIO_SEL_2 (*C macro*), 385
gpio_set_direction (*C++ function*), 379
gpio_set_drive_capability (*C++ function*), 382
gpio_set_intr_type (*C++ function*), 378
gpio_set_level (*C++ function*), 378
gpio_set_pull_mode (*C++ function*), 379
gpio_uninstall_isr_service (*C++ function*), 382
gpio_wakeup_disable (*C++ function*), 380
gpio_wakeup_enable (*C++ function*), 380

H

hall_sensor_read (*C++ function*), 366
heap_caps_add_region (*C++ function*), 778
heap_caps_add_region_with_caps (*C++ function*), 779
heap_caps_calloc (*C++ function*), 774
heap_caps_calloc_prefer (*C++ function*), 776
heap_caps_check_integrity (*C++ function*), 775
heap_caps_check_integrity_addr (*C++ function*), 775
heap_caps_check_integrity_all (*C++ function*), 775
heap_caps_dump (*C++ function*), 777
heap_caps_dump_all (*C++ function*), 777
heap_caps_enable_nonos_stack_heaps (*C++ function*), 778
heap_caps_free (*C++ function*), 773
heap_caps_get_free_size (*C++ function*), 774
heap_caps_get_info (*C++ function*), 774
heap_caps_get_largest_free_block (*C++ function*), 774
heap_caps_get_minimum_free_size (*C++ function*), 774

heap_caps_init (*C++ function*), 778
heap_caps_malloc (*C++ function*), 773
heap_caps_malloc_extmem_enable (*C++ function*), 776
heap_caps_malloc_prefer (*C++ function*), 776
heap_caps_print_heap_info (*C++ function*), 775
heap_caps_realloc (*C++ function*), 773
heap_caps_realloc_prefer (*C++ function*), 776
HEAP_TRACE_ALL (*C++ enumerator*), 791
heap_trace_dump (*C++ function*), 791
heap_trace_get (*C++ function*), 790
heap_trace_get_count (*C++ function*), 790
heap_trace_init_standalone (*C++ function*), 789
HEAP_TRACE_LEAKS (*C++ enumerator*), 791
heap_trace_mode_t (*C++ type*), 791
heap_trace_record_t (*C++ class*), 791
heap_trace_record_t::address (*C++ member*), 791
heap_trace_record_t::alloced_by (*C++ member*), 791
heap_trace_record_t::ccount (*C++ member*), 791
heap_trace_record_t::freed_by (*C++ member*), 791
heap_trace_record_t::size (*C++ member*), 791
heap_trace_resume (*C++ function*), 790
heap_trace_start (*C++ function*), 789
heap_trace_stop (*C++ function*), 790
HSPI_HOST (*C++ enumerator*), 515
HTTP_AUTH_TYPE_BASIC (*C++ enumerator*), 598
HTTP_AUTH_TYPE_DIGEST (*C++ enumerator*), 598
HTTP_AUTH_TYPE_NONE (*C++ enumerator*), 598
HTTP_EVENT_DISCONNECTED (*C++ enumerator*), 598
HTTP_EVENT_ERROR (*C++ enumerator*), 597
http_event_handle_cb (*C++ type*), 597
HTTP_EVENT_HEADER_SENT (*C++ enumerator*), 597
HTTP_EVENT_ON_CONNECTED (*C++ enumerator*), 597
HTTP_EVENT_ON_DATA (*C++ enumerator*), 597
HTTP_EVENT_ON_FINISH (*C++ enumerator*), 597
HTTP_EVENT_ON_HEADER (*C++ enumerator*), 597
HTTP_METHOD_DELETE (*C++ enumerator*), 598
HTTP_METHOD_GET (*C++ enumerator*), 598
HTTP_METHOD_MAX (*C++ enumerator*), 598
HTTP_METHOD_PATCH (*C++ enumerator*), 598
HTTP_METHOD_POST (*C++ enumerator*), 598
HTTP_METHOD_PUT (*C++ enumerator*), 598
HTTP_TRANSPORT_OVER_SSL (*C++ enumerator*), 598
HTTP_TRANSPORT_OVER_TCP (*C++ enumerator*), 598
HTTP_TRANSPORT_UNKNOWN (*C++ enumerator*), 598

I

i2c_ack_type_t (*C++ type*), 407
I2C_ADDR_BIT_10 (*C++ enumerator*), 407
I2C_ADDR_BIT_7 (*C++ enumerator*), 407
I2C_ADDR_BIT_MAX (*C++ enumerator*), 407
i2c_addr_mode_t (*C++ type*), 407
I2C_APB_CLK_FREQ (*C macro*), 406
I2C_CMD_END (*C++ enumerator*), 406
i2c_cmd_handle_t (*C++ type*), 406
i2c_cmd_link_create (*C++ function*), 398
i2c_cmd_link_delete (*C++ function*), 398
I2C_CMD_READ (*C++ enumerator*), 406
I2C_CMD_RESTART (*C++ enumerator*), 406
I2C_CMD_STOP (*C++ enumerator*), 406
I2C_CMD_WRITE (*C++ enumerator*), 406
i2c_config_t (*C++ class*), 405
i2c_config_t::addr_10bit_en (*C++ member*), 405
i2c_config_t::clk_speed (*C++ member*), 405
i2c_config_t::mode (*C++ member*), 405
i2c_config_t::scl_io_num (*C++ member*), 405
i2c_config_t::scl_pullup_en (*C++ member*), 405
i2c_config_t::sda_io_num (*C++ member*), 405
i2c_config_t::sda_pullup_en (*C++ member*), 405
i2c_config_t::slave_addr (*C++ member*), 405
I2C_DATA_MODE_LSB_FIRST (*C++ enumerator*), 406
I2C_DATA_MODE_MAX (*C++ enumerator*), 406
I2C_DATA_MODE_MSB_FIRST (*C++ enumerator*), 406
i2c_driver_delete (*C++ function*), 396
i2c_driver_install (*C++ function*), 396
I2C_FIFO_LEN (*C macro*), 406
i2c_filter_disable (*C++ function*), 402
i2c_filter_enable (*C++ function*), 402
i2c_get_data_mode (*C++ function*), 405
i2c_get_data_timing (*C++ function*), 404
i2c_get_period (*C++ function*), 401
i2c_get_start_timing (*C++ function*), 403
i2c_get_stop_timing (*C++ function*), 403
i2c_get_timeout (*C++ function*), 404
i2c_isr_free (*C++ function*), 397
i2c_isr_register (*C++ function*), 397
I2C_MASTER_ACK (*C++ enumerator*), 407
I2C_MASTER_ACK_MAX (*C++ enumerator*), 407
i2c_master_cmd_begin (*C++ function*), 400
I2C_MASTER_LAST_NACK (*C++ enumerator*), 407
I2C_MASTER_NACK (*C++ enumerator*), 407

I2C_MASTER_READ (*C++ enumerator*), 406
i2c_master_read (*C++ function*), 399
i2c_master_read_byte (*C++ function*), 399
i2c_master_start (*C++ function*), 398
i2c_master_stop (*C++ function*), 400
I2C_MASTER_WRITE (*C++ enumerator*), 406
i2c_master_write (*C++ function*), 399
i2c_master_write_byte (*C++ function*), 398
I2C_MODE_MASTER (*C++ enumerator*), 406
I2C_MODE_MAX (*C++ enumerator*), 406
I2C_MODE_SLAVE (*C++ enumerator*), 406
i2c_mode_t (*C++ type*), 406
I2C_NUM_0 (*C++ enumerator*), 407
I2C_NUM_1 (*C++ enumerator*), 407
I2C_NUM_MAX (*C++ enumerator*), 407
i2c_opmode_t (*C++ type*), 406
i2c_param_config (*C++ function*), 396
i2c_port_t (*C++ type*), 407
i2c_reset_rx_fifo (*C++ function*), 397
i2c_reset_tx_fifo (*C++ function*), 397
i2c_rw_t (*C++ type*), 406
i2c_set_data_mode (*C++ function*), 404
i2c_set_data_timing (*C++ function*), 403
i2c_set_period (*C++ function*), 401
i2c_set_pin (*C++ function*), 397
i2c_set_start_timing (*C++ function*), 402
i2c_set_stop_timing (*C++ function*), 403
i2c_set_timeout (*C++ function*), 404
i2c_slave_read_buffer (*C++ function*), 401
i2c_slave_write_buffer (*C++ function*), 401
i2c_trans_mode_t (*C++ type*), 406
i2s_adc_disable (*C++ function*), 415
i2s_adc_enable (*C++ function*), 414
I2S_BITS_PER_SAMPLE_16BIT (*C++ enumerator*), 416
I2S_BITS_PER_SAMPLE_24BIT (*C++ enumerator*), 416
I2S_BITS_PER_SAMPLE_32BIT (*C++ enumerator*), 416
I2S_BITS_PER_SAMPLE_8BIT (*C++ enumerator*), 416
i2s_bits_per_sample_t (*C++ type*), 416
I2S_CHANNEL_FMT_ALL_LEFT (*C++ enumerator*), 417
I2S_CHANNEL_FMT_ALL_RIGHT (*C++ enumerator*), 417
I2S_CHANNEL_FMT_ONLY_LEFT (*C++ enumerator*), 417
I2S_CHANNEL_FMT_ONLY_RIGHT (*C++ enumerator*), 417
I2S_CHANNEL_FMT_RIGHT_LEFT (*C++ enumerator*), 417
i2s_channel_fmt_t (*C++ type*), 417
I2S_CHANNEL_MONO (*C++ enumerator*), 417
I2S_CHANNEL_STEREO (*C++ enumerator*), 417
i2s_channel_t (*C++ type*), 417
I2S_COMM_FORMAT_I2S (*C++ enumerator*), 417
I2S_COMM_FORMAT_I2S_LSB (*C++ enumerator*), 417
I2S_COMM_FORMAT_I2S_MSB (*C++ enumerator*), 417
I2S_COMM_FORMAT_PCM (*C++ enumerator*), 417
I2S_COMM_FORMAT_PCM_LONG (*C++ enumerator*), 417
I2S_COMM_FORMAT_PCM_SHORT (*C++ enumerator*), 417
i2s_comm_format_t (*C++ type*), 417
i2s_config_t (*C++ class*), 415
i2s_config_t::bits_per_sample (*C++ member*), 415
i2s_config_t::channel_format (*C++ member*), 415
i2s_config_t::communication_format (*C++ member*), 415
i2s_config_t::dma_buf_count (*C++ member*), 415
i2s_config_t::dma_buf_len (*C++ member*), 415
i2s_config_t::fixed_mclk (*C++ member*), 415
i2s_config_t::intr_alloc_flags (*C++ member*), 415
i2s_config_t::mode (*C++ member*), 415
i2s_config_t::sample_rate (*C++ member*), 415
i2s_config_t::use_apll (*C++ member*), 415
I2S_DAC_CHANNEL_BOTH_EN (*C++ enumerator*), 419
I2S_DAC_CHANNEL_DISABLE (*C++ enumerator*), 418
I2S_DAC_CHANNEL_LEFT_EN (*C++ enumerator*), 419
I2S_DAC_CHANNEL_MAX (*C++ enumerator*), 419
I2S_DAC_CHANNEL_RIGHT_EN (*C++ enumerator*), 418
i2s_dac_mode_t (*C++ type*), 418
i2s_driver_install (*C++ function*), 410
i2s_driver_uninstall (*C++ function*), 410
I2S_EVENT_DMA_ERROR (*C++ enumerator*), 418
I2S_EVENT_MAX (*C++ enumerator*), 418
I2S_EVENT_RX_DONE (*C++ enumerator*), 418
i2s_event_t (*C++ class*), 415
i2s_event_t::size (*C++ member*), 416
i2s_event_t::type (*C++ member*), 416
I2S_EVENT_TX_DONE (*C++ enumerator*), 418
i2s_event_type_t (*C++ type*), 418
i2s_isr_handle_t (*C++ type*), 416
I2S_MODE_ADC_BUILT_IN (*C++ enumerator*), 418
I2S_MODE_DAC_BUILT_IN (*C++ enumerator*), 418

I2S_MODE_MASTER (*C++ enumerator*), 418
I2S_MODE_PDM (*C++ enumerator*), 418
I2S_MODE_RX (*C++ enumerator*), 418
I2S_MODE_SLAVE (*C++ enumerator*), 418
i2s_mode_t (*C++ type*), 418
I2S_MODE_TX (*C++ enumerator*), 418
I2S_NUM_0 (*C++ enumerator*), 418
I2S_NUM_1 (*C++ enumerator*), 418
I2S_NUM_MAX (*C++ enumerator*), 418
i2s_pin_config_t (*C++ class*), 416
i2s_pin_config_t::bck_io_num (*C++ member*), 416
i2s_pin_config_t::data_in_num (*C++ member*), 416
i2s_pin_config_t::data_out_num (*C++ member*), 416
i2s_pin_config_t::ws_io_num (*C++ member*), 416
I2S_PIN_NO_CHANGE (*C macro*), 416
i2s_pop_sample (*C++ function*), 412
i2s_port_t (*C++ type*), 417
i2s_push_sample (*C++ function*), 412
i2s_read (*C++ function*), 412
i2s_read_bytes (*C++ function*), 411
i2s_set_adc_mode (*C++ function*), 414
i2s_set_clk (*C++ function*), 414
i2s_set_dac_mode (*C++ function*), 409
i2s_set_pin (*C++ function*), 409
i2s_set_sample_rates (*C++ function*), 413
i2s_start (*C++ function*), 413
i2s_stop (*C++ function*), 413
i2s_write (*C++ function*), 410
i2s_write_bytes (*C++ function*), 410
i2s_write_expand (*C++ function*), 411
i2s_zero_dma_buffer (*C++ function*), 413
I_ADDI (*C macro*), 1115
I_ADDR (*C macro*), 1114
I_ANDI (*C macro*), 1115
I_ANDR (*C macro*), 1114
I_BGE (*C macro*), 1114
I_BL (*C macro*), 1114
I_BXFI (*C macro*), 1114
I_BXFR (*C macro*), 1114
I_BXI (*C macro*), 1114
I_BXR (*C macro*), 1114
I_BXZI (*C macro*), 1114
I_BXZR (*C macro*), 1114
I_DELAY (*C macro*), 1113
I_END (*C macro*), 1113
I_HALT (*C macro*), 1113
I_LD (*C macro*), 1113
I_LSHI (*C macro*), 1115
I_LSHR (*C macro*), 1114
I_MOVI (*C macro*), 1115
I_MOVR (*C macro*), 1114
I_ORI (*C macro*), 1115
I_ORR (*C macro*), 1114
I_RD_REG (*C macro*), 1114
I_RSHI (*C macro*), 1115
I_RSHR (*C macro*), 1115
I_ST (*C macro*), 1113
I_SUBI (*C macro*), 1115
I_SUBR (*C macro*), 1114
I_WR_REG (*C macro*), 1113
intr_handle_data_t (*C++ type*), 798
intr_handle_t (*C++ type*), 798
intr_handler_t (*C++ type*), 798

