
Read the Docs Template Documentation

Release v3.0-dev-616-g9a26296

Read the Docs

Sep 12, 2017

Contents

1 Get Started	3
1.1 Introduction	3
1.2 What You Need	3
1.3 Guides	5
1.4 Setup Toolchain	23
1.5 Get ESP-IDF	30
1.6 Setup Path to ESP-IDF	31
1.7 Start a Project	31
1.8 Connect	31
1.9 Configure	32
1.10 Build and Flash	33
1.11 Monitor	33
1.12 Related Documents	34
2 API Reference	49
2.1 Wi-Fi API	49
2.2 Bluetooth API	67
2.3 Ethernet API	156
2.4 Peripherals API	163
2.5 Protocols API	320
2.6 Storage API	327
2.7 System API	373
2.8 Configuration Options	416
3 ESP32 Hardware Reference	479
3.1 ESP32 Modules and Boards	479
4 API Guides	487
4.1 General Notes About ESP-IDF Programming	487
4.2 Build System	490
4.3 Deep Sleep Wake Stubs	502
4.4 ESP32 Core Dump	504
4.5 Flash Encryption	506
4.6 High-Level Interrupts	514
4.7 JTAG Debugging	515
4.8 Partition Tables	570
4.9 Secure Boot	574

4.10 ULP coprocessor programming	579
4.11 Unit Testing in ESP32	600
4.12 Console	601
4.13 ESP32 ROM console	604
4.14 Wi-Fi Driver	607
5 Contributions Guide	639
5.1 How to Contribute	639
5.2 Before Contributing	639
5.3 Pull Request Process	640
5.4 Legal Part	640
5.5 Related Documents	640
6 Resources	655
7 Copyrights and Licenses	657
7.1 Software Copyrights	657
7.2 ROM Source Code Copyrights	658
7.3 Xtensa libhal MIT License	658
7.4 TinyBasic Plus MIT License	658
7.5 TJpgDec License	659
8 About	661

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

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

CHAPTER 1

Get Started

This document is intended to help 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.

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**

Preparation of development environment consists of three steps:

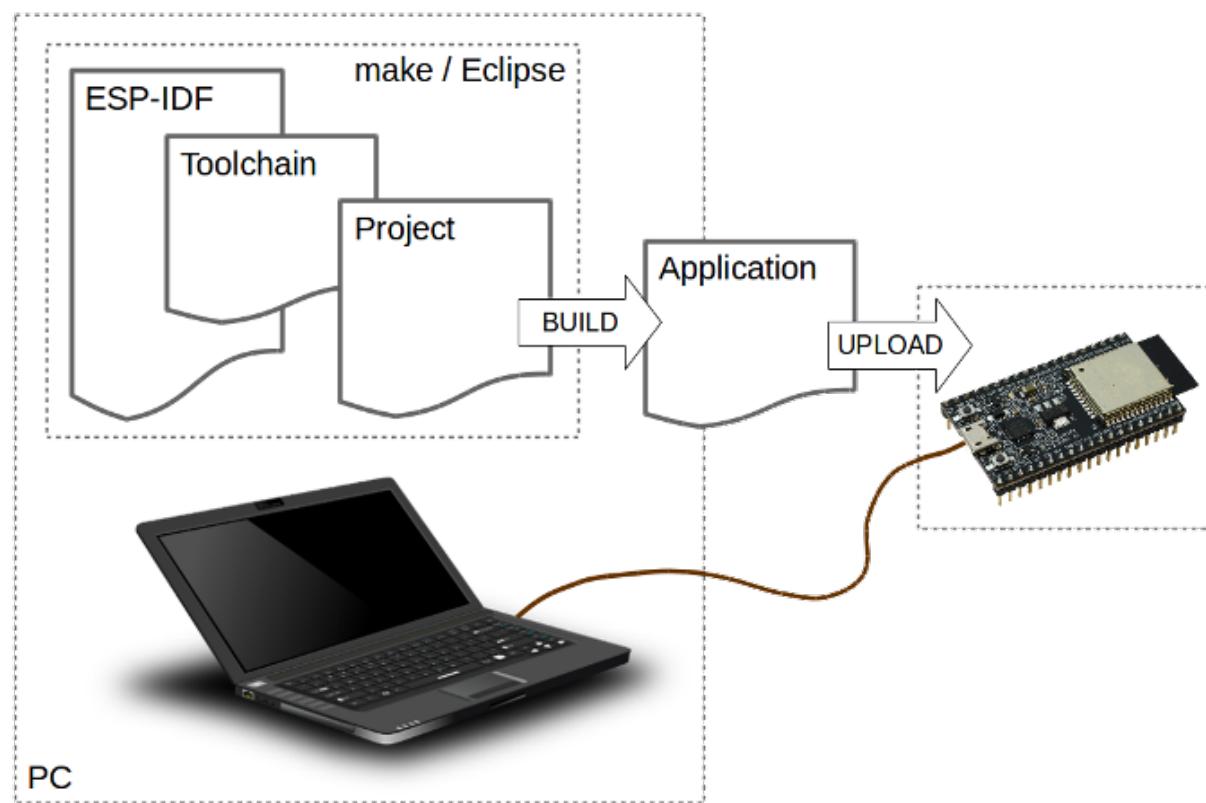


Fig. 1.1: Development of applications for ESP32

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

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

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

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

See instructions below that will walk you through these steps.

1.3 Guides

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

1.3.1 ESP32-DevKitC Getting Started Guide

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

What You Need

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

Overview

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

Functional Description

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

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

EN Reset button: pressing this button resets the system.

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

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

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

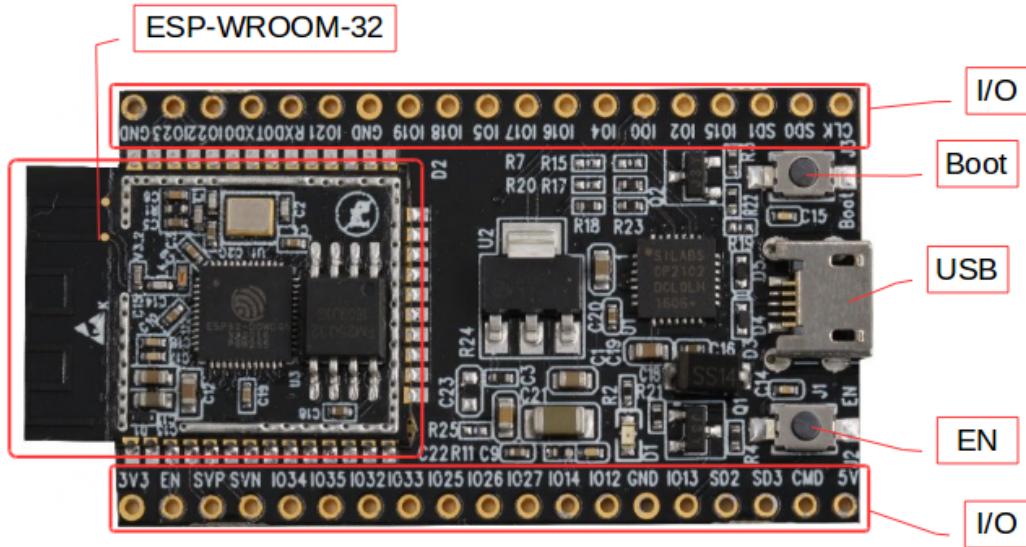


Fig. 1.2: ESP32-DevKitC board layout

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. You can find out what version you have in section [ESP-WROVER-KIT](#).

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.

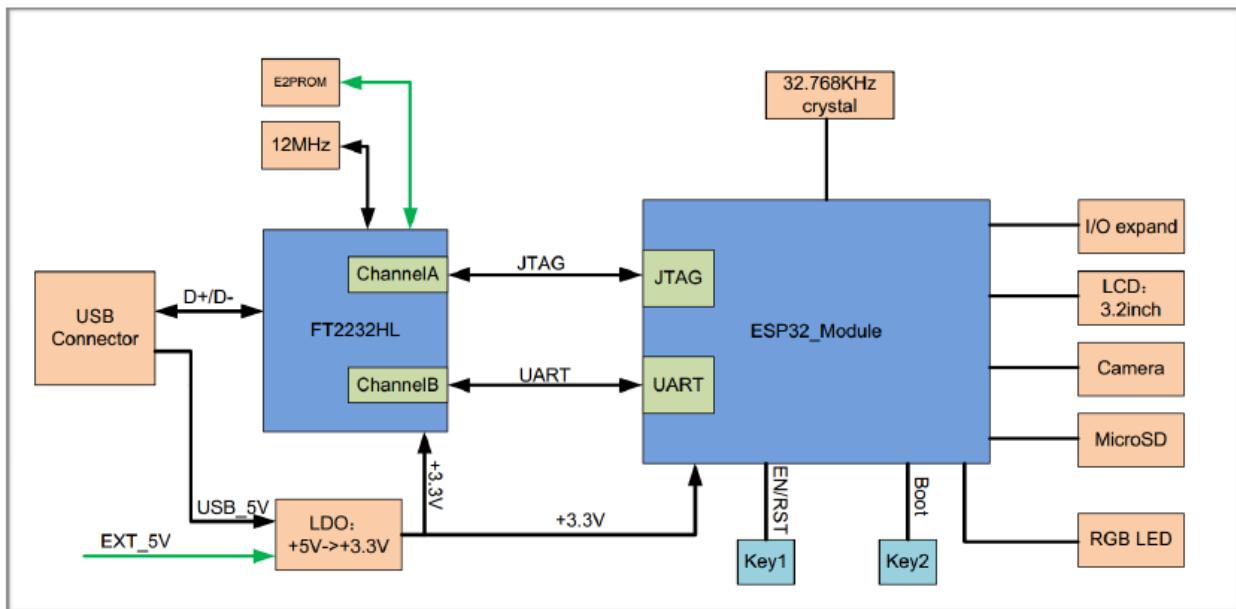


Fig. 1.3: ESP-WROVER-KIT block diagram

Functional Description

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

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

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

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

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

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

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

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

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

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

EN Reset button: pressing this button resets the system.

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

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

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

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

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

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

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

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

I/O All the pins on the ESP32 module are led out to the pin headers on the 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.