L

LEDC_APB_CLK (*C++ enumerator*), 432
LEDC_APB_CLK_HZ (*C macro*), 431
ledc_bind_channel_timer (*C++ function*), 427
LEDC_CHANNEL_0 (*C++ enumerator*), 433
LEDC_CHANNEL_1 (*C++ enumerator*), 433
LEDC_CHANNEL_2 (*C++ enumerator*), 433
LEDC_CHANNEL_3 (*C++ enumerator*), 433
LEDC_CHANNEL_4 (*C++ enumerator*), 433
LEDC_CHANNEL_5 (*C++ enumerator*), 433
LEDC_CHANNEL_6 (*C++ enumerator*), 433
LEDC_CHANNEL_7 (*C++ enumerator*), 433
ledc_channel_config (*C++ function*), 423
ledc_channel_config_t (*C++ class*), 431
ledc_channel_config_t::channel (*C++ member*), 431
ledc_channel_config_t::duty (*C++ member*), 431
ledc_channel_config_t::gpio_num (*C++ member*), 431
ledc_channel_config_t::hpoint (*C++ member*), 431
ledc_channel_config_t::intr_type (*C++ member*), 431
ledc_channel_config_t::speed_mode (*C++ member*), 431
ledc_channel_config_t::timer_sel (*C++ member*), 431
LEDC_CHANNEL_MAX (*C++ enumerator*), 433
ledc_channel_t (*C++ type*), 433
ledc_clk_src_t (*C++ type*), 432
LEDC_DUTY_DIR_DECREASE (*C++ enumerator*), 432
LEDC_DUTY_DIR_INCREASE (*C++ enumerator*), 432
LEDC_DUTY_DIR_MAX (*C++ enumerator*), 432
ledc_duty_direction_t (*C++ type*), 432
LEDC_ERR_DUTY (*C macro*), 431
LEDC_ERR_VAL (*C macro*), 431
ledc_fade_func_install (*C++ function*), 429
ledc_fade_func_uninstall (*C++ function*), 429
LEDC_FADE_MAX (*C++ enumerator*), 434

ledc_fade_mode_t (*C++ type*), 434
LEDC_FADE_NO_WAIT (*C++ enumerator*), 434
ledc_fade_start (*C++ function*), 429
LEDC_FADE_WAIT_DONE (*C++ enumerator*), 434
ledc_get_duty (*C++ function*), 425
ledc_get_freq (*C++ function*), 424
ledc_get_hpoint (*C++ function*), 425
LEDC_HIGH_SPEED_MODE (*C++ enumerator*), 432
LEDC_INTR_DISABLE (*C++ enumerator*), 432
LEDC_INTR_FADE_END (*C++ enumerator*), 432
ledc_intr_type_t (*C++ type*), 432
ledc_isr_handle_t (*C++ type*), 432
ledc_isr_register (*C++ function*), 426
LEDC_LOW_SPEED_MODE (*C++ enumerator*), 432
ledc_mode_t (*C++ type*), 432
LEDC_REF_CLK_HZ (*C macro*), 431
LEDC_REF_TICK (*C++ enumerator*), 432
ledc_set_duty (*C++ function*), 425
ledc_set_duty_and_update (*C++ function*), 429
ledc_set_duty_with_hpoint (*C++ function*), 424
ledc_set_fade (*C++ function*), 425
ledc_set_fade_step_and_start (*C++ function*), 430
ledc_set_fade_time_and_start (*C++ function*), 430
ledc_set_fade_with_step (*C++ function*), 428
ledc_set_fade_with_time (*C++ function*), 428
ledc_set_freq (*C++ function*), 424
LEDC_SPEED_MODE_MAX (*C++ enumerator*), 432
ledc_stop (*C++ function*), 423
LEDC_TIMER_0 (*C++ enumerator*), 432
LEDC_TIMER_1 (*C++ enumerator*), 432
LEDC_TIMER_10_BIT (*C++ enumerator*), 433
LEDC_TIMER_11_BIT (*C++ enumerator*), 433
LEDC_TIMER_12_BIT (*C++ enumerator*), 434
LEDC_TIMER_13_BIT (*C++ enumerator*), 434
LEDC_TIMER_14_BIT (*C++ enumerator*), 434
LEDC_TIMER_15_BIT (*C++ enumerator*), 434
LEDC_TIMER_16_BIT (*C++ enumerator*), 434
LEDC_TIMER_17_BIT (*C++ enumerator*), 434
LEDC_TIMER_18_BIT (*C++ enumerator*), 434
LEDC_TIMER_19_BIT (*C++ enumerator*), 434
LEDC_TIMER_1_BIT (*C++ enumerator*), 433
LEDC_TIMER_2 (*C++ enumerator*), 432
LEDC_TIMER_20_BIT (*C++ enumerator*), 434
LEDC_TIMER_2_BIT (*C++ enumerator*), 433
LEDC_TIMER_3 (*C++ enumerator*), 432
LEDC_TIMER_3_BIT (*C++ enumerator*), 433
LEDC_TIMER_4_BIT (*C++ enumerator*), 433
LEDC_TIMER_5_BIT (*C++ enumerator*), 433
LEDC_TIMER_6_BIT (*C++ enumerator*), 433
LEDC_TIMER_7_BIT (*C++ enumerator*), 433
LEDC_TIMER_8_BIT (*C++ enumerator*), 433
LEDC_TIMER_9_BIT (*C++ enumerator*), 433
LEDC_TIMER_BIT_MAX (*C++ enumerator*), 434
ledc_timer_bit_t (*C++ type*), 433
ledc_timer_config (*C++ function*), 423
ledc_timer_config_t (*C++ class*), 431
ledc_timer_config_t::bit_num (*C++ member*), 431
ledc_timer_config_t::duty_resolution (*C++ member*), 431
ledc_timer_config_t::freq_hz (*C++ member*), 431
ledc_timer_config_t::speed_mode (*C++ member*), 431
ledc_timer_config_t::timer_num (*C++ member*), 431
LEDC_TIMER_MAX (*C++ enumerator*), 432
ledc_timer_pause (*C++ function*), 427
ledc_timer_resume (*C++ function*), 427
ledc_timer_rst (*C++ function*), 427
ledc_timer_set (*C++ function*), 426
ledc_timer_t (*C++ type*), 432
ledc_update_duty (*C++ function*), 423

M

M_BGE (*C macro*), 1115
M_BL (*C macro*), 1115
M_BX (*C macro*), 1115
M_BXF (*C macro*), 1115
M_BXZ (*C macro*), 1115
M_LABEL (*C macro*), 1115
MALLOC_CAP_32BIT (*C macro*), 777
MALLOC_CAP_8BIT (*C macro*), 777
MALLOC_CAP_DEFAULT (*C macro*), 778
MALLOC_CAP_DMA (*C macro*), 777
MALLOC_CAP_EXEC (*C macro*), 777
MALLOC_CAP_INTERNAL (*C macro*), 778
MALLOC_CAP_INVALID (*C macro*), 778
MALLOC_CAP_PID2 (*C macro*), 777
MALLOC_CAP_PID3 (*C macro*), 777
MALLOC_CAP_PID4 (*C macro*), 777
MALLOC_CAP_PID5 (*C macro*), 777
MALLOC_CAP_PID6 (*C macro*), 777
MALLOC_CAP_PID7 (*C macro*), 777
MALLOC_CAP_SPIRAM (*C macro*), 778
MAX_FDS (*C macro*), 652
MCPWM0A (*C++ enumerator*), 447
MCPWM0B (*C++ enumerator*), 447
MCPWM1A (*C++ enumerator*), 447
MCPWM1B (*C++ enumerator*), 447
MCPWM2A (*C++ enumerator*), 447
MCPWM2B (*C++ enumerator*), 447
mcpwm_action_on_pwmxa_t (*C++ type*), 450
mcpwm_action_on_pwmxb_t (*C++ type*), 450

MCPWM_ACTIVE_HIGH_COMPLIMENT_MODE (*C++ enumerator*), 451
MCPWM_ACTIVE_HIGH_MODE (*C++ enumerator*), 451
MCPWM_ACTIVE_LOW_COMPLIMENT_MODE (*C++ enumerator*), 451
MCPWM_ACTIVE_LOW_MODE (*C++ enumerator*), 451
MCPWM_ACTIVE_RED_FED_FROM_PWMXA (*C++ enumerator*), 451
MCPWM_ACTIVE_RED_FED_FROM_PWMXB (*C++ enumerator*), 451
MCPWM_BYPASS_FED (*C++ enumerator*), 451
MCPWM_BYPASS_RED (*C++ enumerator*), 451
MCPWM_CAP_0 (*C++ enumerator*), 448
MCPWM_CAP_1 (*C++ enumerator*), 448
MCPWM_CAP_2 (*C++ enumerator*), 448
mcpwm_capture_disable (*C++ function*), 444
mcpwm_capture_enable (*C++ function*), 443
mcpwm_capture_on_edge_t (*C++ type*), 451
mcpwm_capture_signal_get_edge (*C++ function*), 444
mcpwm_capture_signal_get_value (*C++ function*), 444
mcpwm_capture_signal_t (*C++ type*), 450
mcpwm_carrier_config_t (*C++ class*), 446
mcpwm_carrier_config_t::carrier_duty (*C++ member*), 447
mcpwm_carrier_config_t::carrier_ivt_mode (*C++ member*), 447
mcpwm_carrier_config_t::carrier_os_mode (*C++ member*), 447
mcpwm_carrier_config_t::carrier_period (*C++ member*), 447
mcpwm_carrier_config_t::pulse_width_in_os (*C++ member*), 447
mcpwm_carrier_disable (*C++ function*), 440
mcpwm_carrier_enable (*C++ function*), 440
mcpwm_carrier_init (*C++ function*), 439
mcpwm_carrier_oneshot_mode_disable (*C++ function*), 441
mcpwm_carrier_oneshot_mode_enable (*C++ function*), 441
mcpwm_carrier_os_t (*C++ type*), 449
MCPWM_CARRIER_OUT_IVT_DIS (*C++ enumerator*), 449
MCPWM_CARRIER_OUT_IVT_EN (*C++ enumerator*), 449
mcpwm_carrier_out_ivt_t (*C++ type*), 449
mcpwm_carrier_output_invert (*C++ function*), 441
mcpwm_carrier_set_duty_cycle (*C++ function*), 440
mcpwm_carrier_set_period (*C++ function*), 440
mcpwm_config_t (*C++ class*), 446
mcpwm_config_t::cmpr_a (*C++ member*), 446
mcpwm_config_t::cmpr_b (*C++ member*), 446
mcpwm_config_t::counter_mode (*C++ member*), 446
mcpwm_config_t::duty_mode (*C++ member*), 446
mcpwm_config_t::frequency (*C++ member*), 446
MCPWM_COUNTER_MAX (*C++ enumerator*), 449
mcpwm_counter_type_t (*C++ type*), 448
mcpwm_deadtime_disable (*C++ function*), 442
mcpwm_deadtime_enable (*C++ function*), 441
MCPWM_DEADTIME_TYPE_MAX (*C++ enumerator*), 451
mcpwm_deadtime_type_t (*C++ type*), 451
MCPWM_DOWN_COUNTER (*C++ enumerator*), 449
MCPWM_DUTY_MODE_0 (*C++ enumerator*), 449
MCPWM_DUTY_MODE_1 (*C++ enumerator*), 449
MCPWM_DUTY_MODE_MAX (*C++ enumerator*), 449
mcpwm_duty_type_t (*C++ type*), 449
MCPWM_FAULT_0 (*C++ enumerator*), 447
MCPWM_FAULT_1 (*C++ enumerator*), 447
MCPWM_FAULT_2 (*C++ enumerator*), 448
mcpwm_fault_deinit (*C++ function*), 443
mcpwm_fault_init (*C++ function*), 442
mcpwm_fault_input_level_t (*C++ type*), 450
mcpwm_fault_set_cyc_mode (*C++ function*), 443
mcpwm_fault_set_oneshot_mode (*C++ function*), 442
mcpwm_fault_signal_t (*C++ type*), 449
MCPWM_FORCE_MCPWMXA_HIGH (*C++ enumerator*), 450
MCPWM_FORCE_MCPWMXA_LOW (*C++ enumerator*), 450
MCPWM_FORCE_MCPWMXB_HIGH (*C++ enumerator*), 450
MCPWM_FORCE_MCPWMXB_LOW (*C++ enumerator*), 450
mcpwm_get_duty (*C++ function*), 438
mcpwm_get_frequency (*C++ function*), 438
mcpwm_gpio_init (*C++ function*), 436
MCPWM_HIGH_LEVEL_TGR (*C++ enumerator*), 450
mcpwm_init (*C++ function*), 436
mcpwm_io_signals_t (*C++ type*), 447
mcpwm_isr_register (*C++ function*), 445
MCPWM_LOW_LEVEL_TGR (*C++ enumerator*), 450
MCPWM_NEG_EDGE (*C++ enumerator*), 451
MCPWM_NO_CHANGE_IN_MCPWMXA (*C++ enumerator*), 450
MCPWM_NO_CHANGE_IN_MCPWMXB (*C++ enumerator*), 450
MCPWM_ONESHOT_MODE_DIS (*C++ enumerator*), 449
MCPWM_ONESHOT_MODE_EN (*C++ enumerator*), 449
mcpwm_operator_t (*C++ type*), 448
MCPWM_OPR_A (*C++ enumerator*), 448

MCPWM_OPR_B (*C++ enumerator*), 448
MCPWM_OPR_MAX (*C++ enumerator*), 448
mcpwm_pin_config_t (*C++ class*), 445
mcpwm_pin_config_t::mcpwm0a_out_num (*C++ member*), 445
mcpwm_pin_config_t::mcpwm0b_out_num (*C++ member*), 445
mcpwm_pin_config_t::mcpwm1a_out_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm1b_out_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm2a_out_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm2b_out_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_cap0_in_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_cap1_in_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_cap2_in_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_fault0_in_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_fault1_in_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_fault2_in_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_sync0_in_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_sync1_in_num (*C++ member*), 446
mcpwm_pin_config_t::mcpwm_sync2_in_num (*C++ member*), 446
MCPWM_POS_EDGE (*C++ enumerator*), 451
MCPWM_SELECT_CAP0 (*C++ enumerator*), 450
MCPWM_SELECT_CAP1 (*C++ enumerator*), 451
MCPWM_SELECT_CAP2 (*C++ enumerator*), 451
MCPWM_SELECT_F0 (*C++ enumerator*), 450
MCPWM_SELECT_F1 (*C++ enumerator*), 450
MCPWM_SELECT_F2 (*C++ enumerator*), 450
MCPWM_SELECT_SYNC0 (*C++ enumerator*), 449
MCPWM_SELECT_SYNC1 (*C++ enumerator*), 449
MCPWM_SELECT_SYNC2 (*C++ enumerator*), 449
mcpwm_set_duty (*C++ function*), 437
mcpwm_set_duty_in_us (*C++ function*), 437
mcpwm_set_duty_type (*C++ function*), 437
mcpwm_set_frequency (*C++ function*), 437
mcpwm_set_pin (*C++ function*), 436
mcpwm_set_signal_high (*C++ function*), 438
mcpwm_set_signal_low (*C++ function*), 439
mcpwm_start (*C++ function*), 439
mcpwm_stop (*C++ function*), 439
MCPWM_SYNC_0 (*C++ enumerator*), 447
MCPWM_SYNC_1 (*C++ enumerator*), 447
MCPWM_SYNC_2 (*C++ enumerator*), 447
mcpwm_sync_disable (*C++ function*), 445
mcpwm_sync_enable (*C++ function*), 444
mcpwm_sync_signal_t (*C++ type*), 449
MCPWM_TIMER_0 (*C++ enumerator*), 448
MCPWM_TIMER_1 (*C++ enumerator*), 448
MCPWM_TIMER_2 (*C++ enumerator*), 448
MCPWM_TIMER_MAX (*C++ enumerator*), 448
mcpwm_timer_t (*C++ type*), 448
MCPWM_TOG_MCPWMXA (*C++ enumerator*), 450
MCPWM_TOG_MCPWMXB (*C++ enumerator*), 450
MCPWM_UNIT_0 (*C++ enumerator*), 448
MCPWM_UNIT_1 (*C++ enumerator*), 448
MCPWM_UNIT_MAX (*C++ enumerator*), 448
mcpwm_unit_t (*C++ type*), 448
MCPWM_UP_COUNTER (*C++ enumerator*), 448
MCPWM_UP_DOWN_COUNTER (*C++ enumerator*), 449
mdns_free (*C++ function*), 577
mdns_handle_system_event (*C++ function*), 582
mdns_hostname_set (*C++ function*), 577
mdns_init (*C++ function*), 577
mdns_instance_name_set (*C++ function*), 577
mdns_ip_addr_s (*C++ class*), 583
mdns_ip_addr_s::addr (*C++ member*), 583
mdns_ip_addr_s::next (*C++ member*), 583
mdns_ip_addr_t (*C++ type*), 584
MDNS_IP_PROTOCOL_MAX (*C++ enumerator*), 584
mdns_ip_protocol_t (*C++ type*), 584
MDNS_IP_PROTOCOL_V4 (*C++ enumerator*), 584
MDNS_IP_PROTOCOL_V6 (*C++ enumerator*), 584
mdns_query (*C++ function*), 580
mdns_query_a (*C++ function*), 582
mdns_query_aaaa (*C++ function*), 582
mdns_query_ptr (*C++ function*), 581
mdns_query_results_free (*C++ function*), 580
mdns_query_srv (*C++ function*), 581
mdns_query_txt (*C++ function*), 581
mdns_result_s (*C++ class*), 583
mdns_result_s::addr (*C++ member*), 583
mdns_result_s::hostname (*C++ member*), 583
mdns_result_s::instance_name (*C++ member*), 583
mdns_result_s::ip_protocol (*C++ member*), 583
mdns_result_s::next (*C++ member*), 583
mdns_result_s::port (*C++ member*), 583
mdns_result_s::tcpip_if (*C++ member*), 583
mdns_result_s::txt (*C++ member*), 583
mdns_result_s::txt_count (*C++ member*), 583
mdns_result_t (*C++ type*), 584
mdns_service_add (*C++ function*), 577
mdns_service_instance_name_set (*C++ function*), 578
mdns_service_port_set (*C++ function*), 578

mdns_service_remove (*C++ function*), 578
mdns_service_remove_all (*C++ function*), 580
mdns_service_txt_item_remove (*C++ function*), 579
mdns_service_txt_item_set (*C++ function*), 579
mdns_service_txt_set (*C++ function*), 579
mdns_txt_item_t (*C++ class*), 583
mdns_txt_item_t::key (*C++ member*), 583
mdns_txt_item_t::value (*C++ member*), 583
MDNS_TYPE_A (*C macro*), 584
MDNS_TYPE_AAAA (*C macro*), 584
MDNS_TYPE_ANY (*C macro*), 584
MDNS_TYPE_NSEC (*C macro*), 584
MDNS_TYPE_OPT (*C macro*), 584
MDNS_TYPE_PTR (*C macro*), 584
MDNS_TYPE_SRV (*C macro*), 584
MDNS_TYPE_TXT (*C macro*), 584
mesh_addr_t (*C++ type*), 190
mesh_addr_t::addr (*C++ member*), 190
mesh_addr_t::mip (*C++ member*), 190
mesh_ap_cfg_t (*C++ class*), 194
mesh_ap_cfg_t::max_connection (*C++ member*), 195
mesh_ap_cfg_t::password (*C++ member*), 195
MESH_ASSOC_FLAG_NETWORK_FREE (*C macro*), 198
MESH_ASSOC_FLAG_ROOT_FIXED (*C macro*), 198
MESH_ASSOC_FLAG_ROOTS_FOUND (*C macro*), 198
MESH_ASSOC_FLAG_VOTE_IN_PROGRESS (*C macro*), 198
mesh_cfg_t (*C++ class*), 195
mesh_cfg_t::channel (*C++ member*), 195
mesh_cfg_t::crypto_funcs (*C++ member*), 195
mesh_cfg_t::event_cb (*C++ member*), 195
mesh_cfg_t::mesh_ap (*C++ member*), 195
mesh_cfg_t::mesh_id (*C++ member*), 195
mesh_cfg_t::router (*C++ member*), 195
MESH_DATA_DROP (*C macro*), 197
MESH_DATA_ENC (*C macro*), 197
MESH_DATA_FROMDS (*C macro*), 197
MESH_DATA_GROUP (*C macro*), 197
MESH_DATA_NONBLOCK (*C macro*), 197
MESH_DATA_P2P (*C macro*), 197
mesh_data_t (*C++ class*), 194
mesh_data_t::data (*C++ member*), 194
mesh_data_t::proto (*C++ member*), 194
mesh_data_t::size (*C++ member*), 194
mesh_data_t::tos (*C++ member*), 194
MESH_DATA_TODS (*C macro*), 197
mesh_disconnect_reason_t (*C++ type*), 200
mesh_event_cb_t (*C++ type*), 198
MESH_EVENT_CHANNEL_SWITCH (*C++ enumerator*), 198
mesh_event_channel_switch_t (*C++ class*), 191
mesh_event_channel_switch_t::channel (*C++ member*), 192
MESH_EVENT_CHILD_CONNECTED (*C++ enumerator*), 199
mesh_event_child_connected_t (*C++ type*), 198
MESH_EVENT_CHILD_DISCONNECTED (*C++ enumerator*), 199
mesh_event_child_disconnected_t (*C++ type*), 198
mesh_event_connected_t (*C++ class*), 192
mesh_event_connected_t::connected (*C++ member*), 192
mesh_event_connected_t::self_layer (*C++ member*), 192
mesh_event_disconnected_t (*C++ type*), 198
mesh_event_id_t (*C++ type*), 198
mesh_event_info_t (*C++ type*), 190
mesh_event_info_t::channel_switch (*C++ member*), 190
mesh_event_info_t::child_connected (*C++ member*), 190
mesh_event_info_t::child_disconnected (*C++ member*), 190
mesh_event_info_t::connected (*C++ member*), 190
mesh_event_info_t::disconnected (*C++ member*), 190
mesh_event_info_t::got_ip (*C++ member*), 191
mesh_event_info_t::layer_change (*C++ member*), 191
mesh_event_info_t::no_parent (*C++ member*), 191
mesh_event_info_t::root_addr (*C++ member*), 191
mesh_event_info_t::root_conflict (*C++ member*), 191
mesh_event_info_t::root_fixed (*C++ member*), 191
mesh_event_info_t::routing_table (*C++ member*), 190
mesh_event_info_t::scan_done (*C++ member*), 191
mesh_event_info_t::switch_req (*C++ member*), 191
mesh_event_info_t::toDS_state (*C++ member*), 191
mesh_event_info_t::vote_started (*C++ member*), 191
MESH_EVENT_LAYER_CHANGE (*C++ enumerator*), 199

mesh_event_layer_change_t (*C++ class*), 192
mesh_event_layer_change_t::new_layer (*C++ member*), 192
MESH_EVENT_MAX (*C++ enumerator*), 200
MESH_EVENT_NO_PARENT_FOUND (*C++ enumerator*), 199
mesh_event_no_parent_found_t (*C++ class*), 192
mesh_event_no_parent_found_t::scan_times (*C++ member*), 192
MESH_EVENT_PARENT_CONNECTED (*C++ enumerator*), 199
MESH_EVENT_PARENT_DISCONNECTED (*C++ enumerator*), 199
MESH_EVENT_ROOT_ADDRESS (*C++ enumerator*), 199
mesh_event_root_address_t (*C++ type*), 198
MESH_EVENT_ROOT_ASKED_YIELD (*C++ enumerator*), 199
mesh_event_root_conflict_t (*C++ class*), 193
mesh_event_root_conflict_t::addr (*C++ member*), 193
mesh_event_root_conflict_t::capacity (*C++ member*), 193
mesh_event_root_conflict_t::rssi (*C++ member*), 193
MESH_EVENT_ROOT_FIXED (*C++ enumerator*), 199
mesh_event_root_fixed_t (*C++ class*), 193
mesh_event_root_fixed_t::is_fixed (*C++ member*), 193
MESH_EVENT_ROOT_GOT_IP (*C++ enumerator*), 199
mesh_event_root_got_ip_t (*C++ type*), 198
MESH_EVENT_ROOT_LOST_IP (*C++ enumerator*), 199
MESH_EVENT_ROOT_SWITCH_ACK (*C++ enumerator*), 199
MESH_EVENT_ROOT_SWITCH_REQ (*C++ enumerator*), 199
mesh_event_root_switch_req_t (*C++ class*), 192
mesh_event_root_switch_req_t::rc_addr (*C++ member*), 193
mesh_event_root_switch_req_t::reason (*C++ member*), 193
MESH_EVENT_ROUTING_TABLE_ADD (*C++ enumerator*), 199
mesh_event_routing_table_change_t (*C++ class*), 193
mesh_event_routing_table_change_t::rt_size_change (*C++ member*), 193
mesh_event_routing_table_change_t::rt_size_new (*C++ member*), 193
MESH_EVENT_ROUTING_TABLE_REMOVE (*C++ enumerator*), 199
MESH_EVENT_SCAN_DONE (*C++ enumerator*), 199
mesh_event_scan_done_t (*C++ class*), 193
mesh_event_scan_done_t::number (*C++ member*), 193
MESH_EVENT_STARTED (*C++ enumerator*), 198
MESH_EVENT_STOPPED (*C++ enumerator*), 198
mesh_event_t (*C++ class*), 193
mesh_event_t::id (*C++ member*), 194
mesh_event_t::info (*C++ member*), 194
MESH_EVENT_TODS_STATE (*C++ enumerator*), 199
mesh_event_toDS_state_t (*C++ type*), 201
MESH_EVENT_VOTE_STARTED (*C++ enumerator*), 199
mesh_event_vote_started_t (*C++ class*), 192
mesh_event_vote_started_t::attempts (*C++ member*), 192
mesh_event_vote_started_t::rc_addr (*C++ member*), 192
mesh_event_vote_started_t::reason (*C++ member*), 192
MESH_EVENT_VOTE_STOPPED (*C++ enumerator*), 199
MESH_IDLE (*C++ enumerator*), 200
MESH_INIT_CONFIG_DEFAULT (*C macro*), 198
MESH_LEAF (*C++ enumerator*), 200
MESH_MPS (*C macro*), 196
MESH_MTU (*C macro*), 196
MESH_NODE (*C++ enumerator*), 200
MESH_OPT_RECV_DS_ADDR (*C macro*), 198
MESH_OPT_SEND_GROUP (*C macro*), 197
mesh_opt_t (*C++ class*), 194
mesh_opt_t::len (*C++ member*), 194
mesh_opt_t::type (*C++ member*), 194
mesh_opt_t::val (*C++ member*), 194
MESH_PROTO_BIN (*C++ enumerator*), 200
MESH_PROTO_HTTP (*C++ enumerator*), 200
MESH_PROTO_JSON (*C++ enumerator*), 200
MESH_PROTO_MQTT (*C++ enumerator*), 200
mesh_proto_t (*C++ type*), 200
mesh_rc_config_t (*C++ type*), 191
mesh_rc_config_t::attempts (*C++ member*), 191
mesh_rc_config_t::rc_addr (*C++ member*), 191
MESH_REASON_CYCLIC (*C++ enumerator*), 201
MESH_REASON_DIFF_ID (*C++ enumerator*), 201
MESH_REASON_LEAF (*C++ enumerator*), 201
MESH_REASON_PARENT_IDLE (*C++ enumerator*), 201
MESH_REASON_PARENT_STOPPED (*C++ enumerator*), 201
MESH_REASON_ROOTS (*C++ enumerator*), 201
MESH_REASON_SCAN_FAIL (*C++ enumerator*), 201
MESH_ROOT (*C++ enumerator*), 200

MESH_ROOT_LAYER (*C macro*), 196
mesh_router_t (*C++ class*), 194
mesh_router_t::bssid (*C++ member*), 194
mesh_router_t::password (*C++ member*), 194
mesh_router_t::ssid (*C++ member*), 194
mesh_router_t::ssid_len (*C++ member*), 194
mesh_rx_pending_t (*C++ class*), 196
mesh_rx_pending_t::toDS (*C++ member*), 196
mesh_rx_pending_t::toSelf (*C++ member*), 196
MESH_TODS_REACHABLE (*C++ enumerator*), 201
MESH_TODS_UNREACHABLE (*C++ enumerator*), 201
MESH_TOS_DEF (*C++ enumerator*), 200
MESH_TOS_E2E (*C++ enumerator*), 200
MESH_TOS_P2P (*C++ enumerator*), 200
mesh_tos_t (*C++ type*), 200
mesh_tx_pending_t (*C++ class*), 195
mesh_tx_pending_t::broadcast (*C++ member*), 196
mesh_tx_pending_t::mgmt (*C++ member*), 196
mesh_tx_pending_t::to_child (*C++ member*), 195
mesh_tx_pending_t::to_child_p2p (*C++ member*), 196
mesh_tx_pending_t::to_parent (*C++ member*), 195
mesh_tx_pending_t::to_parent_p2p (*C++ member*), 195
mesh_type_t (*C++ type*), 200
MESH_VOTE_REASON_CHILD_INITIATED (*C++ enumerator*), 200
MESH_VOTE_REASON_ROOT_INITIATED (*C++ enumerator*), 200
mesh_vote_reason_t (*C++ type*), 200
mesh_vote_t (*C++ class*), 195
mesh_vote_t::config (*C++ member*), 195
mesh_vote_t::is_rc_specified (*C++ member*), 195
mesh_vote_t::percentage (*C++ member*), 195
mip_t (*C++ class*), 191
mip_t::ip4 (*C++ member*), 191
mip_t::port (*C++ member*), 191
multi_heap_check (*C++ function*), 781
multi_heap_dump (*C++ function*), 781
multi_heap_free (*C++ function*), 780
multi_heap_free_size (*C++ function*), 781
multi_heap_get_allocated_size (*C++ function*), 780
multi_heap_get_info (*C++ function*), 782
multi_heap_handle_t (*C++ type*), 783
multi_heap_info_t (*C++ class*), 782
multi_heap_info_t::allocated_blocks (*C++ member*), 782
multi_heap_info_t::free_blocks (*C++ member*), 783
multi_heap_info_t::largest_free_block (*C++ member*), 782
multi_heap_info_t::minimum_free_bytes (*C++ member*), 782
multi_heap_info_t::total_allocated_bytes (*C++ member*), 782
multi_heap_info_t::total_blocks (*C++ member*), 783
multi_heap_info_t::total_free_bytes (*C++ member*), 782
multi_heap_malloc (*C++ function*), 780
multi_heap_minimum_free_size (*C++ function*), 782
multi_heap_realloc (*C++ function*), 780
multi_heap_register (*C++ function*), 781
multi_heap_set_lock (*C++ function*), 781

N

nvs_close (*C++ function*), 639
nvs_commit (*C++ function*), 638
NVS_DEFAULT_PART_NAME (*C macro*), 642
nvs_erase_all (*C++ function*), 638
nvs_erase_key (*C++ function*), 638
nvs_flash_deinit (*C++ function*), 631
nvs_flash_deinit_partition (*C++ function*), 631
nvs_flash_erase (*C++ function*), 631
nvs_flash_erase_partition (*C++ function*), 632
nvs_flash_generate_keys (*C++ function*), 633
nvs_flash_init (*C++ function*), 631
nvs_flash_init_partition (*C++ function*), 631
nvs_flash_read_security_cfg (*C++ function*), 633
nvs_flash_secure_init (*C++ function*), 632
nvs_flash_secure_init_partition (*C++ function*), 632
nvs_get_blob (*C++ function*), 636
nvs_get_i16 (*C++ function*), 635
nvs_get_i32 (*C++ function*), 635
nvs_get_i64 (*C++ function*), 635
nvs_get_i8 (*C++ function*), 634
nvs_get_stats (*C++ function*), 639
nvs_get_str (*C++ function*), 635
nvs_get_u16 (*C++ function*), 635
nvs_get_u32 (*C++ function*), 635
nvs_get_u64 (*C++ function*), 635
nvs_get_u8 (*C++ function*), 635
nvs_get_used_entry_count (*C++ function*), 639
nvs_handle (*C++ type*), 642
NVS_KEY_SIZE (*C macro*), 633
nvs_open (*C++ function*), 636

nvs_open_from_partition (*C++ function*), 637
nvs_open_mode (*C++ type*), 642
NVS_READONLY (*C++ enumerator*), 642
NVS_READWRITE (*C++ enumerator*), 642
nvs_sec_cfg_t (*C++ class*), 633
nvs_sec_cfg_t::eky (*C++ member*), 633
nvs_sec_cfg_t::tky (*C++ member*), 633
nvs_set_blob (*C++ function*), 637
nvs_set_i16 (*C++ function*), 634
nvs_set_i32 (*C++ function*), 634
nvs_set_i64 (*C++ function*), 634
nvs_set_i8 (*C++ function*), 634
nvs_set_str (*C++ function*), 634
nvs_set_u16 (*C++ function*), 634
nvs_set_u32 (*C++ function*), 634
nvs_set_u64 (*C++ function*), 634
nvs_set_u8 (*C++ function*), 634
nvs_stats_t (*C++ class*), 640
nvs_stats_t::free_entries (*C++ member*), 640
nvs_stats_t::namespace_count (*C++ member*), 640
nvs_stats_t::total_entries (*C++ member*), 640
nvs_stats_t::used_entries (*C++ member*), 640

O

OTA_SIZE_UNKNOWN (*C macro*), 837

P

PCNT_CHANNEL_0 (*C++ enumerator*), 461
PCNT_CHANNEL_1 (*C++ enumerator*), 461
PCNT_CHANNEL_MAX (*C++ enumerator*), 461
pcnt_channel_t (*C++ type*), 461
pcnt_config_t (*C++ class*), 459
pcnt_config_t::channel (*C++ member*), 459
pcnt_config_t::counter_h_lim (*C++ member*), 459
pcnt_config_t::counter_l_lim (*C++ member*), 459
pcnt_config_t::ctrl_gpio_num (*C++ member*), 459
pcnt_config_t::hctrl_mode (*C++ member*), 459
pcnt_config_t::lctrl_mode (*C++ member*), 459
pcnt_config_t::neg_mode (*C++ member*), 459
pcnt_config_t::pos_mode (*C++ member*), 459
pcnt_config_t::pulse_gpio_num (*C++ member*), 459
pcnt_config_t::unit (*C++ member*), 459
PCNT_COUNT_DEC (*C++ enumerator*), 460
PCNT_COUNT_DIS (*C++ enumerator*), 460
PCNT_COUNT_INC (*C++ enumerator*), 460
PCNT_COUNT_MAX (*C++ enumerator*), 460
pcnt_count_mode_t (*C++ type*), 460
pcnt_counter_clear (*C++ function*), 454
pcnt_counter_pause (*C++ function*), 454
pcnt_counter_resume (*C++ function*), 454
pcnt_ctrl_mode_t (*C++ type*), 460
pcnt_event_disable (*C++ function*), 455
pcnt_event_enable (*C++ function*), 455
PCNT_EVT_H_LIM (*C++ enumerator*), 461
PCNT_EVT_L_LIM (*C++ enumerator*), 461
PCNT_EVT_MAX (*C++ enumerator*), 461
PCNT_EVT_THRES_0 (*C++ enumerator*), 461
PCNT_EVT_THRES_1 (*C++ enumerator*), 461
pcnt_evt_type_t (*C++ type*), 461
PCNT_EVT_ZERO (*C++ enumerator*), 461
pcnt_filter_disable (*C++ function*), 457
pcnt_filter_enable (*C++ function*), 457
pcnt_get_counter_value (*C++ function*), 453
pcnt_get_event_value (*C++ function*), 456
pcnt_get_filter_value (*C++ function*), 457
pcnt_intr_disable (*C++ function*), 455
pcnt_intr_enable (*C++ function*), 454
pcnt_isr_handle_t (*C++ type*), 460
pcnt_isr_handler_add (*C++ function*), 458
pcnt_isr_handler_remove (*C++ function*), 459
pcnt_isr_register (*C++ function*), 456
pcnt_isr_service_install (*C++ function*), 458
pcnt_isr_service_uninstall (*C++ function*), 459
PCNT_MODE_DISABLE (*C++ enumerator*), 460
PCNT_MODE_KEEP (*C++ enumerator*), 460
PCNT_MODE_MAX (*C++ enumerator*), 460
PCNT_MODE_REVERSE (*C++ enumerator*), 460
PCNT_PIN_NOT_USED (*C macro*), 460
pcnt_set_event_value (*C++ function*), 455
pcnt_set_filter_value (*C++ function*), 457
pcnt_set_mode (*C++ function*), 457
pcnt_set_pin (*C++ function*), 456
PCNT_UNIT_0 (*C++ enumerator*), 460
PCNT_UNIT_1 (*C++ enumerator*), 460
PCNT_UNIT_2 (*C++ enumerator*), 460
PCNT_UNIT_3 (*C++ enumerator*), 461
PCNT_UNIT_4 (*C++ enumerator*), 461
PCNT_UNIT_5 (*C++ enumerator*), 461
PCNT_UNIT_6 (*C++ enumerator*), 461
PCNT_UNIT_7 (*C++ enumerator*), 461
pcnt_unit_config (*C++ function*), 453
PCNT_UNIT_MAX (*C++ enumerator*), 461
pcnt_unit_t (*C++ type*), 460
pcQueueGetName (*C++ function*), 698
pcTaskGetTaskName (*C++ function*), 678
pcTimerGetTimerName (*C++ function*), 736
PDM_PCM_CONV_DISABLE (*C++ enumerator*), 417