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

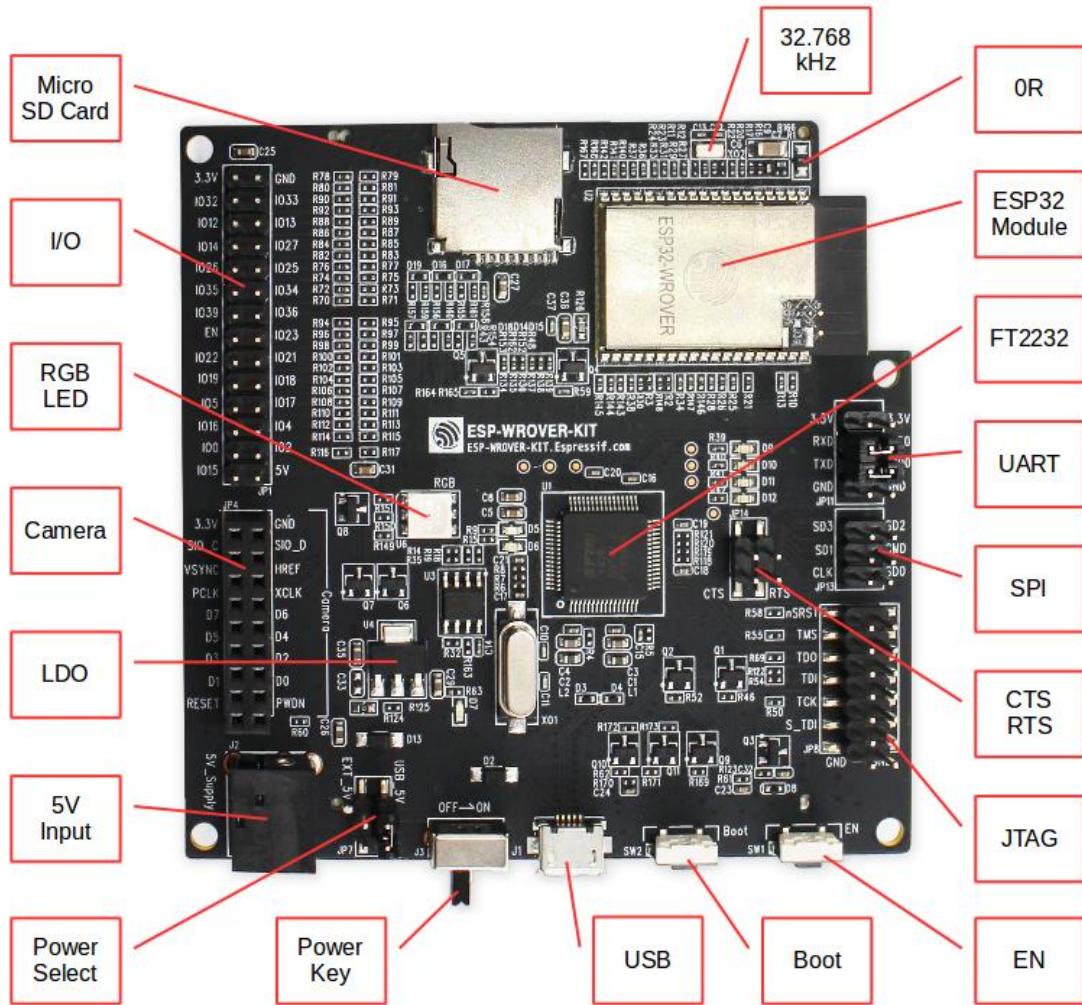


Fig. 1.4: ESP-WROVER-KIT board layout - front

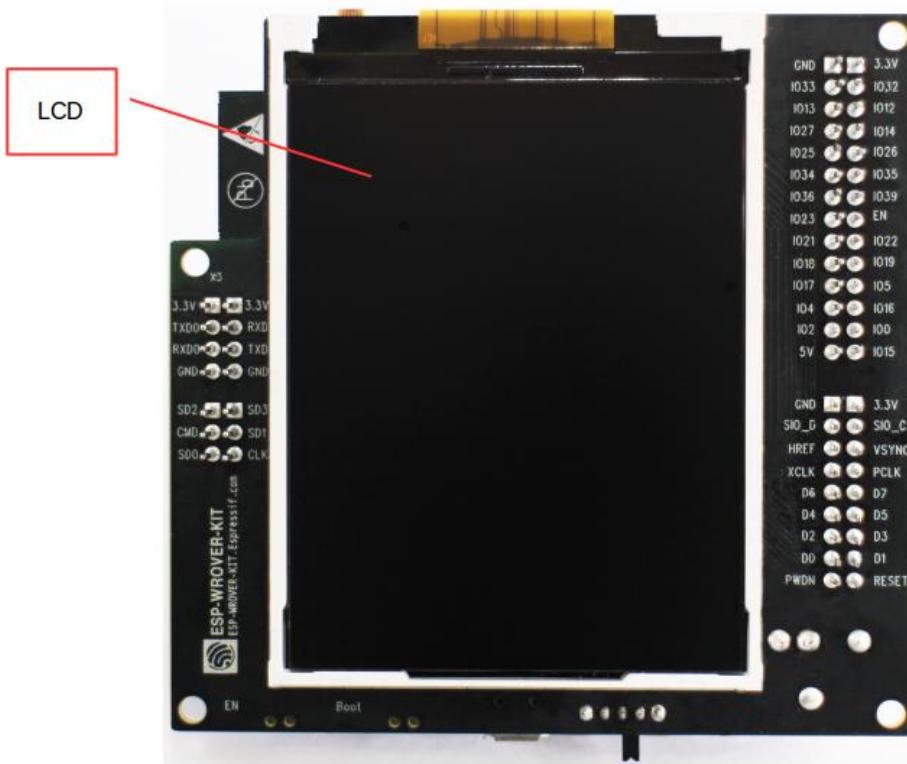
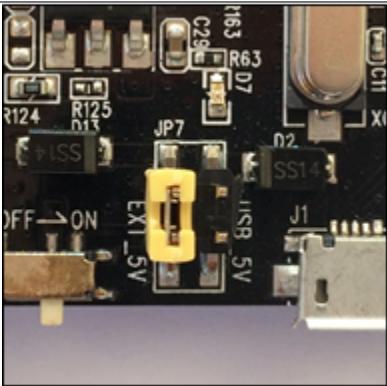
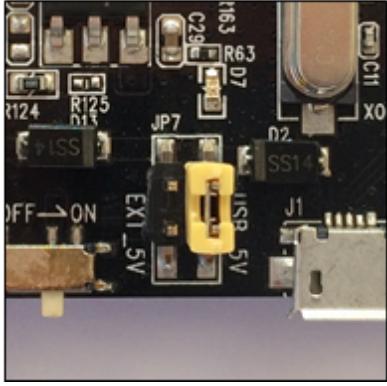
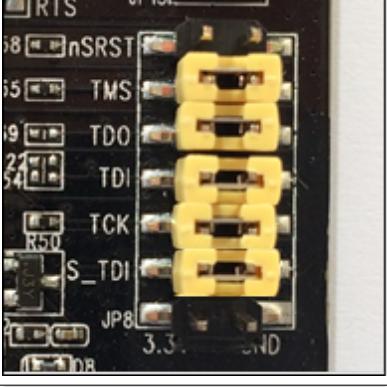
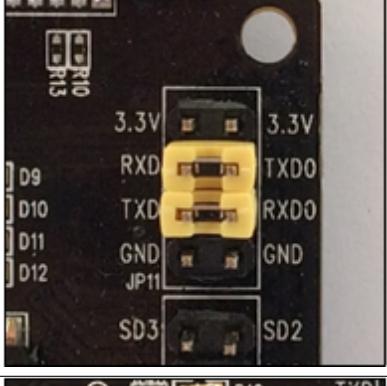
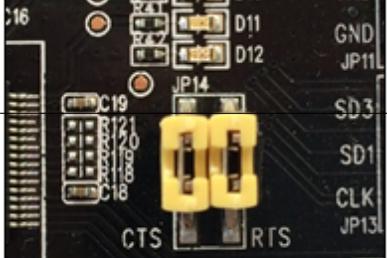


Fig. 1.5: ESP-WROVER-KIT board layout - back

Setup Options

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

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

Allocation of ESP32 Pins

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

32.768 kHz Oscillator

	ESP32 Pin
1	GPIO32
2	GPIO33

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

SPI Flash / JP13

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

JTAG / JP8

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

Camera / JP4

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

RGB LED

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

MicroSD Card / J4

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

LCD / U5

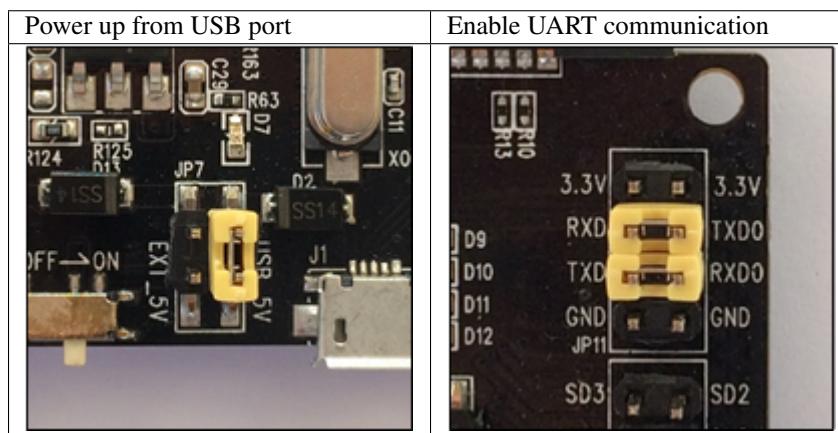
	ESP32 Pin	LCD Signal
1	GPIO18	RESET
2	GPIO19	SCL
3	GPIO21	D/C
4	GPIO22	CS
5	GPIO23	SDA
6	GPIO25	SDO
7	GPIO5	Backlight

Start Application Development

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

Initial Setup

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



Do not install any other jumpers.

Now to Development

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

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

Related Documents

- [ESP-WROVER-KIT V3 schematic \(PDF\)](#)
- [ESP32 Datasheet \(PDF\)](#)
- [ESP32-WROVER Datasheet \(PDF\)](#)
- [ESP-WROOM-32 Datasheet \(PDF\)](#)
- [JTAG Debugging](#)
- [ESP32 Modules and Boards](#)

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. You can find out what version you have in section [ESP-WROVER-KIT](#).

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.

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.

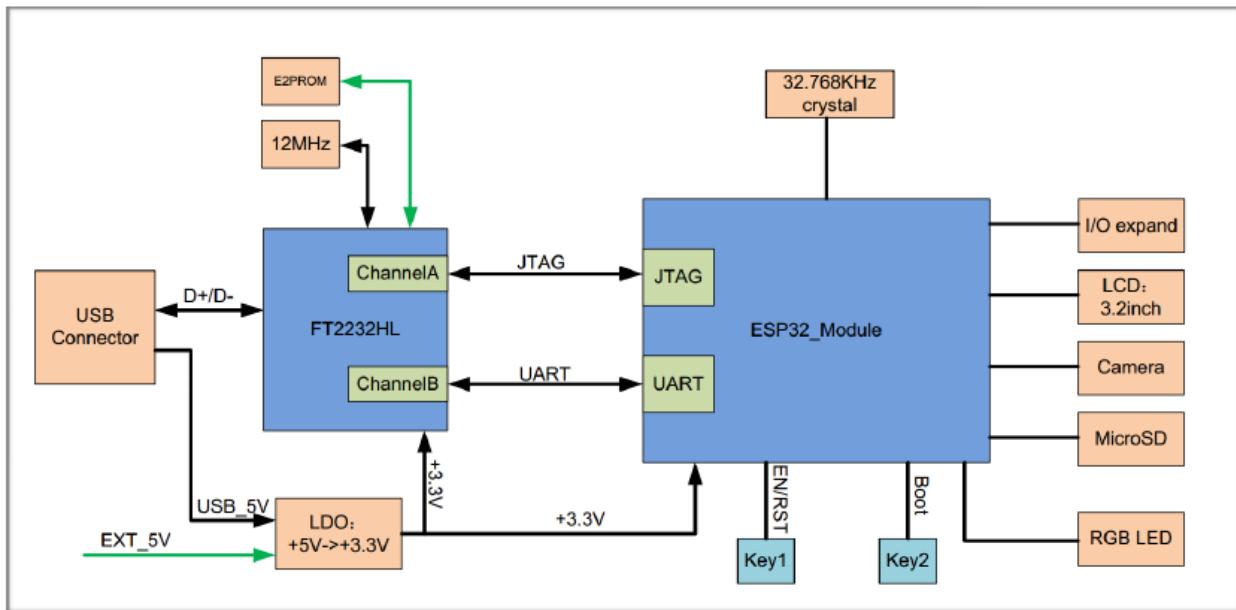


Fig. 1.6: ESP-WROVER-KIT block diagram

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

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

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

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

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

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

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

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

EN Reset button: pressing this button resets the system.