PDM_PCM_CONV_ENABLE (*C++ enumerator*), 417
pdm_pcm_conv_t (*C++ type*), 417
PDM_SAMPLE_RATE_RATIO_128 (*C++ enumerator*), 417
PDM_SAMPLE_RATE_RATIO_64 (*C++ enumerator*), 417
pdm_sample_rate_ratio_t (*C++ type*), 417
PendedFunction_t (*C++ type*), 746
PHY0 (*C++ enumerator*), 355
PHY1 (*C++ enumerator*), 355
PHY10 (*C++ enumerator*), 356
PHY11 (*C++ enumerator*), 356
PHY12 (*C++ enumerator*), 356
PHY13 (*C++ enumerator*), 356
PHY14 (*C++ enumerator*), 356
PHY15 (*C++ enumerator*), 356
PHY16 (*C++ enumerator*), 356
PHY17 (*C++ enumerator*), 356
PHY18 (*C++ enumerator*), 356
PHY19 (*C++ enumerator*), 356
PHY2 (*C++ enumerator*), 355
PHY20 (*C++ enumerator*), 356
PHY21 (*C++ enumerator*), 356
PHY22 (*C++ enumerator*), 356
PHY23 (*C++ enumerator*), 356
PHY24 (*C++ enumerator*), 356
PHY25 (*C++ enumerator*), 356
PHY26 (*C++ enumerator*), 356
PHY27 (*C++ enumerator*), 356
PHY28 (*C++ enumerator*), 356
PHY29 (*C++ enumerator*), 356
PHY3 (*C++ enumerator*), 355
PHY30 (*C++ enumerator*), 356
PHY31 (*C++ enumerator*), 356
PHY4 (*C++ enumerator*), 355
PHY5 (*C++ enumerator*), 355
PHY6 (*C++ enumerator*), 356
PHY7 (*C++ enumerator*), 356
PHY8 (*C++ enumerator*), 356
PHY9 (*C++ enumerator*), 356
phy_lan8720_check_phy_init (*C++ function*), 358
phy_lan8720_default_ethernet_config (*C++ member*), 351
phy_lan8720_dump_registers (*C++ function*), 358
phy_lan8720_get_duplex_mode (*C++ function*), 358
phy_lan8720_get_speed_mode (*C++ function*), 358
phy_lan8720_init (*C++ function*), 358
phy_lan8720_power_enable (*C++ function*), 358
phy_mii_check_link_status (*C++ function*), 357
phy_mii_enable_flow_ctrl (*C++ function*), 357
phy_mii_get_partner_pause_enable (*C++ function*), 357
phy_rmii_configure_data_interface_pins (*C++ function*), 357
phy_rmii_smi_configure_pins (*C++ function*), 357
phy_tlk110_check_phy_init (*C++ function*), 357
phy_tlk110_default_ethernet_config (*C++ member*), 351
phy_tlk110_dump_registers (*C++ function*), 357
phy_tlk110_get_duplex_mode (*C++ function*), 357
phy_tlk110_get_speed_mode (*C++ function*), 357
phy_tlk110_init (*C++ function*), 357
phy_tlk110_power_enable (*C++ function*), 357
pvTaskGetThreadLocalStoragePointer (*C++ function*), 679
pvTimerGetTimerID (*C++ function*), 733
pxTaskGetStackStart (*C++ function*), 678

Q

QueueHandle_t (*C++ type*), 714
QueueSetHandle_t (*C++ type*), 714
QueueSetMemberHandle_t (*C++ type*), 714

R

R0 (*C macro*), 1113
R1 (*C macro*), 1113
R2 (*C macro*), 1113
R3 (*C macro*), 1113
RINGBUF_TYPE_ALLOWSPLIT (*C++ enumerator*), 769
RINGBUF_TYPE_BYTEBUF (*C++ enumerator*), 769
RINGBUF_TYPE_NOSPLIT (*C++ enumerator*), 769
ringbuf_type_t (*C++ type*), 769
RingbufHandle_t (*C++ type*), 769
RMT_BASECLK_APB (*C++ enumerator*), 480
RMT_BASECLK_MAX (*C++ enumerator*), 480
RMT_BASECLK_REF (*C++ enumerator*), 480
RMT_CARRIER_LEVEL_HIGH (*C++ enumerator*), 481
RMT_CARRIER_LEVEL_LOW (*C++ enumerator*), 481
RMT_CARRIER_LEVEL_MAX (*C++ enumerator*), 481
rmt_carrier_level_t (*C++ type*), 481
RMT_CHANNEL_0 (*C++ enumerator*), 480
RMT_CHANNEL_1 (*C++ enumerator*), 480
RMT_CHANNEL_2 (*C++ enumerator*), 480
RMT_CHANNEL_3 (*C++ enumerator*), 480
RMT_CHANNEL_4 (*C++ enumerator*), 480
RMT_CHANNEL_5 (*C++ enumerator*), 480
RMT_CHANNEL_6 (*C++ enumerator*), 480

RMT_CHANNEL_7 (*C++ enumerator*), 480
RMT_CHANNEL_MAX (*C++ enumerator*), 480
rmt_channel_t (*C++ type*), 480
rmt_clr_intr_enable_mask (*C++ function*), 473
rmt_config (*C++ function*), 474
rmt_config_t (*C++ class*), 478
rmt_config_t::channel (*C++ member*), 478
rmt_config_t::clk_div (*C++ member*), 478
rmt_config_t::gpio_num (*C++ member*), 479
rmt_config_t::mem_block_num (*C++ member*), 479
rmt_config_t::rmt_mode (*C++ member*), 478
rmt_config_t::rx_config (*C++ member*), 479
rmt_config_t::tx_config (*C++ member*), 479
RMT_DATA_MODE_FIFO (*C++ enumerator*), 480
RMT_DATA_MODE_MAX (*C++ enumerator*), 480
RMT_DATA_MODE_MEM (*C++ enumerator*), 480
rmt_data_mode_t (*C++ type*), 480
rmt_driver_install (*C++ function*), 475
rmt_driver_uninstall (*C++ function*), 476
rmt_fill_tx_items (*C++ function*), 475
rmt_get_clk_div (*C++ function*), 466
rmt_get_idle_level (*C++ function*), 472
rmt_get_mem_block_num (*C++ function*), 468
rmt_get_mem_pd (*C++ function*), 469
rmt_get_memory_owner (*C++ function*), 470
rmt_get_ringbuf_handle (*C++ function*), 477
rmt_get_rx_idle_thresh (*C++ function*), 467
rmt_get_source_clk (*C++ function*), 472
rmt_get_status (*C++ function*), 472
rmt_get_tx_loop_mode (*C++ function*), 471
RMT_IDLE_LEVEL_HIGH (*C++ enumerator*), 481
RMT_IDLE_LEVEL_LOW (*C++ enumerator*), 481
RMT_IDLE_LEVEL_MAX (*C++ enumerator*), 481
rmt_idle_level_t (*C++ type*), 481
rmt_isr_deregister (*C++ function*), 475
rmt_isr_handle_t (*C++ type*), 479
rmt_isr_register (*C++ function*), 474
RMT_MEM_BLOCK_BYTE_NUM (*C macro*), 479
RMT_MEM_ITEM_NUM (*C macro*), 479
RMT_MEM_OWNER_MAX (*C++ enumerator*), 480
RMT_MEM_OWNER_RX (*C++ enumerator*), 480
rmt_mem_owner_t (*C++ type*), 480
RMT_MEM_OWNER_TX (*C++ enumerator*), 480
rmt_memory_rw_rst (*C++ function*), 470
RMT_MODE_MAX (*C++ enumerator*), 481
RMT_MODE_RX (*C++ enumerator*), 481
rmt_mode_t (*C++ type*), 481
RMT_MODE_TX (*C++ enumerator*), 481
rmt_register_tx_end_callback (*C++ function*), 477
rmt_rx_config_t (*C++ class*), 478
rmt_rx_config_t::filter_en (*C++ member*), 478
rmt_rx_config_t::filter_ticks_thresh (*C++ member*), 478
rmt_rx_config_t::idle_threshold (*C++ member*), 478
rmt_rx_start (*C++ function*), 469
rmt_rx_stop (*C++ function*), 470
rmt_set_clk_div (*C++ function*), 466
rmt_set_err_intr_en (*C++ function*), 473
rmt_set_idle_level (*C++ function*), 472
rmt_set_intr_enable_mask (*C++ function*), 473
rmt_set_mem_block_num (*C++ function*), 467
rmt_set_mem_pd (*C++ function*), 468
rmt_set_memory_owner (*C++ function*), 470
rmt_set_pin (*C++ function*), 474
rmt_set_rx_filter (*C++ function*), 471
rmt_set_rx_idle_thresh (*C++ function*), 467
rmt_set_rx_intr_en (*C++ function*), 473
rmt_set_source_clk (*C++ function*), 471
rmt_set_tx_carrier (*C++ function*), 468
rmt_set_tx_intr_en (*C++ function*), 473
rmt_set_tx_loop_mode (*C++ function*), 470
rmt_set_tx_thr_intr_en (*C++ function*), 473
rmt_source_clk_t (*C++ type*), 480
rmt_translator_init (*C++ function*), 477
rmt_tx_config_t (*C++ class*), 478
rmt_tx_config_t::carrier_duty_percent (*C++ member*), 478
rmt_tx_config_t::carrier_en (*C++ member*), 478
rmt_tx_config_t::carrier_freq_hz (*C++ member*), 478
rmt_tx_config_t::carrier_level (*C++ member*), 478
rmt_tx_config_t::idle_level (*C++ member*), 478
rmt_tx_config_t::idle_output_en (*C++ member*), 478
rmt_tx_config_t::loop_en (*C++ member*), 478
rmt_tx_end_callback_t (*C++ class*), 479
rmt_tx_end_callback_t::arg (*C++ member*), 479
rmt_tx_end_callback_t::function (*C++ member*), 479
rmt_tx_end_fn_t (*C++ type*), 479
rmt_tx_start (*C++ function*), 469
rmt_tx_stop (*C++ function*), 469
rmt_wait_tx_done (*C++ function*), 476
rmt_write_items (*C++ function*), 476
rmt_write_sample (*C++ function*), 477
rtc_gpio_deinit (*C++ function*), 387
rtc_gpio_force_hold_dis_all (*C++ function*), 390
rtc_gpio_get_drive_capability (*C++ function*), 390

rtc_gpio_get_level (*C++ function*), 388
rtc_gpio_hold_dis (*C++ function*), 390
rtc_gpio_hold_en (*C++ function*), 389
rtc_gpio_init (*C++ function*), 387
RTC_GPIO_IS_VALID_GPIO (*C macro*), 391
rtc_gpio_is_valid_gpio (*C++ function*), 387
rtc_gpio_isolate (*C++ function*), 390
RTC_GPIO_MODE_DISABLED (*C++ enumerator*), 391
RTC_GPIO_MODE_INPUT_ONLY (*C++ enumerator*), 391
RTC_GPIO_MODE_INPUT_OUTPUT (*C++ enumerator*), 391
RTC_GPIO_MODE_OUTPUT_ONLY (*C++ enumerator*), 391
rtc_gpio_mode_t (*C++ type*), 391
rtc_gpio_pulldown_dis (*C++ function*), 389
rtc_gpio_pulldown_en (*C++ function*), 388
rtc_gpio_pullup_dis (*C++ function*), 389
rtc_gpio_pullup_en (*C++ function*), 388
rtc_gpio_set_direction (*C++ function*), 388
rtc_gpio_set_drive_capability (*C++ function*), 390
rtc_gpio_set_level (*C++ function*), 388
RTC_SLOW_MEM (*C macro*), 1116

S

sample_to_rmt_t (*C++ type*), 479
sc_callback_t (*C++ type*), 160
SC_STATUS_FIND_CHANNEL (*C++ enumerator*), 161
SC_STATUS_GETTING_SSID_PSWD (*C++ enumerator*), 161
SC_STATUS_LINK (*C++ enumerator*), 161
SC_STATUS_LINK_OVER (*C++ enumerator*), 161
SC_STATUS_WAIT (*C++ enumerator*), 161
SC_TYPE_AIRKISS (*C++ enumerator*), 161
SC_TYPE_ESPTOUCH (*C++ enumerator*), 161
SC_TYPE_ESPTOUCH_AIRKISS (*C++ enumerator*), 161
sdio_event_cb_t (*C++ type*), 498
sdio_slave_buf_handle_t (*C++ type*), 498
sdio_slave_clear_host_int (*C++ function*), 497
sdio_slave_config_t (*C++ class*), 498
sdio_slave_config_t::event_cb (*C++ member*), 498
sdio_slave_config_t::recv_buffer_size (*C++ member*), 498
sdio_slave_config_t::send_queue_size (*C++ member*), 498
sdio_slave_config_t::sending_mode (*C++ member*), 498
sdio_slave_config_t::timing (*C++ member*), 498
sdio_slave_deinit (*C++ function*), 494
sdio_slave_get_host_intena (*C++ function*), 497
SDIO_SLAVE_HOSTINT_BIT0 (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_BIT1 (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_BIT2 (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_BIT3 (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_BIT4 (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_BIT5 (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_BIT6 (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_BIT7 (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_RECV_OVF (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_SEND_NEW_PACKET (*C++ enumerator*), 499
SDIO_SLAVE_HOSTINT_SEND_UDF (*C++ enumerator*), 499
sdio_slave_hostint_t (*C++ type*), 499
sdio_slave_initialize (*C++ function*), 494
sdio_slave_read_reg (*C++ function*), 497
sdio_slave_recv (*C++ function*), 495
sdio_slave_recv_get_buf (*C++ function*), 496
sdio_slave_recv_load_buf (*C++ function*), 495
SDIO_SLAVE_RECV_MAX_BUFFER (*C macro*), 498
sdio_slave_recv_register_buf (*C++ function*), 495
sdio_slave_recv_unregister_buf (*C++ function*), 495
sdio_slave_reset (*C++ function*), 495
sdio_slave_send_get_finished (*C++ function*), 496
sdio_slave_send_host_int (*C++ function*), 497
SDIO_SLAVE_SEND_PACKET (*C++ enumerator*), 499
sdio_slave_send_queue (*C++ function*), 496
SDIO_SLAVE_SEND_STREAM (*C++ enumerator*), 499
sdio_slave_sending_mode_t (*C++ type*), 499
sdio_slave_set_host_intena (*C++ function*), 497
sdio_slave_start (*C++ function*), 494
sdio_slave_stop (*C++ function*), 494
SDIO_SLAVE_TIMING_NSEND_NSAMPLE (*C++ enumerator*), 499
SDIO_SLAVE_TIMING_NSEND_PSAMPLE (*C++ enumerator*), 499
SDIO_SLAVE_TIMING_PSEND_NSAMPLE (*C++ enumerator*), 499
SDIO_SLAVE_TIMING_PSEND_PSAMPLE (*C++

enumerator), 499
sdio_slave_timing_t (*C++ type*), 499
sdio_slave_transmit (*C++ function*), 496
sdio_slave_wait_int (*C++ function*), 498
sdio_slave_write_reg (*C++ function*), 497
sdmmc_card_init (*C++ function*), 616
sdmmc_card_print_info (*C++ function*), 617
sdmmc_card_t (*C++ class*), 623
sdmmc_card_t::cid (*C++ member*), 623
sdmmc_card_t::csd (*C++ member*), 623
sdmmc_card_t::host (*C++ member*), 623
sdmmc_card_t::is_mem (*C++ member*), 623
sdmmc_card_t::is_sdio (*C++ member*), 623
sdmmc_card_t::num_io_functions (*C++ member*), 623
sdmmc_card_t::ocr (*C++ member*), 623
sdmmc_card_t::rca (*C++ member*), 623
sdmmc_card_t::reserved (*C++ member*), 623
sdmmc_card_t::scr (*C++ member*), 623
sdmmc_cid_t (*C++ class*), 621
sdmmc_cid_t::date (*C++ member*), 621
sdmmc_cid_t::mfg_id (*C++ member*), 621
sdmmc_cid_t::name (*C++ member*), 621
sdmmc_cid_t::oem_id (*C++ member*), 621
sdmmc_cid_t::revision (*C++ member*), 621
sdmmc_cid_t::serial (*C++ member*), 621
sdmmc_command_t (*C++ class*), 621
sdmmc_command_t::arg (*C++ member*), 622
sdmmc_command_t::blklen (*C++ member*), 622
sdmmc_command_t::data (*C++ member*), 622
sdmmc_command_t::datalen (*C++ member*), 622
sdmmc_command_t::error (*C++ member*), 622
sdmmc_command_t::flags (*C++ member*), 622
sdmmc_command_t::opcode (*C++ member*), 622
sdmmc_command_t::response (*C++ member*), 622
sdmmc_command_t::timeout_ms (*C++ member*), 622
sdmmc_csd_t (*C++ class*), 620
sdmmc_csd_t::capacity (*C++ member*), 620
sdmmc_csd_t::card_command_class (*C++ member*), 621
sdmmc_csd_t::csd_ver (*C++ member*), 620
sdmmc_csd_t::mmc_ver (*C++ member*), 620
sdmmc_csd_t::read_block_len (*C++ member*), 620
sdmmc_csd_t::sector_size (*C++ member*), 620
sdmmc_csd_t::tr_speed (*C++ member*), 621
SDMMC_FREQ_DEFAULT (*C macro*), 624
SDMMC_FREQ_HIGHSPEED (*C macro*), 624
SDMMC_FREQ_PROBING (*C macro*), 624
SDMMC_HOST_DEFAULT (*C macro*), 485
sdmmc_host_deinit (*C++ function*), 484
sdmmc_host_do_transaction (*C++ function*), 483
SDMMC_HOST_FLAG_1BIT (*C macro*), 623
SDMMC_HOST_FLAG_4BIT (*C macro*), 623
SDMMC_HOST_FLAG_8BIT (*C macro*), 623
SDMMC_HOST_FLAG_SPI (*C macro*), 623
sdmmc_host_get_slot_width (*C++ function*), 483
sdmmc_host_init (*C++ function*), 482
sdmmc_host_init_slot (*C++ function*), 482
sdmmc_host_io_int_enable (*C++ function*), 484
sdmmc_host_io_int_wait (*C++ function*), 484
sdmmc_host_set_bus_width (*C++ function*), 483
sdmmc_host_set_card_clk (*C++ function*), 483
SDMMC_HOST_SLOT_0 (*C macro*), 485
SDMMC_HOST_SLOT_1 (*C macro*), 485
sdmmc_host_t (*C++ class*), 622
sdmmc_host_t::command_timeout_ms (*C++ member*), 623
sdmmc_host_t::deinit (*C++ member*), 623
sdmmc_host_t::do_transaction (*C++ member*), 622
sdmmc_host_t::flags (*C++ member*), 622
sdmmc_host_t::get_bus_width (*C++ member*), 622
sdmmc_host_t::init (*C++ member*), 622
sdmmc_host_t::io_int_enable (*C++ member*), 623
sdmmc_host_t::io_int_wait (*C++ member*), 623
sdmmc_host_t::io_voltage (*C++ member*), 622
sdmmc_host_t::max_freq_khz (*C++ member*), 622
sdmmc_host_t::set_bus_width (*C++ member*), 622
sdmmc_host_t::set_card_clk (*C++ member*), 622
sdmmc_host_t::slot (*C++ member*), 622
sdmmc_io_enable_int (*C++ function*), 620
sdmmc_io_read_blocks (*C++ function*), 619
sdmmc_io_read_byte (*C++ function*), 617
sdmmc_io_read_bytes (*C++ function*), 618
sdmmc_io_wait_int (*C++ function*), 620
sdmmc_io_write_blocks (*C++ function*), 619
sdmmc_io_write_byte (*C++ function*), 618
sdmmc_io_write_bytes (*C++ function*), 618
sdmmc_read_sectors (*C++ function*), 617
sdmmc_response_t (*C++ type*), 624
sdmmc_scr_t (*C++ class*), 621
sdmmc_scr_t::bus_width (*C++ member*), 621
sdmmc_scr_t::sd_spec (*C++ member*), 621
SDMMC_SLOT_CONFIG_DEFAULT (*C macro*), 485
sdmmc_slot_config_t (*C++ class*), 485

sdmmc_slot_config_t::gpio_cd (C++ member), 485
sdmmc_slot_config_t::gpio_wp (C++ member), 485
sdmmc_slot_config_t::width (C++ member), 485
SDMMC_SLOT_NO_CD (C macro), 485
SDMMC_SLOT_NO_WP (C macro), 485
SDMMC_SLOT_WIDTH_DEFAULT (C macro), 485
sdmmc_switch_func_rsp_t (C++ class), 621
sdmmc_switch_func_rsp_t::data (C++ member), 621
sdmmc_write_sectors (C++ function), 617
SDSPI_HOST_DEFAULT (C macro), 488
sdspi_host_deinit (C++ function), 487
sdspi_host_do_transaction (C++ function), 486
sdspi_host_init (C++ function), 486
sdspi_host_init_slot (C++ function), 486
sdspi_host_set_card_clk (C++ function), 487
SDSPI_SLOT_CONFIG_DEFAULT (C macro), 488
sdspi_slot_config_t (C++ class), 487
sdspi_slot_config_t::dma_channel (C++ member), 488
sdspi_slot_config_t::gpio_cd (C++ member), 488
sdspi_slot_config_t::gpio_cs (C++ member), 487
sdspi_slot_config_t::gpio_miso (C++ member), 487
sdspi_slot_config_t::gpio_mosi (C++ member), 487
sdspi_slot_config_t::gpio_sck (C++ member), 487
sdspi_slot_config_t::gpio_wp (C++ member), 488
SDSPI_SLOT_NO_CD (C macro), 488
SDSPI_SLOT_NO_WP (C macro), 488
SemaphoreHandle_t (C++ type), 728
semBINARY_SEMAPHORE_QUEUE_LENGTH (C macro), 714
semGIVE_BLOCK_TIME (C macro), 714
semSEMAPHORE_QUEUE_ITEM_LENGTH (C macro), 714
SIGMADELTA_CHANNEL_0 (C++ enumerator), 502
SIGMADELTA_CHANNEL_1 (C++ enumerator), 502
SIGMADELTA_CHANNEL_2 (C++ enumerator), 502
SIGMADELTA_CHANNEL_3 (C++ enumerator), 502
SIGMADELTA_CHANNEL_4 (C++ enumerator), 502
SIGMADELTA_CHANNEL_5 (C++ enumerator), 502
SIGMADELTA_CHANNEL_6 (C++ enumerator), 502
SIGMADELTA_CHANNEL_7 (C++ enumerator), 502
SIGMADELTA_CHANNEL_MAX (C++ enumerator), 502
sigmadelta_channel_t (C++ type), 502
sigmadelta_config (C++ function), 500
sigmadelta_config_t (C++ class), 501
sigmadelta_config_t::channel (C++ member), 502
sigmadelta_config_t::sigmadelta_duty (C++ member), 502
sigmadelta_config_t::sigmadelta_gpio (C++ member), 502
sigmadelta_config_t::sigmadelta_prescale (C++ member), 502
sigmadelta_set_duty (C++ function), 500
sigmadelta_set_pin (C++ function), 501
sigmadelta_set_prescale (C++ function), 501
slave_transaction_cb_t (C++ type), 527
smartconfig_status_t (C++ type), 161
smartconfig_type_t (C++ type), 161
spi_bus_add_device (C++ function), 516
spi_bus_config_t (C++ class), 514
spi_bus_config_t::flags (C++ member), 514
spi_bus_config_t::intr_flags (C++ member), 514
spi_bus_config_t::max_transfer_sz (C++ member), 514
spi_bus_config_t::miso_io_num (C++ member), 514
spi_bus_config_t::mosi_io_num (C++ member), 514
spi_bus_config_t::quadhd_io_num (C++ member), 514
spi_bus_config_t::quadwp_io_num (C++ member), 514
spi_bus_config_t::sclk_io_num (C++ member), 514
spi_bus_free (C++ function), 516
spi_bus_initialize (C++ function), 516
spi_bus_remove_device (C++ function), 517
spi_cal_clock (C++ function), 518
SPI_DEVICE_3WIRE (C macro), 522
SPI_DEVICE_BIT_LSBFIRST (C macro), 522
SPI_DEVICE_CLK_AS_CS (C macro), 522
spi_device_get_trans_result (C++ function), 517
SPI_DEVICE_HALFDUPLEX (C macro), 522
spi_device_handle_t (C++ type), 522
spi_device_interface_config_t (C++ class), 519
spi_device_interface_config_t::address_bits (C++ member), 519
spi_device_interface_config_t::clock_speed_hz (C++ member), 519
spi_device_interface_config_t::command_bits (C++ member), 519
spi_device_interface_config_t::cs_ena_posttrans (C++ member), 519

spi_device_interface_config_t::cs_ena_pretrans (C++ member), 519
spi_device_interface_config_t::dummy_bits (C++ member), 519
spi_device_interface_config_t::duty_cycle_pos (C++ member), 519
spi_device_interface_config_t::flags (C++ member), 520
spi_device_interface_config_t::input_delay_ns (C++ member), 519
spi_device_interface_config_t::mode (C++ member), 519
spi_device_interface_config_t::post_cb (C++ member), 520
spi_device_interface_config_t::pre_cb (C++ member), 520
spi_device_interface_config_t::queue_size (C++ member), 520
spi_device_interface_config_t::spics_io_num (C++ member), 520
SPI_DEVICE_NO_DUMMY (C macro), 522
SPI_DEVICE_POSITIVE_CS (C macro), 522
spi_device_queue_trans (C++ function), 517
SPI_DEVICE_RXBIT_LSBFIRST (C macro), 522
spi_device_transmit (C++ function), 518
SPI_DEVICE_TXBIT_LSBFIRST (C macro), 521
spi_flash_cache2phys (C++ function), 605
SPI_FLASH_CACHE2PHYS_FAIL (C macro), 607
spi_flash_cache_enabled (C++ function), 606
spi_flash_erase_range (C++ function), 602
spi_flash_erase_sector (C++ function), 602
spi_flash_get_chip_size (C++ function), 602
spi_flash_guard_end_func_t (C++ type), 608
spi_flash_guard_funcs_t (C++ class), 606
spi_flash_guard_funcs_t::end (C++ member), 607
spi_flash_guard_funcs_t::is_safe_write_address (C++ member), 607
spi_flash_guard_funcs_t::op_lock (C++ member), 607
spi_flash_guard_funcs_t::op_unlock (C++ member), 607
spi_flash_guard_funcs_t::start (C++ member), 607
spi_flash_guard_get (C++ function), 606
spi_flash_guard_set (C++ function), 606
spi_flash_guard_start_func_t (C++ type), 608
spi_flash_init (C++ function), 602
spi_flash_is_safe_write_address_t (C++ type), 608
spi_flash_mmap (C++ function), 604
SPI_FLASH_MMAP_DATA (C++ enumerator), 608
spi_flash_mmap_dump (C++ function), 605
spi_flash_mmap_get_free_pages (C++ function), 605
spi_flash_mmap_handle_t (C++ type), 608
SPI_FLASH_MMAP_INST (C++ enumerator), 608
spi_flash_mmap_memory_t (C++ type), 608
spi_flash_mmap_pages (C++ function), 604
SPI_FLASH_MMU_PAGE_SIZE (C macro), 607
spi_flash_munmap (C++ function), 605
spi_flash_op_lock_func_t (C++ type), 608
spi_flash_op_unlock_func_t (C++ type), 608
spi_flash_phys2cache (C++ function), 605
spi_flash_read (C++ function), 603
spi_flash_read_encrypted (C++ function), 603
SPI_FLASH_SEC_SIZE (C macro), 607
spi_flash_write (C++ function), 603
spi_flash_write_encrypted (C++ function), 603
spi_get_freq_limit (C++ function), 519
spi_get_timing (C++ function), 518
SPI_HOST (C++ enumerator), 515
spi_host_device_t (C++ type), 515
SPI_MASTER_FREQ_10M (C macro), 521
SPI_MASTER_FREQ_11M (C macro), 521
SPI_MASTER_FREQ_13M (C macro), 521
SPI_MASTER_FREQ_16M (C macro), 521
SPI_MASTER_FREQ_20M (C macro), 521
SPI_MASTER_FREQ_26M (C macro), 521
SPI_MASTER_FREQ_40M (C macro), 521
SPI_MASTER_FREQ_80M (C macro), 521
SPI_MASTER_FREQ_8M (C macro), 521
SPI_MASTER_FREQ_9M (C macro), 521
SPI_MAX_DMA_LEN (C macro), 514
SPI_SLAVE_BIT_LSBFIRST (C macro), 527
spi_slave_free (C++ function), 525
spi_slave_get_trans_result (C++ function), 525
spi_slave_initialize (C++ function), 524
spi_slave_interface_config_t (C++ class), 526
spi_slave_interface_config_t::flags (C++ member), 526
spi_slave_interface_config_t::mode (C++ member), 527
spi_slave_interface_config_t::post_setup_cb (C++ member), 527
spi_slave_interface_config_t::post_trans_cb (C++ member), 527
spi_slave_interface_config_t::queue_size (C++ member), 526
spi_slave_interface_config_t::spics_io_num (C++ member), 526
spi_slave_queue_trans (C++ function), 525
SPI_SLAVE_RXBIT_LSBFIRST (C macro), 527
spi_slave_transaction_t (C++ class), 527

spi_slave_transaction_t (*C++ type*), 527
spi_slave_transaction_t::length (*C++ member*), 527
spi_slave_transaction_t::rx_buffer (*C++ member*), 527
spi_slave_transaction_t::trans_len (*C++ member*), 527
spi_slave_transaction_t::tx_buffer (*C++ member*), 527
spi_slave_transaction_t::user (*C++ member*), 527
spi_slave_transmit (*C++ function*), 526
SPI_SLAVE_TXBIT_LSBFIRST (*C macro*), 527
SPI_TRANS_MODE_DIO (*C macro*), 522
SPI_TRANS_MODE_DIOQIO_ADDR (*C macro*), 522
SPI_TRANS_MODE_QIO (*C macro*), 522
SPI_TRANS_USE_RXDATA (*C macro*), 522
SPI_TRANS_USE_TXDATA (*C macro*), 522
SPI_TRANS_VARIABLE_ADDR (*C macro*), 522
SPI_TRANS_VARIABLE_CMD (*C macro*), 522
spi_transaction_ext_t (*C++ class*), 521
spi_transaction_ext_t::address_bits (*C++ member*), 521
spi_transaction_ext_t::base (*C++ member*), 521
spi_transaction_ext_t::command_bits (*C++ member*), 521
spi_transaction_t (*C++ class*), 520
spi_transaction_t (*C++ type*), 522
spi_transaction_t::addr (*C++ member*), 520
spi_transaction_t::cmd (*C++ member*), 520
spi_transaction_t::flags (*C++ member*), 520
spi_transaction_t::length (*C++ member*), 520
spi_transaction_t::rx_buffer (*C++ member*), 521
spi_transaction_t::rx_data (*C++ member*), 521
spi_transaction_t::rxlength (*C++ member*), 520
spi_transaction_t::tx_buffer (*C++ member*), 520
spi_transaction_t::tx_data (*C++ member*), 521
spi_transaction_t::user (*C++ member*), 520
spicommon_bus_free_io (*C++ function*), 511
spicommon_bus_free_io_cfg (*C++ function*), 512
spicommon_bus_initialize_io (*C++ function*), 511
SPICOMMON_BUSFLAG_DUAL (*C macro*), 515
SPICOMMON_BUSFLAG_MASTER (*C macro*), 515
SPICOMMON_BUSFLAG_MISO (*C macro*), 515
SPICOMMON_BUSFLAG_MOSI (*C macro*), 515
SPICOMMON_BUSFLAG_NATIVE_PINS (*C macro*), 515
SPICOMMON_BUSFLAG_QUAD (*C macro*), 515
SPICOMMON_BUSFLAG_SCLK (*C macro*), 515
SPICOMMON_BUSFLAG_SLAVE (*C macro*), 514
SPICOMMON_BUSFLAG_WPHD (*C macro*), 515
spicommon_cs_free (*C++ function*), 512
spicommon_cs_free_io (*C++ function*), 512
spicommon_cs_initialize (*C++ function*), 512
spicommon_dma_chan_claim (*C++ function*), 510
spicommon_dma_chan_free (*C++ function*), 510
spicommon_dmaworkaround_idle (*C++ function*), 513
spicommon_dmaworkaround_req_reset (*C++ function*), 513
spicommon_dmaworkaround_reset_in_progress (*C++ function*), 513
spicommon_dmaworkaround_transfer_active (*C++ function*), 514
spicommon_hw_for_host (*C++ function*), 513
spicommon_irqsource_for_host (*C++ function*), 513
spicommon_periph_claim (*C++ function*), 510
spicommon_periph_free (*C++ function*), 510
spicommon_setup_dma_desc_links (*C++ function*), 512