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

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

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

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

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

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

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

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

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

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

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

Setup Options

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

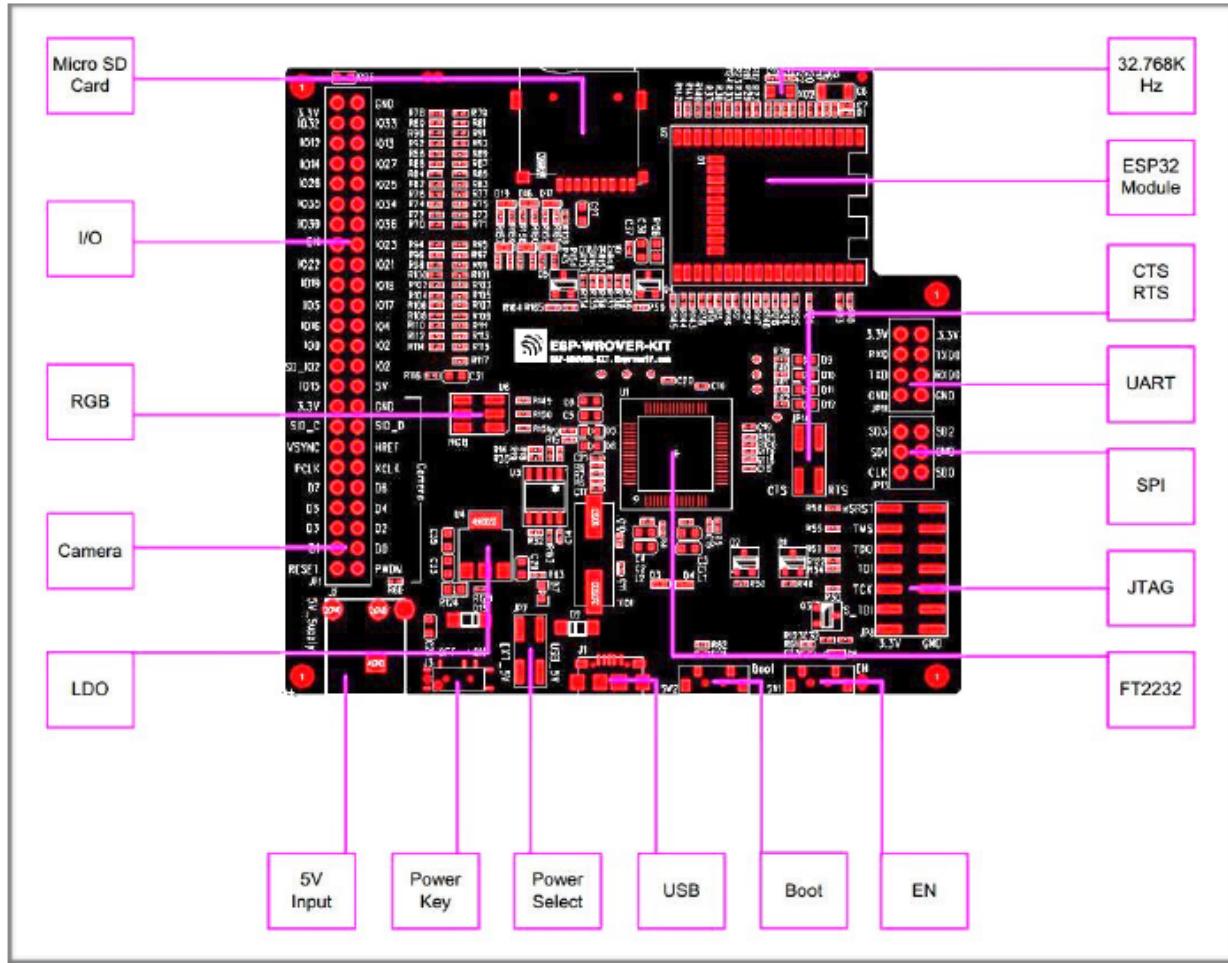


Fig. 1.7: ESP-WROVER-KIT board layout - front

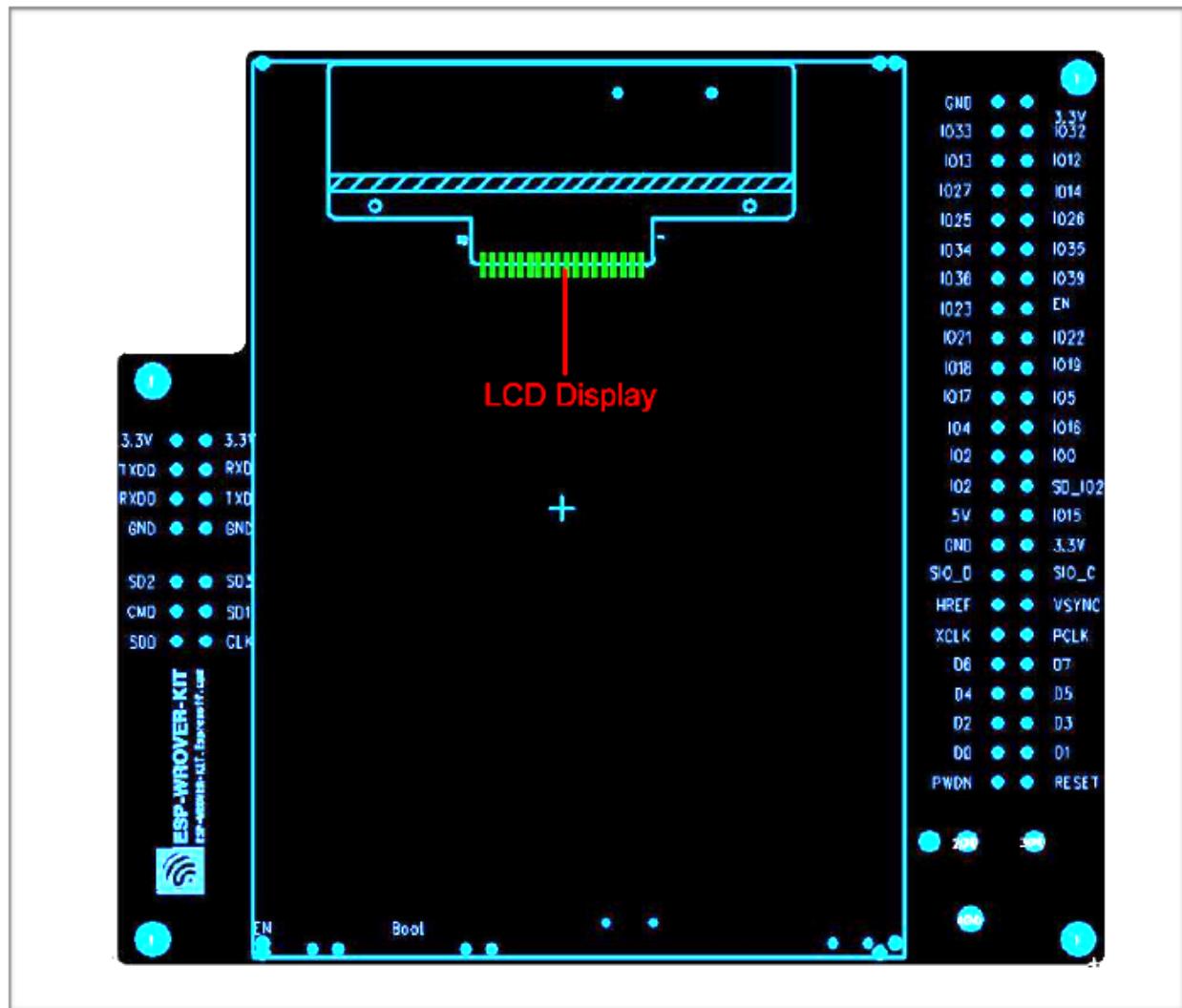
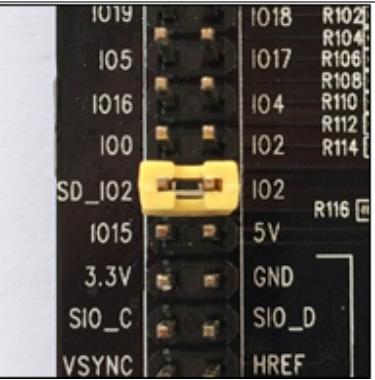
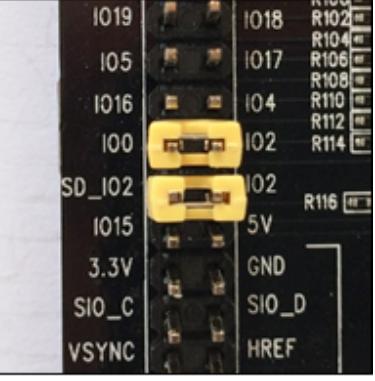
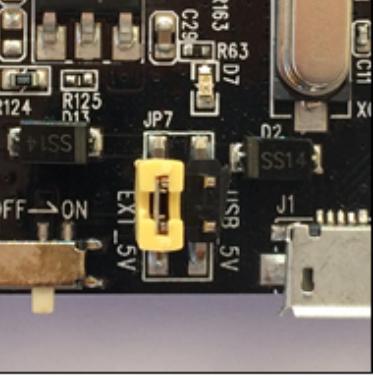
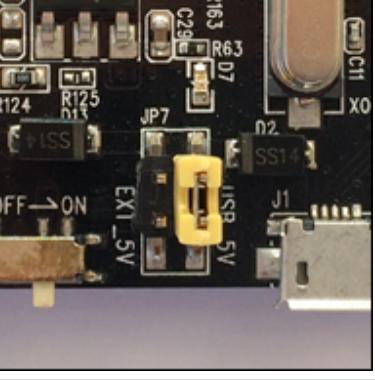
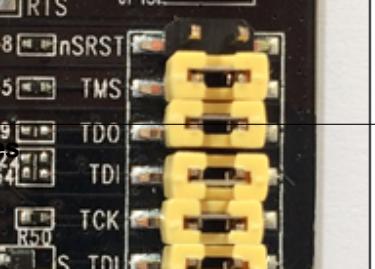


Fig. 1.8: ESP-WROVER-KIT board layout - back

Header	Jumper Setting	Description of Functionality
JP1		Enable pull up for the Micro SD Card
JP1		Assert GPIO2 low during each download (by jumping it to GPIO00)
JP7		Power ESP-WROVER-KIT board from an external power supply
JP7		Power ESP-WROVER-KIT board from an USB port
1.3. Guide		

Start Application Development

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

Initial Setup

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

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 Modules and Boards](#)

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 & MSYS zip file from dl.espressif.com:

https://dl.espressif.com/dl/esp32_win32_msys2_environment_and_toolchain-20170330.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.

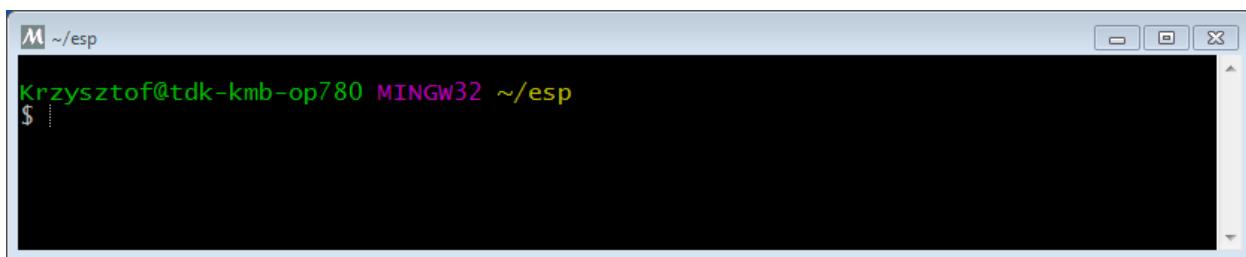


Fig. 1.9: MSYS2 MINGW32 shell window

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](#).

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-xxxxxx.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.
- During the initial update step, 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 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-win32-1.22.0-61-gab8375a-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](#).

1.4.2 Standard Setup of Toolchain for Linux

Install Prerequisites

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

- CentOS 7:

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

- Ubuntu and Debian:

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

- Arch:

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

Toolchain Setup

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

- for 64-bit Linux:

<https://dl.espressif.com/dl/xtensa-esp32-elf-linux64-1.22.0-61-gab8375a-5.2.0.tar.gz>

- for 32-bit Linux:

<https://dl.espressif.com/dl/xtensa-esp32-elf-linux32-1.22.0-61-gab8375a-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-linux64-1.22.0-61-gab8375a-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.

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/>

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

Next Steps

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

Related Documents

Setup Linux Toolchain from Scratch

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

Install Prerequisites

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

- Ubuntu and Debian:

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

- Arch:

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

Compile the Toolchain from Source

- Install dependencies:

- CentOS 7:

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

- Ubuntu pre-16.04:

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

- Ubuntu 16.04:

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

- Debian:

```
TODO
```

- Arch:

```
TODO
```

Download crosstool-NG and build it:

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

Build the toolchain:

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

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

Next Steps

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

1.4.3 Standard Setup of Toolchain for Mac OS

Install Prerequisites

- install pip:

```
sudo easy_install pip
```

- install pyserial:

```
sudo pip install pyserial
```

Toolchain Setup

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

<https://dl.espressif.com/dl/xtensa-esp32-elf-osx-1.22.0-61-gab8375a-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-1.22.0-61-gab8375a-5.2.0.tar.gz
```

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

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

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

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

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

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

Next Steps

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

Related Documents

Setup Toolchain for Mac OS from Scratch

Install Prerequisites

- install pip:

```
sudo easy_install pip
```

- install pyserial:

```
sudo pip install pyserial
```

Compile the Toolchain from Source

- Install dependencies:

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

- with MacPorts:

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

- with homebrew:

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

Create a case-sensitive filesystem image:

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

Mount it:

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

Create a symlink to your work directory:

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

Download crosstool-NG and build it:

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

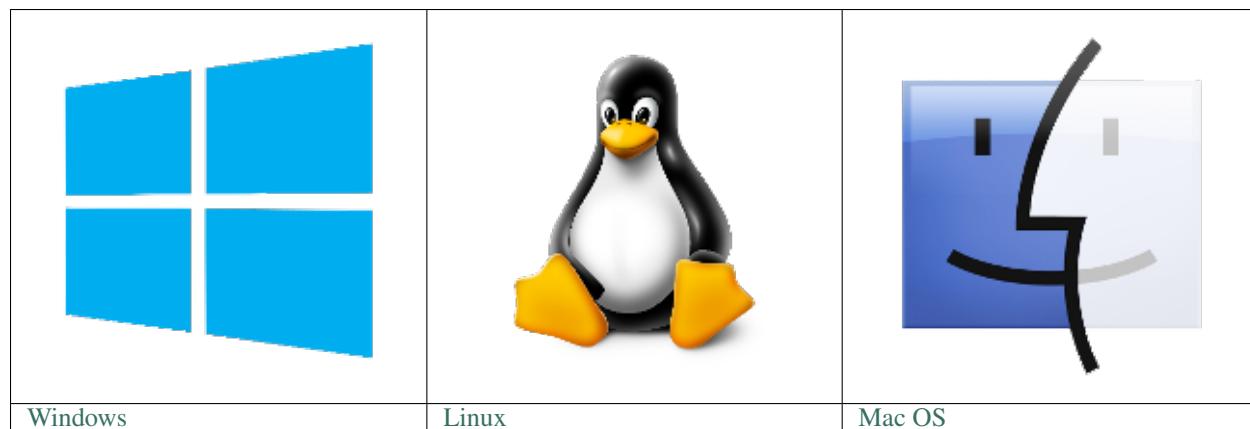
Build the toolchain:

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

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

Next Steps

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



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

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

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

1.5 Get ESP-IDF

Besides the toolchain (that contains programs to compile and build the application), you also need ESP32 specific API / libraries. They are provided by Espressif in [ESP-IDF repository](#). To get it, open terminal, navigate to the directory you want to put ESP-IDF, and clone it using `git clone` command:

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

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

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

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

Note: While cloning submodules on **Windows** platform, the `git clone` command may print some output starting '`:`' not a valid identifier.... This is a [known issue](#) but the `git clone` still succeeds without any problems.

1.6 Setup Path to ESP-IDF

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

1.7 Start a Project

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

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

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

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

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

1.8 Connect

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

1.9 Configure

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

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

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

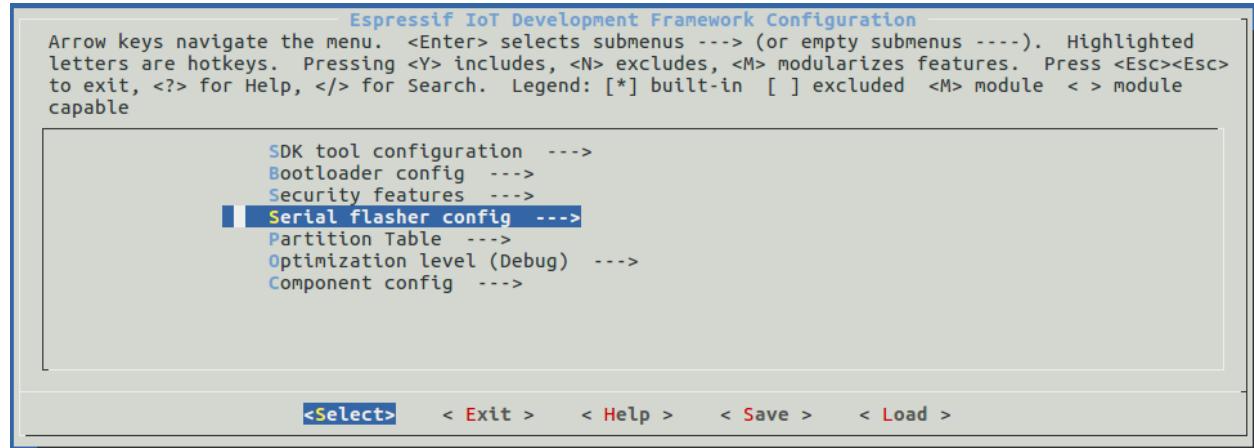


Fig. 1.10: Project configuration - Home window

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

Note: Most ESP32 development boards have a 40MHz crystal installed. However, some boards use a 26MHz crystal. If your board uses a 26MHz crystal, or you get garbage output from serial port after code upload, adjust the

CONFIG_ESP32_XTAL_FREQ option in menuconfig.

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 monitor use shortcut **Ctrl+]**. To execute `make flash` and `make monitor` in one shoot type `make flash monitor`. Check section [IDF Monitor](#) for handy shortcuts and more details on using this application.

1.12 Related Documents

1.12.1 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](#).

1.12.2 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 Windows and MacOS drivers for ESP32 boards produced by Espressif:

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

For Linux, suitable drivers should already be bundled with the operating system.

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

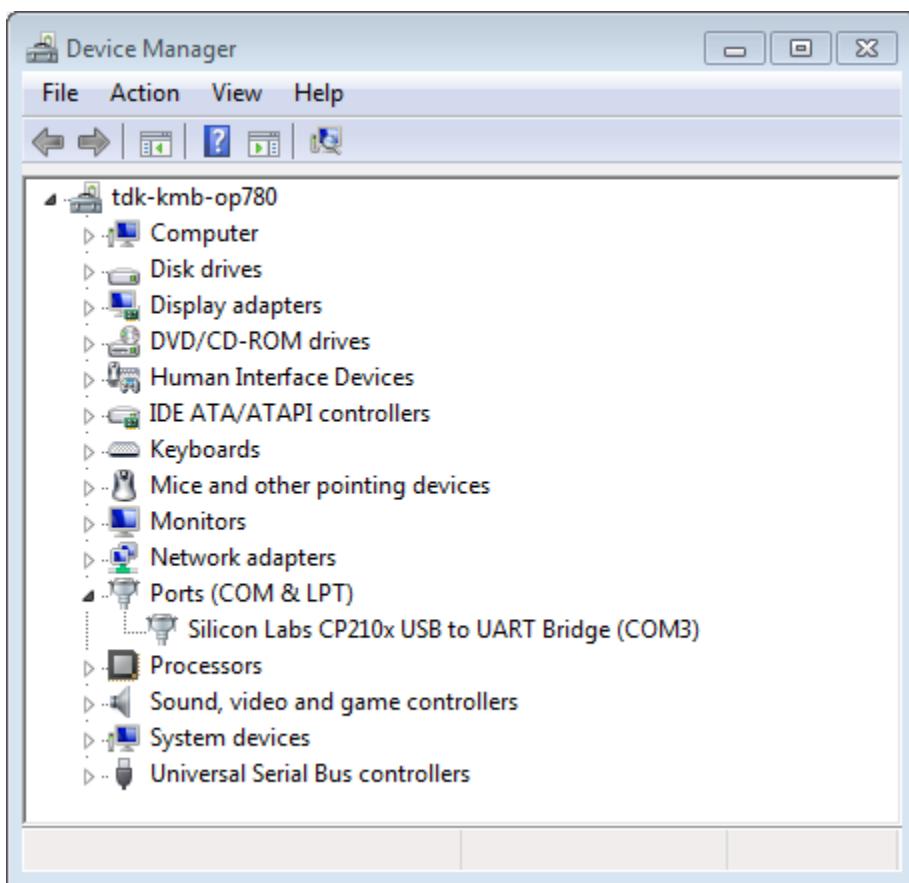


Fig. 1.11: USB to UART bridge of ESP32-DevKitC in Windows Device Manager

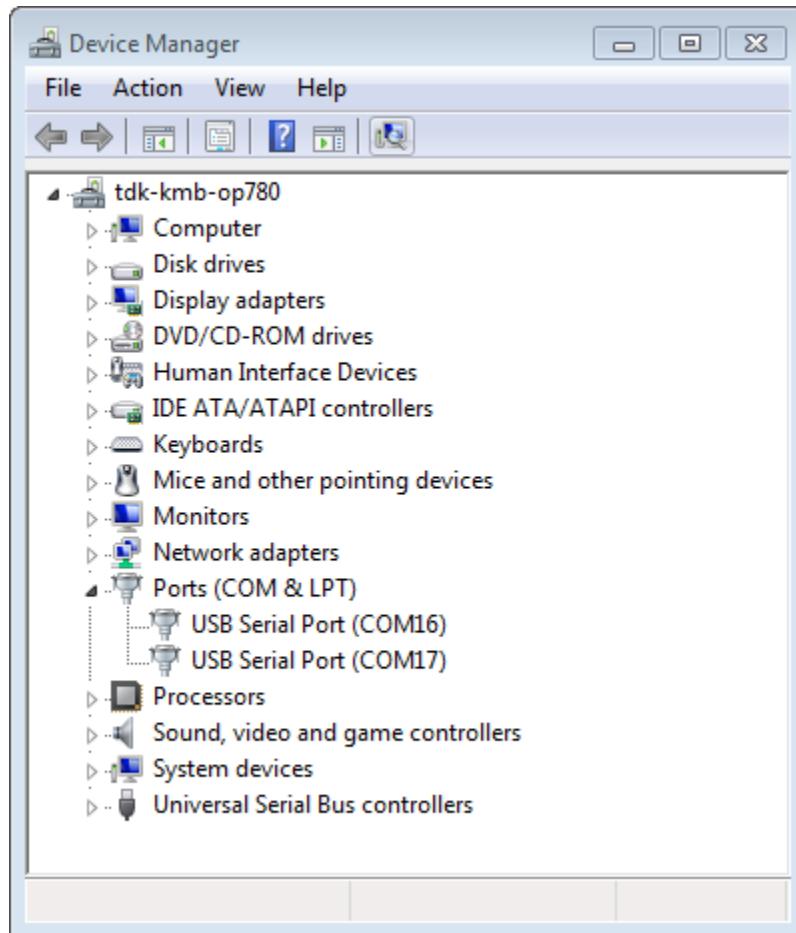


Fig. 1.12: Two USB Serial Ports of ESP-WROVER-KIT in Windows Device Manager

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. 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

...
```