T

taskDISABLE_INTERRUPTS (*C macro*), 690
taskENABLE_INTERRUPTS (*C macro*), 690
taskENTER_CRITICAL (*C macro*), 689
taskENTER_CRITICAL_ISR (*C macro*), 690
taskEXIT_CRITICAL (*C macro*), 690
taskEXIT_CRITICAL_ISR (*C macro*), 690
TaskHandle_t (*C++ type*), 691
TaskHookFunction_t (*C++ type*), 691
taskSCHEDULER_NOT_STARTED (*C macro*), 690
taskSCHEDULER_RUNNING (*C macro*), 690
taskSCHEDULER_SUSPENDED (*C macro*), 690
TaskSnapshot_t (*C++ type*), 691
TaskStatus_t (*C++ type*), 691
taskYIELD (*C macro*), 689
TIMER_0 (*C++ enumerator*), 535
TIMER_1 (*C++ enumerator*), 536
TIMER_ALARM_DIS (*C++ enumerator*), 536
TIMER_ALARM_EN (*C++ enumerator*), 536
TIMER_ALARM_MAX (*C++ enumerator*), 536
timer_alarm_t (*C++ type*), 536
TIMER_AUTORELOAD_DIS (*C++ enumerator*), 536
TIMER_AUTORELOAD_EN (*C++ enumerator*), 536
TIMER_AUTORELOAD_MAX (*C++ enumerator*), 536
timer_autoreload_t (*C++ type*), 536
TIMER_BASE_CLK (*C macro*), 535
timer_config_t (*C++ class*), 535

timer_config_t::alarm_en (*C++ member*), 535
timer_config_t::auto_reload (*C++ member*), 535
timer_config_t::counter_dir (*C++ member*), 535
timer_config_t::counter_en (*C++ member*), 535
timer_config_t::divider (*C++ member*), 535
timer_config_t::intr_type (*C++ member*), 535
timer_count_dir_t (*C++ type*), 536
TIMER_COUNT_DOWN (*C++ enumerator*), 536
TIMER_COUNT_MAX (*C++ enumerator*), 536
TIMER_COUNT_UP (*C++ enumerator*), 536
timer_disable_intr (*C++ function*), 534
timer_enable_intr (*C++ function*), 534
timer_get_alarm_value (*C++ function*), 532
timer_get_config (*C++ function*), 533
timer_get_counter_time_sec (*C++ function*), 530
timer_get_counter_value (*C++ function*), 530
TIMER_GROUP_0 (*C++ enumerator*), 535
TIMER_GROUP_1 (*C++ enumerator*), 535
timer_group_intr_disable (*C++ function*), 534
timer_group_intr_enable (*C++ function*), 534
TIMER_GROUP_MAX (*C++ enumerator*), 535
timer_group_t (*C++ type*), 535
timer_idx_t (*C++ type*), 535
timer_init (*C++ function*), 533
TIMER_INTR_LEVEL (*C++ enumerator*), 536
TIMER_INTR_MAX (*C++ enumerator*), 536
timer_intr_mode_t (*C++ type*), 536
timer_isr_handle_t (*C++ type*), 535
timer_isr_register (*C++ function*), 533
TIMER_MAX (*C++ enumerator*), 536
TIMER_PAUSE (*C++ enumerator*), 536
timer_pause (*C++ function*), 531
timer_set_alarm (*C++ function*), 532
timer_set_alarm_value (*C++ function*), 532
timer_set_auto_reload (*C++ function*), 531
timer_set_counter_mode (*C++ function*), 531
timer_set_counter_value (*C++ function*), 530
timer_set_divider (*C++ function*), 531
TIMER_START (*C++ enumerator*), 536
timer_start (*C++ function*), 530
timer_start_t (*C++ type*), 536
TimerCallbackFunction_t (*C++ type*), 746
TimerHandle_t (*C++ type*), 746
TlsDeleteCallbackFunction_t (*C++ type*), 691
tmrCOMMAND_CHANGE_PERIOD (*C macro*), 736
tmrCOMMAND_CHANGE_PERIOD_FROM_ISR (*C macro*), 736
tmrCOMMAND_DELETE (*C macro*), 736
tmrCOMMAND_EXECUTE_CALLBACK (*C macro*), 736
tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR (*C macro*), 736
tmrCOMMAND_RESET (*C macro*), 736
tmrCOMMAND_RESET_FROM_ISR (*C macro*), 736
tmrCOMMAND_START (*C macro*), 736
tmrCOMMAND_START_DONT_TRACE (*C macro*), 736
tmrCOMMAND_START_FROM_ISR (*C macro*), 736
tmrCOMMAND_STOP (*C macro*), 736
tmrCOMMAND_STOP_FROM_ISR (*C macro*), 736
tmrFIRST_FROM_ISR_COMMAND (*C macro*), 736
touch_cnt_slope_t (*C++ type*), 552
TOUCH_FSM_MODE_DEFAULT (*C macro*), 550
TOUCH_FSM_MODE_MAX (*C++ enumerator*), 553
TOUCH_FSM_MODE_SW (*C++ enumerator*), 553
touch_fsm_mode_t (*C++ type*), 553
TOUCH_FSM_MODE_TIMER (*C++ enumerator*), 553
touch_high_volt_t (*C++ type*), 551
TOUCH_HVOLT_2V4 (*C++ enumerator*), 551
TOUCH_HVOLT_2V5 (*C++ enumerator*), 551
TOUCH_HVOLT_2V6 (*C++ enumerator*), 551
TOUCH_HVOLT_2V7 (*C++ enumerator*), 551
TOUCH_HVOLT_ATTEN_0V (*C++ enumerator*), 552
TOUCH_HVOLT_ATTEN_0V5 (*C++ enumerator*), 552
TOUCH_HVOLT_ATTEN_1V (*C++ enumerator*), 551
TOUCH_HVOLT_ATTEN_1V5 (*C++ enumerator*), 551
TOUCH_HVOLT_ATTEN_KEEP (*C++ enumerator*), 551
TOUCH_HVOLT_ATTEN_MAX (*C++ enumerator*), 552
TOUCH_HVOLT_KEEP (*C++ enumerator*), 551
TOUCH_HVOLT_MAX (*C++ enumerator*), 551
touch_isr_handle_t (*C++ type*), 550
touch_low_volt_t (*C++ type*), 551
TOUCH_LVOLT_0V5 (*C++ enumerator*), 551
TOUCH_LVOLT_0V6 (*C++ enumerator*), 551
TOUCH_LVOLT_0V7 (*C++ enumerator*), 551
TOUCH_LVOLT_0V8 (*C++ enumerator*), 551
TOUCH_LVOLT_KEEP (*C++ enumerator*), 551
TOUCH_LVOLT_MAX (*C++ enumerator*), 551
TOUCH_PAD_BIT_MASK_MAX (*C macro*), 550
touch_pad_clear_group_mask (*C++ function*), 547
touch_pad_clear_status (*C++ function*), 548
touch_pad_config (*C++ function*), 541
touch_pad_deinit (*C++ function*), 540
touch_pad_filter_delete (*C++ function*), 549
touch_pad_filter_start (*C++ function*), 549
touch_pad_filter_stop (*C++ function*), 549
touch_pad_get_cnt_mode (*C++ function*), 544
touch_pad_get_filter_period (*C++ function*), 548
touch_pad_get_fsm_mode (*C++ function*), 545
touch_pad_get_group_mask (*C++ function*), 547
touch_pad_get_meas_time (*C++ function*), 543
touch_pad_get_status (*C++ function*), 548
touch_pad_get_thresh (*C++ function*), 546

touch_pad_get_trigger_mode (*C++ function*), 546
touch_pad_get_trigger_source (*C++ function*), 547
touch_pad_get_voltage (*C++ function*), 544
touch_pad_get_wakeup_status (*C++ function*), 549
TOUCH_PAD_GPIO0_CHANNEL (*C macro*), 553
TOUCH_PAD_GPIO12_CHANNEL (*C macro*), 553
TOUCH_PAD_GPIO13_CHANNEL (*C macro*), 553
TOUCH_PAD_GPIO14_CHANNEL (*C macro*), 553
TOUCH_PAD_GPIO15_CHANNEL (*C macro*), 553
TOUCH_PAD_GPIO27_CHANNEL (*C macro*), 553
TOUCH_PAD_GPIO2_CHANNEL (*C macro*), 553
TOUCH_PAD_GPIO32_CHANNEL (*C macro*), 554
TOUCH_PAD_GPIO33_CHANNEL (*C macro*), 554
TOUCH_PAD_GPIO4_CHANNEL (*C macro*), 553
touch_pad_init (*C++ function*), 540
touch_pad_intr_disable (*C++ function*), 548
touch_pad_intr_enable (*C++ function*), 548
touch_pad_io_init (*C++ function*), 545
touch_pad_isr_deregister (*C++ function*), 543
touch_pad_isr_handler_register (*C++ function*), 542
touch_pad_isr_register (*C++ function*), 543
TOUCH_PAD_MAX (*C++ enumerator*), 551
TOUCH_PAD_MEASURE_CYCLE_DEFAULT (*C macro*), 550
TOUCH_PAD_MEASURE_WAIT_DEFAULT (*C macro*), 550
TOUCH_PAD_NUM0 (*C++ enumerator*), 550
TOUCH_PAD_NUM0_GPIO_NUM (*C macro*), 553
TOUCH_PAD_NUM1 (*C++ enumerator*), 550
TOUCH_PAD_NUM1_GPIO_NUM (*C macro*), 553
TOUCH_PAD_NUM2 (*C++ enumerator*), 550
TOUCH_PAD_NUM2_GPIO_NUM (*C macro*), 553
TOUCH_PAD_NUM3 (*C++ enumerator*), 550
TOUCH_PAD_NUM3_GPIO_NUM (*C macro*), 553
TOUCH_PAD_NUM4 (*C++ enumerator*), 550
TOUCH_PAD_NUM4_GPIO_NUM (*C macro*), 553
TOUCH_PAD_NUM5 (*C++ enumerator*), 550
TOUCH_PAD_NUM5_GPIO_NUM (*C macro*), 553
TOUCH_PAD_NUM6 (*C++ enumerator*), 551
TOUCH_PAD_NUM6_GPIO_NUM (*C macro*), 553
TOUCH_PAD_NUM7 (*C++ enumerator*), 551
TOUCH_PAD_NUM7_GPIO_NUM (*C macro*), 553
TOUCH_PAD_NUM8 (*C++ enumerator*), 551
TOUCH_PAD_NUM8_GPIO_NUM (*C macro*), 554
TOUCH_PAD_NUM9 (*C++ enumerator*), 551
TOUCH_PAD_NUM9_GPIO_NUM (*C macro*), 554
touch_pad_read (*C++ function*), 541
touch_pad_read_filtered (*C++ function*), 541
touch_pad_read_raw_data (*C++ function*), 542
touch_pad_set_cnt_mode (*C++ function*), 544
touch_pad_set_filter_period (*C++ function*), 548
touch_pad_set_filter_read_cb (*C++ function*), 542
touch_pad_set_fsm_mode (*C++ function*), 545
touch_pad_set_group_mask (*C++ function*), 547
touch_pad_set_meas_time (*C++ function*), 543
touch_pad_set_thresh (*C++ function*), 545
touch_pad_set_trigger_mode (*C++ function*), 546
touch_pad_set_trigger_source (*C++ function*), 546
touch_pad_set_voltage (*C++ function*), 544
TOUCH_PAD_SLEEP_CYCLE_DEFAULT (*C macro*), 550
TOUCH_PAD_SLOPE_0 (*C++ enumerator*), 552
TOUCH_PAD_SLOPE_1 (*C++ enumerator*), 552
TOUCH_PAD_SLOPE_2 (*C++ enumerator*), 552
TOUCH_PAD_SLOPE_3 (*C++ enumerator*), 552
TOUCH_PAD_SLOPE_4 (*C++ enumerator*), 552
TOUCH_PAD_SLOPE_5 (*C++ enumerator*), 552
TOUCH_PAD_SLOPE_6 (*C++ enumerator*), 552
TOUCH_PAD_SLOPE_7 (*C++ enumerator*), 552
TOUCH_PAD_SLOPE_MAX (*C++ enumerator*), 552
touch_pad_sw_start (*C++ function*), 545
touch_pad_t (*C++ type*), 550
TOUCH_PAD_TIE_OPT_HIGH (*C++ enumerator*), 553
TOUCH_PAD_TIE_OPT_LOW (*C++ enumerator*), 552
TOUCH_PAD_TIE_OPT_MAX (*C++ enumerator*), 553
touch_tie_opt_t (*C++ type*), 552
TOUCH_TRIGGER_ABOVE (*C++ enumerator*), 552
TOUCH_TRIGGER_BELOW (*C++ enumerator*), 552
TOUCH_TRIGGER_MAX (*C++ enumerator*), 552
TOUCH_TRIGGER_MODE_DEFAULT (*C macro*), 550
touch_trigger_mode_t (*C++ type*), 552
TOUCH_TRIGGER_SOURCE_BOTH (*C++ enumerator*), 552
TOUCH_TRIGGER_SOURCE_DEFAULT (*C macro*), 550
TOUCH_TRIGGER_SOURCE_MAX (*C++ enumerator*), 552
TOUCH_TRIGGER_SOURCE_SET1 (*C++ enumerator*), 552
touch_trigger_src_t (*C++ type*), 552
touch_volt_atten_t (*C++ type*), 551
transaction_cb_t (*C++ type*), 522
tskIDLE_PRIORITY (*C macro*), 689
tskKERNEL_VERSION_BUILD (*C macro*), 689
tskKERNEL_VERSION_MAJOR (*C macro*), 689
tskKERNEL_VERSION_MINOR (*C macro*), 689
tskKERNEL_VERSION_NUMBER (*C macro*), 689
tskNO_AFFINITY (*C macro*), 689

U

UART_BITRATE_MAX (*C macro*), 569

UART_BREAK (*C++ enumerator*), 571
UART_BUFFER_FULL (*C++ enumerator*), 571
uart_clear_intr_status (*C++ function*), 560
uart_config_t (*C++ class*), 568
uart_config_t::baud_rate (*C++ member*), 568
uart_config_t::data_bits (*C++ member*), 568
uart_config_t::flow_ctrl (*C++ member*), 569
uart_config_t::parity (*C++ member*), 569
uart_config_t::rx_flow_ctrl_thresh (*C++ member*), 569
uart_config_t::stop_bits (*C++ member*), 569
uart_config_t::use_ref_tick (*C++ member*), 569
UART_CTS_GPIO19_DIRECT_CHANNEL (*C macro*), 572
UART_CTS_GPIO6_DIRECT_CHANNEL (*C macro*), 573
UART_CTS_GPIO8_DIRECT_CHANNEL (*C macro*), 573
UART_DATA (*C++ enumerator*), 571
UART_DATA_5_BITS (*C++ enumerator*), 570
UART_DATA_6_BITS (*C++ enumerator*), 570
UART_DATA_7_BITS (*C++ enumerator*), 570
UART_DATA_8_BITS (*C++ enumerator*), 570
UART_DATA_BITS_MAX (*C++ enumerator*), 570
UART_DATA_BREAK (*C++ enumerator*), 572
uart_disable_intr_mask (*C++ function*), 560
uart_disable_pattern_det_intr (*C++ function*), 567
uart_disable_rx_intr (*C++ function*), 561
uart_disable_tx_intr (*C++ function*), 561
uart_driver_delete (*C++ function*), 564
uart_driver_install (*C++ function*), 564
uart_enable_intr_mask (*C++ function*), 560
uart_enable_pattern_det_intr (*C++ function*), 567
uart_enable_rx_intr (*C++ function*), 561
uart_enable_tx_intr (*C++ function*), 561
UART_EVENT_MAX (*C++ enumerator*), 572
uart_event_t (*C++ class*), 569
uart_event_t::size (*C++ member*), 569
uart_event_t::type (*C++ member*), 569
uart_event_type_t (*C++ type*), 571
UART_FIFO_LEN (*C macro*), 569
UART_FIFO_OVF (*C++ enumerator*), 571
uart_flush (*C++ function*), 566
uart_flush_input (*C++ function*), 566
UART_FRAME_ERR (*C++ enumerator*), 571
uart_get_baudrate (*C++ function*), 558
uart_get_buffered_data_len (*C++ function*), 566
uart_get_hw_flow_ctrl (*C++ function*), 560
uart_get_parity (*C++ function*), 558
uart_get_stop_bits (*C++ function*), 558
uart_get_word_length (*C++ function*), 557
UART_GPIO10_DIRECT_CHANNEL (*C macro*), 572
UART_GPIO11_DIRECT_CHANNEL (*C macro*), 573
UART_GPIO16_DIRECT_CHANNEL (*C macro*), 573
UART_GPIO17_DIRECT_CHANNEL (*C macro*), 573
UART_GPIO19_DIRECT_CHANNEL (*C macro*), 572
UART_GPIO1_DIRECT_CHANNEL (*C macro*), 572
UART_GPIO22_DIRECT_CHANNEL (*C macro*), 572
UART_GPIO3_DIRECT_CHANNEL (*C macro*), 572
UART_GPIO6_DIRECT_CHANNEL (*C macro*), 573
UART_GPIO7_DIRECT_CHANNEL (*C macro*), 573
UART_GPIO8_DIRECT_CHANNEL (*C macro*), 573
UART_GPIO9_DIRECT_CHANNEL (*C macro*), 572
uart_hw_flowcontrol_t (*C++ type*), 571
UART_HW_FLOWCTRL_CTS (*C++ enumerator*), 571
UART_HW_FLOWCTRL_CTS_RTS (*C++ enumerator*), 571
UART_HW_FLOWCTRL_DISABLE (*C++ enumerator*), 571
UART_HW_FLOWCTRL_MAX (*C++ enumerator*), 571
UART_HW_FLOWCTRL_RTS (*C++ enumerator*), 571
uart_intr_config (*C++ function*), 563
uart_intr_config_t (*C++ class*), 569
uart_intr_config_t::intr_enable_mask (*C++ member*), 569
uart_intr_config_t::rx_timeout_thresh (*C++ member*), 569
uart_intr_config_t::rxfifo_full_thresh (*C++ member*), 569
uart_intr_config_t::txfifo_empty_intr_thresh (*C++ member*), 569
UART_INTR_MASK (*C macro*), 569
UART_INVERSE_CTS (*C macro*), 570
UART_INVERSE_DISABLE (*C macro*), 570
UART_INVERSE_RTS (*C macro*), 570
UART_INVERSE_RXD (*C macro*), 570
UART_INVERSE_TXD (*C macro*), 570
uart_isr_free (*C++ function*), 562
uart_isr_handle_t (*C++ type*), 570
uart_isr_register (*C++ function*), 561
UART_LINE_INV_MASK (*C macro*), 569
UART_NUM_0 (*C++ enumerator*), 571
UART_NUM_0_CTS_DIRECT_GPIO_NUM (*C macro*), 572
UART_NUM_0_RTS_DIRECT_GPIO_NUM (*C macro*), 572
UART_NUM_0_RXD_DIRECT_GPIO_NUM (*C macro*), 572
UART_NUM_0_TXD_DIRECT_GPIO_NUM (*C macro*), 572
UART_NUM_1 (*C++ enumerator*), 571
UART_NUM_1_CTS_DIRECT_GPIO_NUM (*C macro*), 573