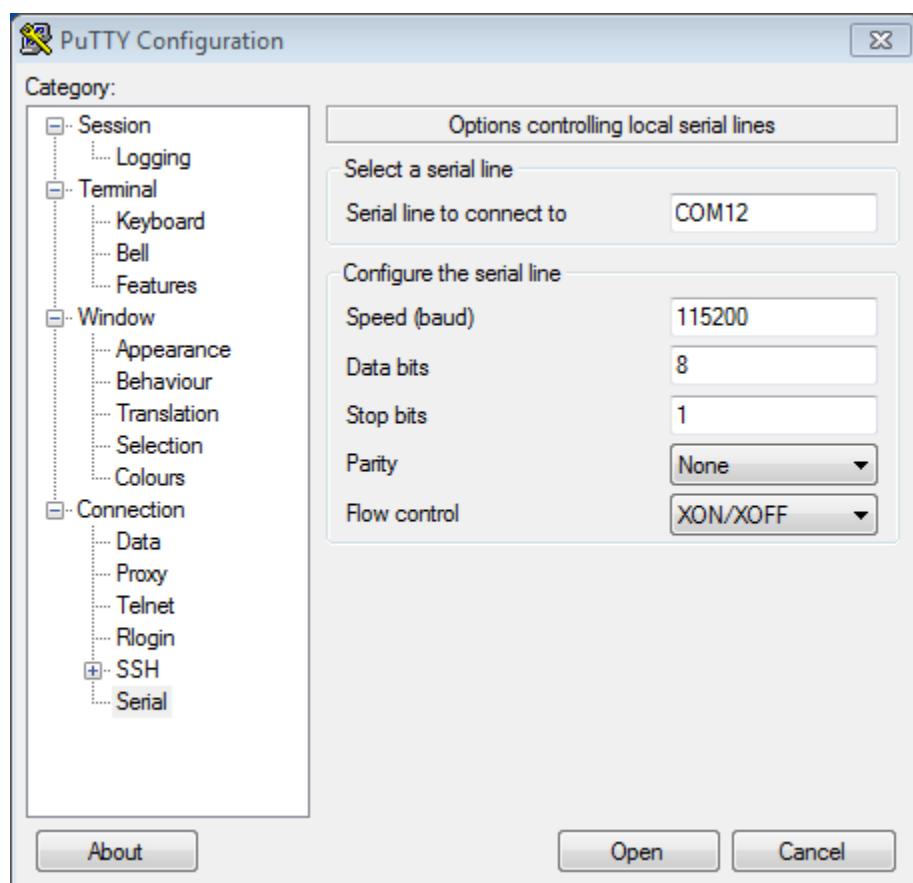


Fig. 1.13: Setting Serial Communication in PuTTY on Windows

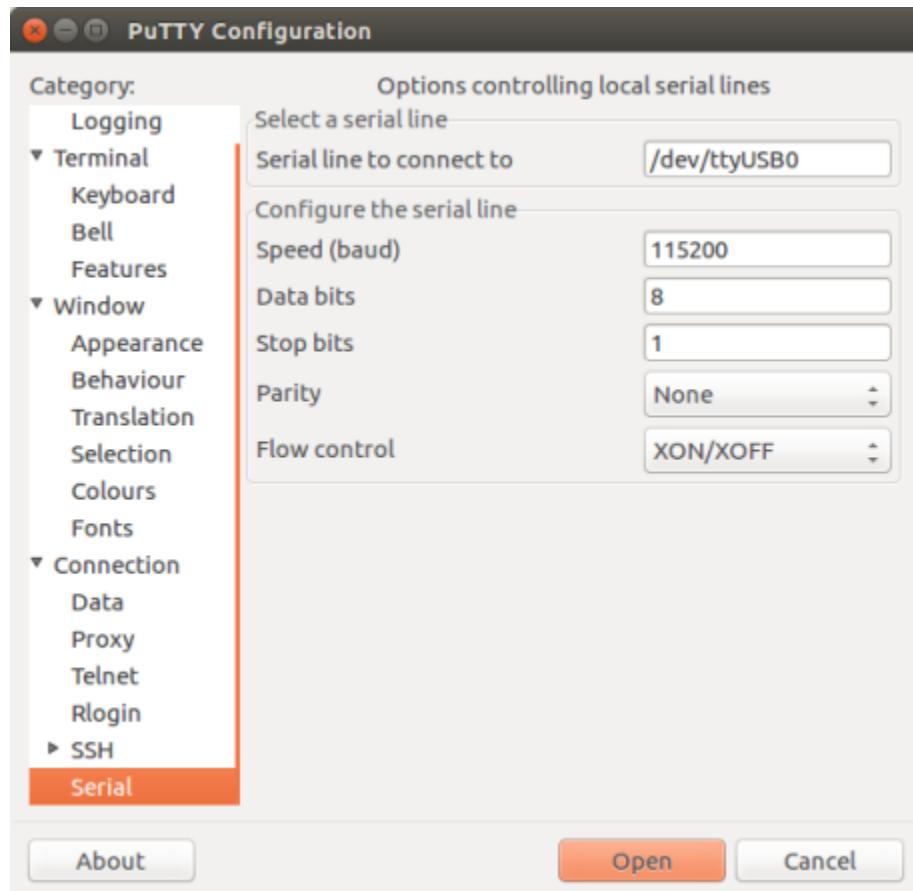


Fig. 1.14: Setting Serial Communication in PuTTY on Linux

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](#).

1.12.3 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*.

1.12.4 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 that will be installed as part of IDF setup (something/xtensa-esp32-elf/bin) if this is not already listed on the PATH.
- On macOS, add a PYTHONPATH environment variable and set it to /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages. This is so that the system Python, which has pyserial installed as part of the setup steps, overrides any built-in Eclipse Python.

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

- Click the “Providers” tab
- In the list of providers, click “CDT Cross GCC Built-in Compiler Settings”. Under “Command to get compiler specs”, replace the text \${COMMAND} at the beginning of the line with xtensa-esp32-elf-gcc. This means the full “Command to get compiler specs” should be xtensa-esp32-elf-gcc \${FLAGS} -E -P -v -dD "\${INPUTS}".
- In the list of providers, click “CDT GCC Build Output Parser” and type xtensa-esp32-elf- at the beginning of the Compiler command pattern. This means the full Compiler command pattern should be xtensa-esp32-elf-(g?cc) | ([gc]\+\+) | (clang)

Building in Eclipse

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

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

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

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

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

Flash from Eclipse

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

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

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

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

Related Documents

Eclipse IDE on Windows

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

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

Installing Eclipse IDE

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

Setting up Eclipse on Windows

Once your new Eclipse installation launches, follow these steps:

Import New Project

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

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

Project Properties

- The new project will appear under Project Explorer. Right-click the project and choose Properties from the context menu.
- Click on the “C/C++ Build” properties page (top-level):
 - Uncheck “Use default build command” and enter this for the custom build command: `python ${IDF_PATH}/tools/windows/eclipse_make.py`.
- Click on the “Environment” properties page under “C/C++ Build”:
 - Click “Add...” and enter name `BATCH_BUILD` and value `1`.
 - Click “Add...” again, and enter name `IDF_PATH`. The value should be the full path where ESP-IDF is installed. The `IDF_PATH` directory should be specified using forwards slashes not backslashes, ie `C:/Users/MyUser/Development/esp-idf`.
 - Edit the `PATH` environment variable. Delete the existing value and replace it with `C:\msys32\usr\bin;C:\msys32\mingw32\bin;C:\msys32\opt\xtensa-esp32-elf\bin` (If you installed msys32 to a different directory then you’ll need to change these paths to match).
- Click on “C/C++ General” -> “Preprocessor Include Paths, Macros, etc.” property page:
 - Click the “Providers” tab
 - * In the list of providers, click “CDT GCC Built-in Compiler Settings Cygwin”. Under “Command to get compiler specs”, replace the text `${COMMAND}` at the beginning of the line with `xtensa-esp32-elf-gcc`. This means the full “Command to get compiler specs” should be `xtensa-esp32-elf-gcc ${FLAGS} -E -P -v -dD "${INPUTS}"`.
 - * In the list of providers, click “CDT GCC Build Output Parser” and type `xtensa-esp32-elf-` at the beginning of the Compiler command pattern. This means the full Compiler command pattern should be `xtensa-esp32-elf-(g?cc) | ([gc]\+\+\+) | (clang)`

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 Cygwin 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.

1.12.5 IDF Monitor

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

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

Interacting With `idf_monitor`

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

Automatically Decoding Addresses

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

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

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

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

idf_monitor will augment the dump:

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

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

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

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

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

Launch GDB for GDBStub

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

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

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

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

Behind the scenes, the command idf_monitor runs is:

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

Quick Compile and Flash

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

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

Quick Reset

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

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`.

1.12.6 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

API Reference

2.1 Wi-Fi API

2.1.1 Wi-Fi

Overview

Instructions

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 (wifi_init_config_t *config)`

Init WiFi Alloc resource for WiFi driver, such as WiFi control structure, RX/TX buffer, WiFi NVS structure etc, this WiFi also start WiFi task.

Attention 1. This API must be called before all other WiFi API can be called

Attention 2. Always use WIFI_INIT_CONFIG_DEFAULT macro to init the config to default values, this can guarantee all the fields got correct value when more fields are added into `wifi_init_config_t` in future release. If you want to set your own initial values, overwrite the default values which are set by WIFI_INIT_CONFIG_DEFAULT, please be notified that the field ‘magic’ of `wifi_init_config_t` should always be WIFI_INIT_CONFIG_MAGIC!

Return

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

Parameters

- config: provide WiFi init configuration

`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_err_t esp_wifi_set_mode (wifi_mode_t mode)`
Set the WiFi operating mode.
Set the WiFi operating mode as station, soft-AP or station+soft-AP, The default mode is soft-AP mode.

Return

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

Parameters

- mode: WiFi operating mode

`esp_err_t esp_wifi_get_mode (wifi_mode_t *mode)`

Get current operating mode of WiFi.

Return

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

Parameters

- mode: store current WiFi mode

`esp_err_t esp_wifi_start (void)`

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

Return

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

`esp_err_t esp_wifi_stop(void)`

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

Return

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

`esp_err_t esp_wifi_restore(void)`

Restore WiFi stack persistent settings to default values.

This function will reset settings made using the following APIs:

- `esp_wifi_get_auto_connect`,
- `esp_wifi_set_protocol`,
- `esp_wifi_set_config` related
- `esp_wifi_set_mode`

Return

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

`esp_err_t esp_wifi_connect(void)`

Connect the ESP32 WiFi station to the AP.

Attention 1. This API only impact WIFI_MODE_STA or WIFI_MODE_APSTA mode

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

Return

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

`esp_err_t esp_wifi_disconnect(void)`

Disconnect the ESP32 WiFi station from the AP.

Return

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

`esp_err_t esp_wifi_clear_fast_connect (void)`

Currently this API is just an stub API.

Return

- ESP_OK: succeed
- others: fail

`esp_err_t esp_wifi_deauth_sta (uint16_t aid)`

deauthenticate all stations or associated id equals to aid

Return

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

Parameters

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

`esp_err_t esp_wifi_scan_start (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
- others: refer to error code in `esp_err.h`

Parameters

- `config`: configuration of scanning

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

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

Return

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

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

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

Return

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

Parameters

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

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

Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_NOT_STARTED`: WiFi is not started by `esp_wifi_start`
- `ESP_ERR_WIFI_ARG`: invalid argument
- `ESP_ERR_WIFI_NO_MEM`: out of memory

Parameters

- `number`: As input param, it stores max AP number `ap_records` can hold. As output param, it receives the actual AP number this API returns.
- `ap_records`: `wifi_ap_record_t` array to hold the found APs

`esp_err_t esp_wifi_sta_get_ap_info(wifi_ap_record_t *ap_info)`
Get information of AP which the ESP32 station is associated with.

Return

- `ESP_OK`: succeed
- others: fail

Parameters

- `ap_info`: the `wifi_ap_record_t` to hold AP information

`esp_err_t esp_wifi_set_ps (wifi_ps_type_t type)`

Set current power save type.

Attention Default power save type is WIFI_PS_NONE.

Return ESP_ERR_WIFI_NOT_SUPPORT: not supported yet

Parameters

- `type`: power save type

`esp_err_t esp_wifi_get_ps (wifi_ps_type_t *type)`

Get current power save type.

Attention Default power save type is WIFI_PS_NONE.

Return ESP_ERR_WIFI_NOT_SUPPORT: not supported yet

Parameters

- `type`: store current power save type

`esp_err_t esp_wifi_set_protocol (wifi_interface_t ifx, uint8_t protocol_bitmap)`

Set protocol type of specified interface The default protocol is (WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROTOCOL_11A)

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

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by `esp_wifi_init`
- ESP_ERR_WIFI_IF: invalid interface
- others: refer to error codes in `esp_err.h`

Parameters

- `ifx`: interfaces
- `protocol_bitmap`: WiFi protocol bitmap

`esp_err_t esp_wifi_get_protocol (wifi_interface_t ifx, uint8_t *protocol_bitmap)`

Get the current protocol bitmap of the specified interface.

Return

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

Parameters

- `ifx`: interface

- `protocol_bitmap`: store current WiFi protocol bitmap of interface ifx

`esp_err_t esp_wifi_set_bandwidth(wifi_interface_t ifx, wifi_bandwidth_t bw)`
Set the bandwidth of ESP32 specified interface.

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

Attention 2. WIFI_BW_HT40 is supported only when the interface support 11N

Return

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

Parameters

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

`esp_err_t esp_wifi_get_bandwidth(wifi_interface_t ifx, wifi_bandwidth_t *bw)`
Get the bandwidth of ESP32 specified interface.

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

Return

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

Parameters

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

`esp_err_t esp_wifi_set_channel(uint8_t primary, wifi_second_chan_t second)`
Set primary/secondary channel of ESP32.

Attention 1. This is a special API for sniffer

Attention 2. This API should be called after `esp_wifi_start()` or `esp_wifi_set_promiscuous()`

Return

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

Parameters

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

esp_err_t **esp_wifi_get_channel** (uint8_t **primary*, wifi_second_chan_t **second*)
Get the primary/secondary channel of ESP32.

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

Return

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

Parameters

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

esp_err_t **esp_wifi_set_country** (wifi_country_t *country*)
Set country code The default value is WIFI_COUNTRY_CN.

Return

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

Parameters

- country: country type

esp_err_t **esp_wifi_get_country** (wifi_country_t **country*)
Get country code.

Return

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

Parameters

- country: store current country

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

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

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

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

Return

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

Parameters

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

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

Get mac of specified interface.

Return

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

Parameters

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

`esp_err_t esp_wifi_set_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)`

Register the RX callback function in the promiscuous mode.

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

Return

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

Parameters

- `cb`: callback

`esp_err_t esp_wifi_set_promiscuous(bool en)`

Enable the promiscuous mode.

Return

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

Parameters

- en: false - disable, true - enable

`esp_err_t esp_wifi_get_promiscuous (bool *en)`

Get the promiscuous mode.

Return

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

Parameters

- en: store the current status of promiscuous mode

`esp_err_t esp_wifi_set_promiscuous_filter (const wifi_promiscuous_filter_t *filter)`

Enable the promiscuous filter.

Attention 1. 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 by promisucous

`esp_err_t esp_wifi_get_promiscuous_filter (wifi_promiscuous_filter_t *filter)`

Get the promiscuous filter.

Return

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

Parameters

- filter: store the current status of promiscuous filter

`esp_err_t esp_wifi_set_config (wifi_interface_t ifx, wifi_config_t *conf)`

Set the configuration of the ESP32 STA or AP.

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

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

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

Return

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

- ESP_ERR_WIFI_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_WIFI_MODE: invalid mode
- ESP_ERR_WIFI_PASSWORD: invalid password
- ESP_ERR_WIFI_NVS: WiFi internal NVS error
- others: refer to the erro code in esp_err.h

Parameters

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

`esp_err_t esp_wifi_get_config(wifi_interface_t ifx, wifi_config_t *conf)`

Get configuration of specified interface.

Return

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

Parameters

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

`esp_err_t esp_wifi_ap_get_sta_list(wifi_sta_list_t *sta)`

Get STAs associated with soft-AP.

Attention SSC only API

Return

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

Parameters

- `sta`: station list

`esp_err_t esp_wifi_set_storage(wifi_storage_t storage)`

Set the WiFi API configuration storage type.

Attention 1. The default value is WIFI_STORAGE_FLASH

Return

- ESP_OK: succeed

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

Parameters

- `storage`: : storage type

`esp_err_t esp_wifi_set_auto_connect (bool en)`

Set auto connect The default value is true.

Return

- **ESP_OK**: succeed
- **ESP_ERR_WIFI_NOT_INIT**: WiFi is not initialized by `esp_wifi_init`
- **ESP_ERR_WIFI_MODE**: WiFi internal error, the station/soft-AP control block is invalid
- others: refer to error code in `esp_err.h`

Parameters

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

`esp_err_t esp_wifi_get_auto_connect (bool *en)`

Get the auto connect flag.

Return

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

Parameters

- `en`: store current auto connect configuration

`esp_err_t esp_wifi_set_vendor_ie (bool enable, wifi_vendor_ie_type_t type, wifi_vendor_ie_id_t idx,`

`const void *vnd_ie)`

Set 802.11 Vendor-Specific Information Element.

Return

- **ESP_OK**: succeed
- **ESP_ERR_WIFI_NOT_INIT**: WiFi is not initialized by `esp_wifi_init()`
- **ESP_ERR_WIFI_ARG**: Invalid argument, including if first byte of `vnd_ie` is not `WIFI_VENDOR_IE_ELEMENT_ID` (0xDD) or second byte is an invalid length.
- **ESP_ERR_WIFI_NO_MEM**: Out of memory

Parameters

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

`esp_err_t esp_wifi_set_vendor_ie_cb (esp_vendor_ie_cb_t cb, void *ctx)`

Register Vendor-Specific Information Element monitoring callback.

Return

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

Parameters

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

`esp_err_t esp_wifi_set_max_tx_power (int8_t power)`

Set maximum WiFi transmitting power.

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

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

Return

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

Parameters

- power: Maximum WiFi transmitting power.

`esp_err_t esp_wifi_get_max_tx_power (int8_t *power)`

Get maximum WiFi transmitting power.

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

- 78: 19.5dBm
- 76: 19dBm
- 74: 18.5dBm
- 68: 17dBm
- 60: 15dBm
- 52: 13dBm
- 44: 11dBm
- 34: 8.5dBm
- 28: 7dBm
- 20: 5dBm
- 8: 2dBm
- -4: -1dBm

Return

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

Parameters

- power: Maximum WiFi transmiting power.

Structures

`struct wifi_init_config_t`

WiFi stack configuration parameters passed to esp_wifi_init call.

Public Members

`system_event_handler_t event_handler`
WiFi event handler

`wpa_crypto_funcs_t wpa_crypto_funcs`
WiFi station crypto functions when connect

`int static_rx_buf_num`
WiFi static RX buffer number

`int dynamic_rx_buf_num`
WiFi dynamic RX buffer number

`int tx_buf_type`
WiFi TX buffer type

```
int static_tx_buf_num
    WiFi static TX buffer number

int dynamic_tx_buf_num
    WiFi dynamic TX buffer number

int ampdu_enable
    WiFi AMPDU feature enable flag

int nvs_enable
    WiFi NVS flash enable flag

int nano_enable
    Nano option for printf/scan family enable flag

int tx_ba_win
    WiFi Block Ack TX window size

int rx_ba_win
    WiFi Block Ack RX window size

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

Macros

```
ESP_ERR_WIFI_OK
    No error

ESP_ERR_WIFI_FAIL
    General fail code

ESP_ERR_WIFI_NO_MEM
    Out of memory

ESP_ERR_WIFI_ARG
    Invalid argument

ESP_ERR_WIFI_NOT_SUPPORT
    Indicates that API is not supported yet

ESP_ERR_WIFI_NOT_INIT
    WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED
    WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED
    WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF
    WiFi interface error

ESP_ERR_WIFI_MODE
    WiFi mode error

ESP_ERR_WIFI_STATE
    WiFi internal state error

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

ESP_ERR_WIFI_NVS

WiFi internal NVS module error

ESP_ERR_WIFI_MAC

MAC address is invalid

ESP_ERR_WIFI_SSID

SSID is invalid

ESP_ERR_WIFI_PASSWORD

Password is invalid

ESP_ERR_WIFI_TIMEOUT

Timeout error

ESP_ERR_WIFI_WAKE_FAIL

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

WIFI_STATIC_TX_BUFFER_NUM**WIFI_DYNAMIC_TX_BUFFER_NUM****WIFI_AMPDU_ENABLED****WIFI_NVS_ENABLED****WIFI_NANO_FORMAT_ENABLED****WIFI_INIT_CONFIG_MAGIC****WIFI_DEFAULT_TX_BA_WIN****WIFI_DEFAULT_RX_BA_WIN****WIFI_INIT_CONFIG_DEFAULT**

Type Definitions

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

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

Parameters

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

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

Function signature for received Vendor-Specific Information Element callback.

Parameters

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

2.1.2 Smart Config

API Reference

Header File

- esp32/include/esp_smartconfig.h

Functions

const char *esp_smartconfig_get_version(void)

Get the version of SmartConfig.

Return

- SmartConfig version const char.

esp_err_t esp_smartconfig_start(*sc_callback_t cb*, ...)

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

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

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

Return

- ESP_OK: succeed
- others: fail

Parameters

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

esp_err_t esp_smartconfig_stop(void)

Stop SmartConfig, free the buffer taken by esp_smartconfig_start.

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

Return

- ESP_OK: succeed
- others: fail

esp_err_t esp_esptouch_set_timeout(uint8_t *time_s*)

Set timeout of SmartConfig process.

Attention Timing starts from SC_STATUS_FIND_CHANNEL status. SmartConfig will restart if timeout.

Return

- ESP_OK: succeed
- others: fail

Parameters

- time_s: range 15s~255s, offset:45s.

esp_err_t **esp_smartconfig_set_type** (*smartconfig_type_t type*)

Set protocol type of SmartConfig.

Attention If users need to set the SmartConfig type, please set it before calling esp_smartconfig_start.

Return

- ESP_OK: succeed
- others: fail

Parameters

- type: Choose from the smartconfig_type_t.

esp_err_t **esp_smartconfig_fast_mode** (bool *enable*)

Set mode of SmartConfig. default normal mode.

Attention 1. Please call it before API esp_smartconfig_start.

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

Attention 3. Two mode is compatible.

Return

- ESP_OK: succeed
- others: fail

Parameters

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

Type Definitions

typedef void (***sc_callback_t**) (*smartconfig_status_t* status, void *pdata)

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

Parameters

- status: Status of SmartConfig:
 - SC_STATUS_GETTING_SSID_PSWD : pdata is a pointer of smartconfig_type_t, means config type.
 - SC_STATUS_LINK : pdata is a pointer of struct station_config.
 - SC_STATUS_LINK_OVER : pdata is a pointer of phone's IP address(4 bytes) if pdata unequal NULL.
 - otherwise : parameter void *pdata is NULL.
- pdata: According to the different status have different values.

Enumerations

`enum smartconfig_status_t`

Values:

`SC_STATUS_WAIT` = 0

Waiting to start connect

`SC_STATUS_FIND_CHANNEL`

Finding target channel

`SC_STATUS_GETTING_SSID_PSWD`

Getting SSID and password of target AP

`SC_STATUS_LINK`

Connecting to target AP

`SC_STATUS_LINK_OVER`

Connected to AP successfully

`enum smartconfig_type_t`

Values:

`SC_TYPE_ESPTOUCH` = 0

protocol: ESPTouch

`SC_TYPE_AIRKISS`

protocol: AirKiss

`SC_TYPE_ESPTOUCH_AIRKISS`

protocol: ESPTouch and AirKiss

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

2.2 Bluetooth API

2.2.1 Controller && VHCI

Overview

Instructions

Application Example

Check `bluetooth` folder in ESP-IDF examples, which contains the following application:

- This is a BLE advertising demo with virtual HCI interface. Send Reset/ADV_PARAM/ADV_DATA/ADV_ENABLE HCI command for BLE advertising - `bluetooth/ble_adv`.

API Reference

Header File

- `bt/include/bt.h`

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_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.