UART_NUM_1_RTS_DIRECT_GPIO_NUM (*C macro*), 573
UART_NUM_1_RXD_DIRECT_GPIO_NUM (*C macro*), 573
UART_NUM_1_TXD_DIRECT_GPIO_NUM (*C macro*), 572
UART_NUM_2 (*C++ enumerator*), 571
UART_NUM_2_CTS_DIRECT_GPIO_NUM (*C macro*), 573
UART_NUM_2_RTS_DIRECT_GPIO_NUM (*C macro*), 573
UART_NUM_2_RXD_DIRECT_GPIO_NUM (*C macro*), 573
UART_NUM_2_TXD_DIRECT_GPIO_NUM (*C macro*), 573
UART_NUM_MAX (*C++ enumerator*), 571
uart_param_config (*C++ function*), 563
UART_PARITY_DISABLE (*C++ enumerator*), 571
UART_PARITY_ERR (*C++ enumerator*), 572
UART_PARITY_EVEN (*C++ enumerator*), 571
UART_PARITY_ODD (*C++ enumerator*), 571
uart_parity_t (*C++ type*), 571
UART_PATTERN_DET (*C++ enumerator*), 572
uart_pattern_get_pos (*C++ function*), 568
uart_pattern_pop_pos (*C++ function*), 567
uart_pattern_queue_reset (*C++ function*), 568
UART_PIN_NO_CHANGE (*C macro*), 570
uart_port_t (*C++ type*), 570
uart_read_bytes (*C++ function*), 566
UART_RTS_GPIO11_DIRECT_CHANNEL (*C macro*), 573
UART_RTS_GPIO22_DIRECT_CHANNEL (*C macro*), 572
UART_RTS_GPIO7_DIRECT_CHANNEL (*C macro*), 573
UART_RXD_GPIO16_DIRECT_CHANNEL (*C macro*), 573
UART_RXD_GPIO3_DIRECT_CHANNEL (*C macro*), 572
UART_RXD_GPIO9_DIRECT_CHANNEL (*C macro*), 573
uart_set_baudrate (*C++ function*), 558
uart_set_dtr (*C++ function*), 563
uart_set_hw_flow_ctrl (*C++ function*), 559
uart_set_line_inverse (*C++ function*), 559
uart_set_parity (*C++ function*), 558
uart_set_pin (*C++ function*), 562
uart_set_rts (*C++ function*), 562
uart_set_stop_bits (*C++ function*), 557
uart_set_sw_flow_ctrl (*C++ function*), 559
uart_set_tx_idle_num (*C++ function*), 563
uart_set_word_length (*C++ function*), 557
UART_STOP_BITS_1 (*C++ enumerator*), 570
UART_STOP_BITS_1_5 (*C++ enumerator*), 570
UART_STOP_BITS_2 (*C++ enumerator*), 570
UART_STOP_BITS_MAX (*C++ enumerator*), 570
uart_stop_bits_t (*C++ type*), 570
uart_tx_chars (*C++ function*), 565
UART_TXD_GPIO10_DIRECT_CHANNEL (*C macro*), 573
UART_TXD_GPIO17_DIRECT_CHANNEL (*C macro*), 573
UART_TXD_GPIO1_DIRECT_CHANNEL (*C macro*), 572
uart_wait_tx_done (*C++ function*), 564
uart_word_length_t (*C++ type*), 570
uart_write_bytes (*C++ function*), 565
uart_write_bytes_with_break (*C++ function*), 565
ulp_load_binary (*C++ function*), 1118
ulp_process_macros_and_load (*C++ function*), 1112
ulp_run (*C++ function*), 1112, 1119
ulp_set_wakeup_period (*C++ function*), 1119
ulTaskNotifyTake (*C++ function*), 687
uxQueueMessagesWaiting (*C++ function*), 696
uxQueueMessagesWaitingFromISR (*C++ function*), 693
uxQueueSpacesAvailable (*C++ function*), 696
uxSemaphoreGetCount (*C macro*), 728
uxTaskGetNumberOfTasks (*C++ function*), 678
uxTaskGetStackHighWaterMark (*C++ function*), 678
uxTaskGetSystemState (*C++ function*), 681
uxTaskPriorityGet (*C++ function*), 673
uxTaskPriorityGetFromISR (*C++ function*), 673

V

vendor_ie_data_t (*C++ class*), 149
vendor_ie_data_t::element_id (*C++ member*), 149
vendor_ie_data_t::length (*C++ member*), 149
vendor_ie_data_t::payload (*C++ member*), 149
vendor_ie_data_t::vendor_oui (*C++ member*), 149
vendor_ie_data_t::vendor_oui_type (*C++ member*), 149
vEventGroupDelete (*C++ function*), 753
vprintf_like_t (*C++ type*), 813
vQueueAddToRegistry (*C++ function*), 698
vQueueDelete (*C++ function*), 696
vQueueUnregisterQueue (*C++ function*), 698
vRingbufferDelete (*C++ function*), 767
vRingbufferGetInfo (*C++ function*), 768
vRingbufferReturnItem (*C++ function*), 766
vRingbufferReturnItemFromISR (*C++ function*), 767

vSemaphoreDelete (*C macro*), 728
VSPI_HOST (*C++ enumerator*), 515
vTaskDelay (*C++ function*), 671
vTaskDelayUntil (*C++ function*), 672
vTaskDelete (*C++ function*), 671
vTaskGetRunTimeStats (*C++ function*), 683
vTaskList (*C++ function*), 682
vTaskNotifyGiveFromISR (*C++ function*), 687
vTaskPrioritySet (*C++ function*), 674
vTaskResume (*C++ function*), 675
vTaskSetApplicationTaskTag (*C++ function*), 679
vTaskSetThreadLocalStoragePointer (*C++ function*), 679
vTaskSetThreadLocalStoragePointerAndDelCallback (*C++ function*), 680
vTaskSuspend (*C++ function*), 674
vTaskSuspendAll (*C++ function*), 676
vTimerSetTimerID (*C++ function*), 733

W

wifi_active_scan_time_t (*C++ class*), 146
wifi_active_scan_time_t::max (*C++ member*), 146
wifi_active_scan_time_t::min (*C++ member*), 146
WIFI_ALL_CHANNEL_SCAN (*C++ enumerator*), 157
WIFI_AMPDU_RX_ENABLED (*C macro*), 144
WIFI_AMPDU_TX_ENABLED (*C macro*), 144
WIFI_ANT_ANT0 (*C++ enumerator*), 157
WIFI_ANT_ANT1 (*C++ enumerator*), 157
wifi_ant_config_t (*C++ class*), 152
wifi_ant_config_t::enabled_ant0 (*C++ member*), 152
wifi_ant_config_t::enabled_ant1 (*C++ member*), 152
wifi_ant_config_t::rx_ant_default (*C++ member*), 152
wifi_ant_config_t::rx_ant_mode (*C++ member*), 152
wifi_ant_config_t::tx_ant_mode (*C++ member*), 152
wifi_ant_gpio_config_t (*C++ class*), 152
wifi_ant_gpio_config_t::gpio_cfg (*C++ member*), 152
wifi_ant_gpio_t (*C++ class*), 152
wifi_ant_gpio_t::gpio_num (*C++ member*), 152
wifi_ant_gpio_t::gpio_select (*C++ member*), 152
WIFI_ANT_MAX (*C++ enumerator*), 157
WIFI_ANT_MODE_ANT0 (*C++ enumerator*), 158
WIFI_ANT_MODE_ANT1 (*C++ enumerator*), 158
WIFI_ANT_MODE_AUTO (*C++ enumerator*), 158
WIFI_ANT_MODE_MAX (*C++ enumerator*), 159
wifi_ant_mode_t (*C++ type*), 158
wifi_ant_t (*C++ type*), 157
wifi_ap_config_t (*C++ class*), 147
wifi_ap_config_t::authmode (*C++ member*), 147
wifi_ap_config_t::beacon_interval (*C++ member*), 148
wifi_ap_config_t::channel (*C++ member*), 147
wifi_ap_config_t::max_connection (*C++ member*), 148
wifi_ap_config_t::password (*C++ member*), 147
wifi_ap_config_t::ssid (*C++ member*), 147
wifi_ap_config_t::ssid_hidden (*C++ member*), 148
wifi_ap_config_t::ssid_len (*C++ member*), 147
wifi_ap_record_t (*C++ class*), 146
wifi_ap_record_t::ant (*C++ member*), 147
wifi_ap_record_t::authmode (*C++ member*), 146
wifi_ap_record_t::bssid (*C++ member*), 146
wifi_ap_record_t::country (*C++ member*), 147
wifi_ap_record_t::group_cipher (*C++ member*), 147
wifi_ap_record_t::pairwise_cipher (*C++ member*), 146
wifi_ap_record_t::phy_11b (*C++ member*), 147
wifi_ap_record_t::phy_11g (*C++ member*), 147
wifi_ap_record_t::phy_11n (*C++ member*), 147
wifi_ap_record_t::phy_lr (*C++ member*), 147
wifi_ap_record_t::primary (*C++ member*), 146
wifi_ap_record_t::reserved (*C++ member*), 147
wifi_ap_record_t::rssi (*C++ member*), 146
wifi_ap_record_t::second (*C++ member*), 146
wifi_ap_record_t::ssid (*C++ member*), 146
wifi_ap_record_t::wps (*C++ member*), 147
WIFI_AUTH_MAX (*C++ enumerator*), 155
wifi_auth_mode_t (*C++ type*), 155
WIFI_AUTH_OPEN (*C++ enumerator*), 155
WIFI_AUTH_WEP (*C++ enumerator*), 155
WIFI_AUTH_WPA2_ENTERPRISE (*C++ enumerator*), 155
WIFI_AUTH_WPA2_PSK (*C++ enumerator*), 155
WIFI_AUTH_WPA_PSK (*C++ enumerator*), 155
WIFI_AUTH_WPA_WPA2_PSK (*C++ enumerator*), 155

wifi_bandwidth_t (*C++ type*), 157
WIFI_BW_HT20 (*C++ enumerator*), 157
WIFI_BW_HT40 (*C++ enumerator*), 157
WIFI_CIPHER_TYPE_CCMP (*C++ enumerator*), 157
WIFI_CIPHER_TYPE_NONE (*C++ enumerator*), 156
wifi_cipher_type_t (*C++ type*), 156
WIFI_CIPHER_TYPE_TKIP (*C++ enumerator*), 156
WIFI_CIPHER_TYPE_TKIP_CCMP (*C++ enumerator*), 157
WIFI_CIPHER_TYPE_UNKNOWN (*C++ enumerator*), 157
WIFI_CIPHER_TYPE_WEP104 (*C++ enumerator*), 156
WIFI_CIPHER_TYPE_WEP40 (*C++ enumerator*), 156
wifi_config_t (*C++ type*), 145
wifi_config_t::ap (*C++ member*), 145
wifi_config_t::sta (*C++ member*), 145
WIFI_CONNECT_AP_BY_SECURITY (*C++ enumerator*), 157
WIFI_CONNECT_AP_BY_SIGNAL (*C++ enumerator*), 157
WIFI_COUNTRY_POLICY_AUTO (*C++ enumerator*), 155
WIFI_COUNTRY_POLICY_MANUAL (*C++ enumerator*), 155
wifi_country_policy_t (*C++ type*), 155
wifi_country_t (*C++ class*), 145
wifi_country_t::cc (*C++ member*), 145
wifi_country_t::max_tx_power (*C++ member*), 145
wifi_country_t::nchan (*C++ member*), 145
wifi_country_t::policy (*C++ member*), 145
wifi_country_t::schan (*C++ member*), 145
wifi_csi_cb_t (*C++ type*), 144
wifi_csi_config_t (*C++ class*), 151
wifi_csi_config_t::channel_filter_en (*C++ member*), 151
wifi_csi_config_t::htltf_en (*C++ member*), 151
wifi_csi_config_t::lltf_en (*C++ member*), 151
wifi_csi_config_t::ltf_merge_en (*C++ member*), 151
wifi_csi_config_t::manu_scale (*C++ member*), 151
wifi_csi_config_t::shift (*C++ member*), 151
wifi_csi_config_t::stbc_htltf2_en (*C++ member*), 151
WIFI_CSI_ENABLED (*C macro*), 143
wifi_csi_info_t (*C++ class*), 151
wifi_csi_info_t::buf (*C++ member*), 152
wifi_csi_info_t::first_word_invalid (*C++ member*), 152
wifi_csi_info_t::len (*C++ member*), 152
wifi_csi_info_t::mac (*C++ member*), 152
wifi_csi_info_t::rx_ctrl (*C++ member*), 152
WIFI_DEFAULT_RX_BA_WIN (*C macro*), 144
WIFI_DEFAULT_TX_BA_WIN (*C macro*), 144
WIFI_DYNAMIC_TX_BUFFER_NUM (*C macro*), 143
wifi_err_reason_t (*C++ type*), 155
WIFI_EVENT_MASK_ALL (*C macro*), 154
WIFI_EVENT_MASK_AP_PROBEREQRECVED (*C macro*), 154
WIFI_EVENT_MASK_NONE (*C macro*), 154
WIFI_FAST_SCAN (*C++ enumerator*), 157
wifi_fast_scan_threshold_t (*C++ class*), 147
wifi_fast_scan_threshold_t::authmode (*C++ member*), 147
wifi_fast_scan_threshold_t::rssi (*C++ member*), 147
wifi_ht2040_coex_t (*C++ class*), 153
wifi_ht2040_coex_t::enable (*C++ member*), 153
WIFI_IF_AP (*C macro*), 153
WIFI_IF_STA (*C macro*), 153
WIFI_INIT_CONFIG_DEFAULT (*C macro*), 144
WIFI_INIT_CONFIG_MAGIC (*C macro*), 144
wifi_init_config_t (*C++ class*), 142
wifi_init_config_t::ampdu_rx_enable (*C++ member*), 142
wifi_init_config_t::ampdu_tx_enable (*C++ member*), 142
wifi_init_config_t::beacon_max_len (*C++ member*), 142
wifi_init_config_t::csi_enable (*C++ member*), 142
wifi_init_config_t::dynamic_rx_buf_num (*C++ member*), 142
wifi_init_config_t::dynamic_tx_buf_num (*C++ member*), 142
wifi_init_config_t::event_handler (*C++ member*), 142
wifi_init_config_t::magic (*C++ member*), 142
wifi_init_config_t::mgmt_sbuf_num (*C++ member*), 142
wifi_init_config_t::nano_enable (*C++ member*), 142
wifi_init_config_t::nvs_enable (*C++ member*), 142
wifi_init_config_t::osi_funcs (*C++ member*), 142
wifi_init_config_t::rx_ba_win (*C++ member*), 142
wifi_init_config_t::static_rx_buf_num (*C++ member*), 142
wifi_init_config_t::static_tx_buf_num (*C++ member*), 142