Return ESP_OK - success, other - failed

Parameters

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

`esp_err_t esp_bt_controller_disable (esp_bt_mode_t mode)`
Disable BT controller.

Return ESP_OK - success, other - failed

Parameters

- `mode`: : the mode(BLE/BT/BTDM) to disable. Now only support BTDM.

`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
```

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

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

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_HDL0-9: 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_P1` 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_CONN_HDL9 = 9  
    For connection handle 9  
ESP_BLE_PWR_TYPE_ADV = 10  
    For advertising  
ESP_BLE_PWR_TYPE_SCAN = 11  
    For scan  
ESP_BLE_PWR_TYPE_DEFAULT = 12  
    For default, if not set other, it will use default value  
ESP_BLE_PWR_TYPE_NUM = 13  
    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_N14 = 0  
    Corresponding to -14dbm  
ESP_PWR_LVL_N11 = 1  
    Corresponding to -11dbm  
ESP_PWR_LVL_N8 = 2  
    Corresponding to -8dbm  
ESP_PWR_LVL_N5 = 3  
    Corresponding to -5dbm  
ESP_PWR_LVL_N2 = 4  
    Corresponding to -2dbm  
ESP_PWR_LVL_P1 = 5  
    Corresponding to 1dbm  
ESP_PWR_LVL_P4 = 6  
    Corresponding to 4dbm  
ESP_PWR_LVL_P7 = 7  
    Corresponding to 7dbm
```

2.2.2 BT COMMON

BT GENERIC DEFINES

Overview

Instructions

Application Example

Instructions

API Reference

Header File

- [bt/bluedroid/api/include/esp_bt_defs.h](#)

Structures

```
struct esp_bt_uuid_t
    UUID type.
```

Public Members

```
uint16_t len
    UUID length, 16bit, 32bit or 128bit
union esp_bt_uuid_t::@0  esp_bt_uuid_t::uuid
    UUID
```

Macros

ESP_BLUEBOARD_STATUS_CHECK(status)

ESP_BT_OCTET16_LEN

ESP_BT_OCTET8_LEN

ESP_DEFAULT_GATT_IF

Default GATT interface id.

ESP_BLE_CONN_PARAM_UNDEF

Default BLE connection param, if the value doesn't be overwritten.

ESP_BLE_IS_VALID_PARAM(x, min, max)

Check the param is valid or not.

ESP_UUID_LEN_16

ESP_UUID_LEN_32

ESP_UUID_LEN_128

ESP_BD_ADDR_LEN

Bluetooth address length.

ESP_BLE_ENC_KEY_MASK

Used to exchange the encryption key in the init key & response key.

ESP_BLE_ID_KEY_MASK

Used to exchange the IRK key in the init key & response key.

ESP_BLE_CSR_KEY_MASK

Used to exchange the CSRK key in the init key & response key.

ESP_BLE_LINK_KEY_MASK

Used to exchange the link key(this key just used in the BLE & BR/EDR coexist mode) in the init key & response key.

ESP_APP_ID_MIN

Minimum of the application id.

ESP_APP_ID_MAX

Maximum of the application id.

ESP_BD_ADDR_STR**ESP_BD_ADDR_HEX (addr)**

Type Definitions

```
typedef uint8_t esp_bt_octet16_t[ESP_BT_OCTET16_LEN]
```

```
typedef uint8_t esp_bt_octet8_t[ESP_BT_OCTET8_LEN]
```

```
typedef uint8_t esp_link_key[ESP_BT_OCTET16_LEN]
```

```
typedef uint8_t esp_bd_addr_t[ESP_BD_ADDR_LEN]
```

Bluetooth device address.

```
typedef uint8_t esp_ble_key_mask_t
```

Enumerations

```
enum esp_bt_status_t
```

Status Return Value.

Values:

```
ESP_BT_STATUS_SUCCESS = 0
```

```
ESP_BT_STATUS_FAIL
```

```
ESP_BT_STATUS_NOT_READY
```

```
ESP_BT_STATUS_NOMEM
```

```
ESP_BT_STATUS_BUSY
```

```
ESP_BT_STATUS_DONE = 5
```

```
ESP_BT_STATUS_UNSUPPORTED
```

```
ESP_BT_STATUS_PARM_INVALID
```

```
ESP_BT_STATUS_UNHANDLED
```

```
ESP_BT_STATUS_AUTH_FAILURE
ESP_BT_STATUS_RMT_DEV_DOWN = 10
ESP_BT_STATUS_AUTH_REJECTED
ESP_BT_STATUS_INVALID_STATIC RAND ADDR
ESP_BT_STATUS_PENDING
ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL
ESP_BT_STATUS_PARAM_OUT_OF_RANGE
ESP_BT_STATUS_TIMEOUT
ESP_BT_STATUS_PEER_LE_DATA_LEN_UNSUPPORTED
ESP_BT_STATUS_CONTROL_LE_DATA_LEN_UNSUPPORTED
ESP_BT_STATUS_ERR_ILLEGAL_PARAMETER_FMT
```

enum esp_bt_dev_type_t

Bluetooth device type.

Values:

```
ESP_BT_DEVICE_TYPE_BREDR = 0x01
ESP_BT_DEVICE_TYPE_BLE = 0x02
ESP_BT_DEVICE_TYPE_DUMO = 0x03
```

enum esp_ble_addr_type_t

BLE device address type.

Values:

```
BLE_ADDR_TYPE_PUBLIC = 0x00
BLE_ADDR_TYPE_RANDOM = 0x01
BLE_ADDR_TYPE_RPA_PUBLIC = 0x02
BLE_ADDR_TYPE_RPA_RANDOM = 0x03
```

BT MAIN API

Overview

Instructions

Application Example

Instructions

API Reference

Header File

- [bt/bluedroid/api/include/esp_bt_main.h](#)

Functions

`esp_bleudroid_status_t esp_bleudroid_get_status(void)`

Get bluetooth stack status.

Return Bluetooth stack status

`esp_err_t esp_bleudroid_enable(void)`

Enable bluetooth, must after `esp_bleudroid_init()`

Return

- `ESP_OK` : Succeed
- Other : Failed

`esp_err_t esp_bleudroid_disable(void)`

Disable bluetooth, must prior to `esp_bleudroid_deinit()`

Return

- `ESP_OK` : Succeed
- Other : Failed

`esp_err_t esp_bleudroid_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_bleudroid_deinit(void)`

Deinit and free the resource for bluetooth, must be after every bluetooth stuff.

Return

- `ESP_OK` : Succeed
- Other : Failed

Enumerations

`enum esp_bleudroid_status_t`

Bluetooth stack status type, to indicate whether the bluetooth stack is ready.

Values:

`ESP_BLUEDROID_STATUS_UNINITIALIZED = 0`

Bluetooth not initialized

`ESP_BLUEDROID_STATUS_INITIALIZED`

Bluetooth initialized but not enabled

`ESP_BLUEDROID_STATUS_ENABLED`

Bluetooth initialized and enabled

BT DEVICE APIs

Overview

Bluetooth device reference APIs.

Instructions

Application Example

Instructions

API Reference

Header File

- `bt/bluedroid/api/include/esp_bt_device.h`

Functions

`const uint8_t *esp_bt_dev_get_address (void)`

Get bluetooth device address. Must use after “`esp_bluedroid_enable()`”.

Return bluetooth device address (six bytes), or NULL if bluetooth stack is not enabled

`esp_err_t esp_bt_dev_set_device_name (const char *name)`

Set bluetooth device name. This function should be called after `esp_bluedroid_enable()` completes successfully.

Return

- `ESP_OK` : Succeed
- `ESP_ERR_INVALID_ARG` : if name is NULL pointer or empty, or string length out of limit
- `ESP_INVALID_STATE` : if bluetooth stack is not yet enabled
- `ESP_FAIL` : others

Parameters

- `name`: : device name to be set

2.2.3 BT LE

GAP API

Overview

Instructions

Application Example

Check `bluetooth` folder in ESP-IDF examples, which contains the following applications:

- The two demos use different GAP APIs, such like advertising, scan, set device name and others - `blue-tooth/gatt_server`, `blue-tooth/gatt_client`

API Reference

Header File

- `bt/bluedroid/api/include/esp_gap_ble_api.h`

Functions

`esp_err_t esp_ble_gap_register_callback (esp_gap_ble_cb_t callback)`

This function is called to occur gap event, such as scan result.

Return

- `ESP_OK` : success
- other : failed

Parameters

- `callback`: callback function

`esp_err_t esp_ble_gap_config_adv_data (esp_ble_adv_data_t *adv_data)`

This function is called to override the BTA default ADV parameters.

Return

- `ESP_OK` : success
- other : failed

Parameters

- `adv_data`: Pointer to User defined ADV data structure. This memory space can not be freed until callback of config_adv_data is received.

`esp_err_t esp_ble_gap_set_scan_params (esp_ble_scan_params_t *scan_params)`

This function is called to set scan parameters.

Return

- `ESP_OK` : success
- other : failed

Parameters

- `scan_params`: Pointer to User defined scan_params data structure. This memory space can not be freed until callback of set_scan_params

`esp_err_t esp_ble_gap_start_scanning (uint32_t duration)`

This procedure keep the device scanning the peer device which advertising on the air.

Return

- ESP_OK : success
- other : failed

Parameters

- duration: Keeping the scanning time, the unit is second.

`esp_err_t esp_ble_gap_stop_scanning(void)`

This function call to stop the device scanning the peer device which advertising on the air.

Return

- ESP_OK : success
- other : failed

`esp_err_t esp_ble_gap_start_advertising(esp_ble_adv_params_t *adv_params)`

This function is called to start advertising.

Return

- ESP_OK : success
- other : failed

Parameters

- adv_params: pointer to User defined adv_params data structure.

`esp_err_t esp_ble_gap_stop_advertising(void)`

This function is called to stop advertising.

Return

- ESP_OK : success
- other : failed

`esp_err_t esp_ble_gap_update_conn_params(esp_ble_conn_update_params_t *params)`

Update connection parameters, can only be used when connection is up.

Return

- ESP_OK : success
- other : failed

Parameters

- params: - connection update parameters

`esp_err_t esp_ble_gap_set_pkt_data_len(esp_bd_addr_t remote_device, uint16_t tx_data_length)`

This function is to set maximum LE data packet size.

Return

- ESP_OK : success
- other : failed

`esp_err_t esp_ble_gap_set_rand_addr (esp_bd_addr_t rand_addr)`

This function set the random address for the application.

Return

- ESP_OK : success
- other : failed

Parameters

- rand_addr: the random address which should be setting

`esp_err_t esp_ble_gap_config_local_privacy (bool privacy_enable)`

Enable/disable privacy on the local device.

Return

- ESP_OK : success
- other : failed

Parameters

- privacy_enable: - enable/disable privacy on remote device.

`esp_err_t esp_ble_gap_set_device_name (const char *name)`

Set device name to the local device.

Return

- ESP_OK : success
- other : failed

Parameters

- name: - device name.

`uint8_t *esp_ble_resolve_adv_data (uint8_t *adv_data, uint8_t type, uint8_t *length)`

This function is called to get ADV data for a specific type.

Return - ESP_OK : success

- other : failed

Parameters

- adv_data: - pointer of ADV data which to be resolved
- type: - finding ADV data type
- length: - return the length of ADV data not including type

`esp_err_t esp_ble_gap_config_adv_data_raw (uint8_t *raw_data, uint32_t raw_data_len)`

This function is called to set raw advertising data. User need to fill ADV data by self.

Return

- ESP_OK : success
- other : failed

Parameters

- raw_data: : raw advertising data
- raw_data_len: : raw advertising data length , less than 31 bytes

esp_err_t **esp_ble_gap_config_scan_rsp_data_raw** (uint8_t *raw_data, uint32_t raw_data_len)

This function is called to set raw scan response data. User need to fill scan response data by self.