wifi_init_config_t::tx_ba_win (*C++ member*), 142
wifi_init_config_t::tx_buf_type (*C++ member*), 142
wifi_init_config_t::wifi_task_core_id (*C++ member*), 142
wifi_init_config_t::wpa_crypto_funcs (*C++ member*), 142
wifi_interface_t (*C++ type*), 154
wifi_ioctl_cmd_t (*C++ type*), 159
wifi_ioctl_config_t (*C++ class*), 153
wifi_ioctl_config_t::data (*C++ member*), 153
wifi_ioctl_config_t::ht2040_coex (*C++ member*), 153
WIFI_IOCTL_GET_STA_HT2040_COEX (*C++ enumerator*), 159
WIFI_IOCTL_MAX (*C++ enumerator*), 159
WIFI_IOCTL_SET_STA_HT2040_COEX (*C++ enumerator*), 159
WIFI_MGMT_SBUF_NUM (*C macro*), 144
WIFI_MODE_AP (*C++ enumerator*), 154
WIFI_MODE_APSTA (*C++ enumerator*), 155
WIFI_MODE_MAX (*C++ enumerator*), 155
WIFI_MODE_NULL (*C++ enumerator*), 154
WIFI_MODE_STA (*C++ enumerator*), 154
wifi_mode_t (*C++ type*), 154
WIFI_NANO_FORMAT_ENABLED (*C macro*), 144
WIFI_NVS_ENABLED (*C macro*), 144
WIFI_PKT_CTRL (*C++ enumerator*), 158
WIFI_PKT_DATA (*C++ enumerator*), 158
WIFI_PKT_MGMT (*C++ enumerator*), 158
WIFI_PKT_MISC (*C++ enumerator*), 158
wifi_pkt_rx_ctrl_t (*C++ class*), 149
wifi_pkt_rx_ctrl_t::__pad0__ (*C++ member*), 149
wifi_pkt_rx_ctrl_t::__pad1__ (*C++ member*), 150
wifi_pkt_rx_ctrl_t::__pad2__ (*C++ member*), 150
wifi_pkt_rx_ctrl_t::__pad3__ (*C++ member*), 150
wifi_pkt_rx_ctrl_t::__pad4__ (*C++ member*), 150
wifi_pkt_rx_ctrl_t::__pad5__ (*C++ member*), 150
wifi_pkt_rx_ctrl_t::__pad6__ (*C++ member*), 150
wifi_pkt_rx_ctrl_t::__pad7__ (*C++ member*), 151
wifi_pkt_rx_ctrl_t::aggregation (*C++ member*), 150
wifi_pkt_rx_ctrl_t::ampdu_cnt (*C++ member*), 150
wifi_pkt_rx_ctrl_t::ant (*C++ member*), 150
wifi_pkt_rx_ctrl_t::channel (*C++ member*), 150
wifi_pkt_rx_ctrl_t::cwb (*C++ member*), 150
wifi_pkt_rx_ctrl_t::fec_coding (*C++ member*), 150
wifi_pkt_rx_ctrl_t::mcs (*C++ member*), 150
wifi_pkt_rx_ctrl_t::noise_floor (*C++ member*), 150
wifi_pkt_rx_ctrl_t::not_sounding (*C++ member*), 150
wifi_pkt_rx_ctrl_t::rate (*C++ member*), 149
wifi_pkt_rx_ctrl_t::rssi (*C++ member*), 149
wifi_pkt_rx_ctrl_t::rx_state (*C++ member*), 151
wifi_pkt_rx_ctrl_t::secondary_channel (*C++ member*), 150
wifi_pkt_rx_ctrl_t::sgi (*C++ member*), 150
wifi_pkt_rx_ctrl_t::sig_len (*C++ member*), 150
wifi_pkt_rx_ctrl_t::sig_mode (*C++ member*), 149
wifi_pkt_rx_ctrl_t::smoothing (*C++ member*), 150
wifi_pkt_rx_ctrl_t::stbc (*C++ member*), 150
wifi_pkt_rx_ctrl_t::timestamp (*C++ member*), 150
WIFI_PROMIS_CTRL_FILTER_MASK_ACK (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_ALL (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_BA (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_BAR (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_CFEND (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_CFENDACK (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_CTS (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_PSPOLL (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_RTS (*C macro*), 154
WIFI_PROMIS_CTRL_FILTER_MASK_WRAPPER (*C macro*), 154
WIFI_PROMIS_FILTER_MASK_ALL (*C macro*), 153
WIFI_PROMIS_FILTER_MASK_CTRL (*C macro*), 153
WIFI_PROMIS_FILTER_MASK_DATA (*C macro*), 153
WIFI_PROMIS_FILTER_MASK_DATA_AMPDU (*C macro*), 153
WIFI_PROMIS_FILTER_MASK_DATA_MPDU (*C macro*), 153

WIFI_PROMIS_FILTER_MASK_MGMT (*C macro*), 153
WIFI_PROMIS_FILTER_MASK_MISC (*C macro*), 153
wifi_promiscuous_cb_t (*C++ type*), 144
wifi_promiscuous_filter_t (*C++ class*), 151
wifi_promiscuous_filter_t::filter_mask (*C++ member*), 151
wifi_promiscuous_pkt_t (*C++ class*), 151
wifi_promiscuous_pkt_t::payload (*C++ member*), 151
wifi_promiscuous_pkt_t::rx_ctrl (*C++ member*), 151
wifi_promiscuous_pkt_type_t (*C++ type*), 158
WIFI_PROTOCOL_11B (*C macro*), 153
WIFI_PROTOCOL_11G (*C macro*), 153
WIFI_PROTOCOL_11N (*C macro*), 153
WIFI_PROTOCOL_LR (*C macro*), 153
WIFI_PS_MAX_MODEM (*C++ enumerator*), 157
WIFI_PS_MIN_MODEM (*C++ enumerator*), 157
WIFI_PS_MODEM (*C macro*), 153
WIFI_PS_NONE (*C++ enumerator*), 157
wifi_ps_type_t (*C++ type*), 157
WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT (*C++ enumerator*), 156
WIFI_REASON_802_1X_AUTH_FAILED (*C++ enumerator*), 156
WIFI_REASON_AKMP_INVALID (*C++ enumerator*), 156
WIFI_REASON_ASSOC_EXPIRE (*C++ enumerator*), 155
WIFI_REASON_ASSOC_FAIL (*C++ enumerator*), 156
WIFI_REASON_ASSOC_LEAVE (*C++ enumerator*), 155
WIFI_REASON_ASSOC_NOT_AUTHED (*C++ enumerator*), 155
WIFI_REASON_ASSOC_TOOMANY (*C++ enumerator*), 155
WIFI_REASON_AUTH_EXPIRE (*C++ enumerator*), 155
WIFI_REASON_AUTH_FAIL (*C++ enumerator*), 156
WIFI_REASON_AUTH_LEAVE (*C++ enumerator*), 155
WIFI_REASON_BEACON_TIMEOUT (*C++ enumerator*), 156
WIFI_REASON_CIPHER_SUITE_REJECTED (*C++ enumerator*), 156
WIFI_REASON_CONNECTION_FAIL (*C++ enumerator*), 156
WIFI_REASON_DISASSOC_PWRCAP_BAD (*C++ enumerator*), 155
WIFI_REASON_DISASSOC_SUPCHAN_BAD (*C++ enumerator*), 155
WIFI_REASON_GROUP_CIPHER_INVALID (*C++ enumerator*), 156
WIFI_REASON_GROUP_KEY_UPDATE_TIMEOUT (*C++ enumerator*), 156
WIFI_REASON_HANDSHAKE_TIMEOUT (*C++ enumerator*), 156
WIFI_REASON_IE_IN_4WAY_DIFFERS (*C++ enumerator*), 156
WIFI_REASON_IE_INVALID (*C++ enumerator*), 155
WIFI_REASON_INVALID_RSN_IE_CAP (*C++ enumerator*), 156
WIFI_REASON_MIC_FAILURE (*C++ enumerator*), 155
WIFI_REASON_NO_AP_FOUND (*C++ enumerator*), 156
WIFI_REASON_NOT_ASSOCED (*C++ enumerator*), 155
WIFI_REASON_NOT_AUTHED (*C++ enumerator*), 155
WIFI_REASON_PAIRWISE_CIPHER_INVALID (*C++ enumerator*), 156
WIFI_REASON_UNSPECIFIED (*C++ enumerator*), 155
WIFI_REASON_UNSUPP_RSN_IE_VERSION (*C++ enumerator*), 156
wifi_scan_config_t (*C++ class*), 146
wifi_scan_config_t::bssid (*C++ member*), 146
wifi_scan_config_t::channel (*C++ member*), 146
wifi_scan_config_t::scan_time (*C++ member*), 146
wifi_scan_config_t::scan_type (*C++ member*), 146
wifi_scan_config_t::show_hidden (*C++ member*), 146
wifi_scan_config_t::ssid (*C++ member*), 146
wifi_scan_method_t (*C++ type*), 157
wifi_scan_threshold_t (*C++ type*), 154
wifi_scan_time_t (*C++ type*), 145
wifi_scan_time_t::active (*C++ member*), 145
wifi_scan_time_t::passive (*C++ member*), 145
WIFI_SCAN_TYPE_ACTIVE (*C++ enumerator*), 156
WIFI_SCAN_TYPE_PASSIVE (*C++ enumerator*), 156
wifi_scan_type_t (*C++ type*), 156
WIFI_SECOND_CHAN_ABOVE (*C++ enumerator*), 156
WIFI_SECOND_CHAN_BELOW (*C++ enumerator*), 156
WIFI_SECOND_CHAN_NONE (*C++ enumerator*), 156
wifi_second_chan_t (*C++ type*), 156
WIFI_SOFTAP_BEACON_MAX_LEN (*C macro*), 144
wifi_sort_method_t (*C++ type*), 157
wifi_sta_config_t (*C++ class*), 148
wifi_sta_config_t::bssid (*C++ member*), 148
wifi_sta_config_t::bssid_set (*C++ member*), 148
wifi_sta_config_t::channel (*C++ member*), 148
wifi_sta_config_t::listen_interval (*C++

member), 148
wifi_sta_config_t::password (C++ member), 148
wifi_sta_config_t::scan_method (C++ member), 148
wifi_sta_config_t::sort_method (C++ member), 148
wifi_sta_config_t::ssid (C++ member), 148
wifi_sta_config_t::threshold (C++ member), 148
wifi_sta_info_t (C++ class), 148
wifi_sta_info_t::mac (C++ member), 148
wifi_sta_info_t::phy_11b (C++ member), 148
wifi_sta_info_t::phy_11g (C++ member), 148
wifi_sta_info_t::phy_11n (C++ member), 149
wifi_sta_info_t::phy_lr (C++ member), 149
wifi_sta_info_t::reserved (C++ member), 149
wifi_sta_info_t::rssi (C++ member), 148
wifi_sta_list_t (C++ class), 149
wifi_sta_list_t::num (C++ member), 149
wifi_sta_list_t::sta (C++ member), 149
WIFI_STATIC_TX_BUFFER_NUM (C macro), 143
WIFI_STORAGE_FLASH (C++ enumerator), 158
WIFI_STORAGE_RAM (C++ enumerator), 158
wifi_storage_t (C++ type), 157
WIFI_TASK_CORE_ID (C macro), 144
WIFI_VENDOR_IE_ELEMENT_ID (C macro), 153
wifi_vendor_ie_id_t (C++ type), 158
wifi_vendor_ie_type_t (C++ type), 158
WIFI_VND_IE_ID_0 (C++ enumerator), 158
WIFI_VND_IE_ID_1 (C++ enumerator), 158
WIFI_VND_IE_TYPE_ASSOC_REQ (C++ enumerator), 158
WIFI_VND_IE_TYPE_ASSOC_RESP (C++ enumerator), 158
WIFI_VND_IE_TYPE_BEACON (C++ enumerator), 158
WIFI_VND_IE_TYPE_PROBE_REQ (C++ enumerator), 158
WIFI_VND_IE_TYPE_PROBE_RESP (C++ enumerator), 158
wl_erase_range (C++ function), 662
wl_handle_t (C++ type), 663
WL_INVALID_HANDLE (C macro), 663
wl_mount (C++ function), 661
wl_read (C++ function), 662
wl_sector_size (C++ function), 663
wl_size (C++ function), 663
wl_unmount (C++ function), 661
wl_write (C++ function), 662

X

xEventGroupClearBits (C++ function), 749
xEventGroupClearBitsFromISR (C macro), 753
xEventGroupCreate (C++ function), 746
xEventGroupCreateStatic (C++ function), 747
xEventGroupGetBits (C macro), 755
xEventGroupGetBitsFromISR (C++ function), 753
xEventGroupSetBits (C++ function), 750
xEventGroupSetBitsFromISR (C macro), 754
xEventGroupSync (C++ function), 751
xEventGroupWaitBits (C++ function), 748
xQueueAddToSet (C++ function), 699
xQueueCreate (C macro), 701
xQueueCreateSet (C++ function), 699
xQueueCreateStatic (C macro), 701
xQueueGenericCreate (C++ function), 699
xQueueGenericCreateStatic (C++ function), 699
xQueueGenericReceive (C++ function), 695
xQueueGenericSend (C++ function), 693
xQueueGenericSendFromISR (C++ function), 692
xQueueGiveFromISR (C++ function), 693
xQueueIsQueueEmptyFromISR (C++ function), 693
xQueueIsQueueFullFromISR (C++ function), 693
xQueueOverwrite (C macro), 706
xQueueOverwriteFromISR (C macro), 711
xQueuePeek (C macro), 707
xQueuePeekFromISR (C++ function), 694
xQueueReceive (C macro), 709
xQueueReceiveFromISR (C++ function), 697
xQueueRemoveFromSet (C++ function), 700
xQueueReset (C macro), 714
xQueueSelectFromSet (C++ function), 700
xQueueSelectFromSetFromISR (C++ function), 701
xQueueSend (C macro), 705
xQueueSendFromISR (C macro), 713
xQueueSendToBack (C macro), 704
xQueueSendToBackFromISR (C macro), 711
xQueueSendToFront (C macro), 703
xQueueSendToFrontFromISR (C macro), 710
xRingbufferAddToQueueSetRead (C++ function), 767
xRingbufferCanRead (C++ function), 768
xRingbufferCreate (C++ function), 763
xRingbufferCreateNoSplit (C++ function), 763
xRingbufferGetCurFreeSize (C++ function), 767
xRingbufferGetMaxItemSize (C++ function), 767
xRingbufferPrintInfo (C++ function), 769
xRingbufferReceive (C++ function), 764
xRingbufferReceiveFromISR (C++ function), 764

xRingbufferReceiveSplit (*C++ function*), 765
xRingbufferReceiveSplitFromISR (*C++ function*), 765
xRingbufferReceiveUpTo (*C++ function*), 766
xRingbufferReceiveUpToFromISR (*C++ function*), 766
xRingbufferRemoveFromQueueSetRead (*C++ function*), 768
xRingbufferSend (*C++ function*), 763
xRingbufferSendFromISR (*C++ function*), 764
xSemaphoreCreateBinary (*C macro*), 714
xSemaphoreCreateBinaryStatic (*C macro*), 715
xSemaphoreCreateCounting (*C macro*), 726
xSemaphoreCreateCountingStatic (*C macro*), 727
xSemaphoreCreateMutex (*C macro*), 722
xSemaphoreCreateMutexStatic (*C macro*), 723
xSemaphoreCreateRecursiveMutex (*C macro*), 724
xSemaphoreCreateRecursiveMutexStatic (*C macro*), 725
xSemaphoreGetMutexHolder (*C macro*), 728
xSemaphoreGive (*C macro*), 718
xSemaphoreGiveFromISR (*C macro*), 720
xSemaphoreGiveRecursive (*C macro*), 719
xSemaphoreTake (*C macro*), 716
xSemaphoreTakeFromISR (*C macro*), 722
xSemaphoreTakeRecursive (*C macro*), 717
xTASK_SNAPSHOT (*C++ class*), 689
xTASK_SNAPSHOT::pxEndOfStack (*C++ member*), 689
xTASK_SNAPSHOT::pxTCB (*C++ member*), 689
xTASK_SNAPSHOT::pxTopOfStack (*C++ member*), 689
xTASK_STATUS (*C++ class*), 688
xTASK_STATUS::eCurrentState (*C++ member*), 688
xTASK_STATUS::pcTaskName (*C++ member*), 688
xTASK_STATUS::pxStackBase (*C++ member*), 689
xTASK_STATUS::ulRunTimeCounter (*C++ member*), 689
xTASK_STATUS::usStackHighWaterMark (*C++ member*), 689
xTASK_STATUS::uxBasePriority (*C++ member*), 688
xTASK_STATUS::uxCurrentPriority (*C++ member*), 688
xTASK_STATUS::xHandle (*C++ member*), 688
xTASK_STATUS::xTaskNumber (*C++ member*), 688
xTaskCallApplicationTaskHook (*C++ function*), 680
xTaskCreate (*C++ function*), 667
xTaskCreatePinnedToCore (*C++ function*), 666
xTaskCreateStatic (*C++ function*), 669
xTaskCreateStaticPinnedToCore (*C++ function*), 668
xTaskGetApplicationTaskTag (*C++ function*), 679
xTaskGetIdleTaskHandle (*C++ function*), 680
xTaskGetIdleTaskHandleForCPU (*C++ function*), 680
xTaskGetTickCount (*C++ function*), 678
xTaskGetTickCountFromISR (*C++ function*), 678
xTaskNotify (*C++ function*), 683
xTaskNotifyFromISR (*C++ function*), 684
xTaskNotifyGive (*C macro*), 690
xTaskNotifyWait (*C++ function*), 686
xTaskResumeAll (*C++ function*), 677
xTaskResumeFromISR (*C++ function*), 676
xTimerChangePeriod (*C macro*), 738
xTimerChangePeriodFromISR (*C macro*), 744
xTimerCreate (*C++ function*), 728
xTimerCreateStatic (*C++ function*), 731
xTimerDelete (*C macro*), 739
xTimerGetExpiryTime (*C++ function*), 734
xTimerGetPeriod (*C++ function*), 734
xTimerGetTimerDaemonTaskHandle (*C++ function*), 734
xTimerIsTimerActive (*C++ function*), 733
xTimerPendFunctionCall (*C++ function*), 736
xTimerPendFunctionCallFromISR (*C++ function*), 734
xTimerReset (*C macro*), 739
xTimerResetFromISR (*C macro*), 745
xTimerStart (*C macro*), 736
xTimerStartFromISR (*C macro*), 741
xTimerStop (*C macro*), 737
xTimerStopFromISR (*C macro*), 743