Return

- ESP_OK : success
- other : failed

Parameters

- raw_data: : raw scan response data
- raw_data_len: : raw scan response data length , less than 31 bytes

esp_err_t **esp_ble_gap_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: :L 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

`esp_err_t esp_ble_clear_bond_device_list(void)`

Removes all of the device from the security database list of peer device. It manages unpairing event while connected.

Return - ESP_OK : success

- other : failed

`esp_err_t esp_ble_get_bond_device_list(void)`

Get the device from the security database list of peer device. It will return the device bonded information from the ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT event.

Return - ESP_OK : success

- other : failed

`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.

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 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_remove_bond_dev_cmpl_evt_param`

`#include <esp_gap_ble_api.h> ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT.`

Public Members

`esp_bt_status_t status`

Indicate the remove bond device operation success status

`esp_bd_addr_t bd_addr`

The device address which has been remove from the bond list

`struct ble_scan_param_cmpl_evt_param`

`#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT.`

Public Members

`esp_bt_status_t status`

Indicate the set scan param operation success status

`struct ble_scan_result_evt_param`

`#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RESULT_EVT.`

Public Members

`esp_gap_search_evt_t search_evt`

Search event type

`esp_bd_addr_t bda`

Bluetooth device address which has been searched

`esp_bt_dev_type_t dev_type`

Device type

`esp_ble_addr_type_t ble_addr_type`

Ble device address type

`esp_ble_evt_type_t ble_evt_type`

Ble scan result event type

`int rssi`

Searched device's RSSI

`uint8_t ble_adv[ESP_BLE_ADV_DATA_LEN_MAX+ESP_BLE_SCAN_RSP_DATA_LEN_MAX]`

Received EIR

`int flag`

Advertising data flag bit

```
int num_resps
    Scan result number

uint8_t adv_data_len
    Adv data length

uint8_t scan_rsp_len
    Scan response length

struct ble_scan_rsp_data_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT.
```

Public Members

esp_bt_status_t status
Indicate the set scan response data operation success status

```
struct ble_scan_rsp_data_raw_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT.
```

Public Members

esp_bt_status_t status
Indicate the set raw advertising data operation success status

```
struct ble_scan_start_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_START_COMPLETE_EVT.
```

Public Members

esp_bt_status_t status
Indicate scan start operation success status

```
struct ble_scan_stop_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT.
```

Public Members

esp_bt_status_t status
Indicate scan stop operation success status

```
struct ble_set_rand_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT.
```

Public Members

esp_bt_status_t status
Indicate set static rand address operation success status

```
struct ble_update_conn_params_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT.
```

Public Members

`esp_bt_status_t status`

Indicate update connection parameters success status

`esp_bd_addr_t bda`

Bluetooth device address

`uint16_t min_int`

Min connection interval

`uint16_t max_int`

Max connection interval

`uint16_t latency`

Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

`uint16_t conn_int`

Current connection interval

`uint16_t timeout`

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N * 10 msec

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

`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

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
```

Macros

```
ESP_BLE_ADV_FLAG_LIMIT_DISC  
    BLE_ADV_DATA_FLAG data flag bit definition used for advertising data flag  
ESP_BLE_ADV_FLAG_GEN_DISC  
ESP_BLE_ADV_FLAG_BREDR_NOT_SPT  
ESP_BLE_ADV_FLAG_DMT_CONTROLLER_SPT  
ESP_BLE_ADV_FLAG_DMT_HOST_SPT  
ESP_BLE_ADV_FLAG_NON_LIMIT_DISC  
ESP_LE_KEY_NONE  
ESP_LE_KEY_PENC  
ESP_LE_KEY_PID  
ESP_LE_KEY_PCSRK  
ESP_LE_KEY_PLK  
ESP_LE_KEY_LLK  
ESP_LE_KEY_LENC  
ESP_LE_KEY_LID  
ESP_LE_KEY_LCSRK  
ESP_LE_AUTH_NO_BOND  
ESP_LE_AUTH_BOND  
ESP_LE_AUTH_REQ_MITM  
ESP_LE_AUTH_REQ_SC_ONLY  
ESP_LE_AUTH_REQ_SC_BOND  
ESP_LE_AUTH_REQ_SC_MITM  
ESP_LE_AUTH_REQ_SC_MITM_BOND  
ESP_IO_CAP_OUT  
ESP_IO_CAP_IO  
ESP_IO_CAP_IN  
ESP_IO_CAP_NONE  
ESP_IO_CAP_KBDISP
```

ESP_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_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_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_NONE = 0

ESP_BLE_SEC_ENCRYPT

ESP_BLE_SEC_ENCRYPT_NO_MITM

ESP_BLE_SEC_ENCRYPT_MITM

enum esp_ble_sm_param_t

Values:

ESP_BLE_SM_PASSKEY = 0

ESP_BLE_SM_AUTHEN_REQ_MODE

ESP_BLE_SM_IOCAP_MODE

ESP_BLE_SM_SET_INIT_KEY

ESP_BLE_SM_SET_RSP_KEY

ESP_BLE_SM_MAX_KEY_SIZE

enum esp_ble_scan_type_t

Ble scan type.

Values:

BLE_SCAN_TYPE_PASSIVE = 0x0

Passive scan

BLE_SCAN_TYPE_ACTIVE = 0x1

Active scan

enum esp_ble_scan_filter_t

Ble scan filter type.

Values:

BLE_SCAN_FILTER_ALLOW_ALL = 0x0

Accept all :

1. advertisement packets except directed advertising packets not addressed to this device (default).

BLE_SCAN_FILTER_ALLOW_ONLY_WLST = 0x1

Accept only :

1. advertisement packets from devices where the advertiser's address is in the White list.
2. Directed advertising packets which are not addressed for this device shall be ignored.

BLE_SCAN_FILTER_ALLOW_UND_RPA_DIR = 0x2

Accept all :

1. undirected advertisement packets, and
2. directed advertising packets where the initiator address is a resolvable private address, and
3. directed advertising packets addressed to this device.

BLE_SCAN_FILTER_ALLOW_WLIST_PRA_DIR = 0x3

Accept all :

1. advertisement packets from devices where the advertiser's address is in the White list, and
2. directed advertising packets where the initiator address is a resolvable private address, and
3. directed advertising packets addressed to this device.

enum esp_gap_search_evt_t

Sub Event of ESP_GAP_BLE_SCAN_RESULT_EVT.

Values:

ESP_GAP_SEARCH_INQ_RES_EVT = 0

Inquiry result for a peer device.

ESP_GAP_SEARCH_INQ_CMPL_EVT = 1

Inquiry complete.

ESP_GAP_SEARCH_DISC_RES_EVT = 2

Discovery result for a peer device.

ESP_GAP_SEARCH_DISC_BLE_RES_EVT = 3

Discovery result for BLE GATT based service on a peer device.

ESP_GAP_SEARCH_DISC_CMPL_EVT = 4

Discovery complete.

ESP_GAP_SEARCH_DI_DISC_CMPL_EVT = 5

Discovery complete.

ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT = 6

Search cancelled

enum esp_ble_evt_type_t

Ble scan result event type, to indicate the result is scan response or advertising data or other.

Values:

ESP_BLE_EVT_CONN_ADV = 0x00

Connectable undirected advertising (ADV_IND)

ESP_BLE_EVT_CONN_DIR_ADV = 0x01

Connectable directed advertising (ADV_DIRECT_IND)

ESP_BLE_EVT_DISC_ADV = 0x02

Scannable undirected advertising (ADV_SCAN_IND)

ESP_BLE_EVT_NON_CONN_ADV = 0x03
Non connectable undirected advertising (ADV_NONCONN_IND)

ESP_BLE_EVT_SCAN_RSP = 0x04
Scan Response (SCAN_RSP)

GATT DEFINES

Overview

Instructions

Application Example

Instructions

API Reference

Header File

- [bt/bluedroid/api/include/esp_gatt_defs.h](#)

Unions

```
union esp_gatt_rsp_t
#include <esp_gatt_defs.h> GATT remote read request response type.
```

Public Members

```
esp_gatt_value_t attr_value
Gatt attribute structure
uint16_t handle
Gatt attribute handle
```

Structures

```
struct esp_gatt_id_t
Gatt id, include uuid and instance id.
```

Public Members

```
esp_bt_uuid_t uuid
UUID
uint8_t inst_id
Instance id
```

```
struct esp_gatt_srvc_id_t
Gatt service id, include id (uuid and instance id) and primary flag.
```

Public Members

```
esp_gatt_id_t id
Gatt id, include uuid and instance
```

```
bool is_primary
This service is primary or not
```

```
struct esp_attr_desc_t
Attribute description (used to create database)
```

Public Members

```
uint16_t uuid_length
UUID length
```

```
uint8_t *uuid_p
UUID value
```

```
uint16_t perm
Attribute permission
```

```
uint16_t max_length
Maximum length of the element
```

```
uint16_t length
Current length of the element
```

```
uint8_t *value
Element value array
```

```
struct esp_attr_control_t
attribute auto response flag
```

Public Members

```
uint8_t auto_rsp
if auto_rsp set to ESP_GATT_RSP_BY_APP, means the response of Write/Read operation will be replied by application. if auto_rsp set to ESP_GATT_AUTO_RSP, means the response of Write/Read operation will be replied by GATT stack automatically.
```

```
struct esp_gatts_attr_db_t
attribute type added to the gatt server database
```

Public Members

```
esp_attr_control_t attr_control
The attribute control type
```

```
esp_attr_desc_t att_desc
The attribute type
```

```
struct esp_attr_value_t
set the attribute value type
```

Public Members

```
uint16_t attr_max_len
attribute max value length
```

```
uint16_t attr_len
attribute current value length
```

```
uint8_t *attr_value
the pointer to attribute value
```

```
struct esp_gatts_incl_svc_desc_t
Gatt include service entry element.
```

Public Members

```
uint16_t start_hdl
Gatt start handle value of included service
```

```
uint16_t end_hdl
Gatt end handle value of included service
```

```
uint16_t uuid
Gatt attribute value UUID of included service
```

```
struct esp_gatts_incl128_svc_desc_t
Gatt include 128 bit service entry element.
```

Public Members

```
uint16_t start_hdl
Gatt start handle value of included 128 bit service
```

```
uint16_t end_hdl
Gatt end handle value of included 128 bit service
```

```
struct esp_gatt_value_t
Gatt attribute value.
```

Public Members

```
uint8_t value[ESP_GATT_MAX_ATTR_LEN]
Gatt attribute value
```

```
uint16_t handle
Gatt attribute handle
```

```
uint16_t offset
Gatt attribute value offset
```

```
uint16_t len
Gatt attribute value length
```

```
uint8_t auth_req  
Gatt authentication request
```

Macros

```
ESP_GATT_UUID_IMMEDIATE_ALERT_SVC  
All “ESP_GATT_UUID_xxx” is attribute types  
ESP_GATT_UUID_LINK_LOSS_SVC  
ESP_GATT_UUID_TX_POWER_SVC  
ESP_GATT_UUID_CURRENT_TIME_SVC  
ESP_GATT_UUID_REF_TIME_UPDATE_SVC  
ESP_GATT_UUID_NEXT_DST_CHANGE_SVC  
ESP_GATT_UUID_GLUCOSE_SVC  
ESP_GATT_UUID_HEALTH_THERMOM_SVC  
ESP_GATT_UUID_DEVICE_INFO_SVC  
ESP_GATT_UUID_HEART_RATE_SVC  
ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC  
ESP_GATT_UUID_BATTERY_SERVICE_SVC  
ESP_GATT_UUID_BLOOD_PRESSURE_SVC  
ESP_GATT_UUID_ALERT_NTF_SVC  
ESP_GATT_UUID_HID_SVC  
ESP_GATT_UUID_SCAN_PARAMETERS_SVC  
ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC  
ESP_GATT_UUID_CYCLING_SPEED_CADENCE_SVC  
ESP_GATT_UUID_CYCLING_POWER_SVC  
ESP_GATT_UUID_LOCATION_AND_NAVIGATION_SVC  
ESP_GATT_UUID_USER_DATA_SVC  
ESP_GATT_UUID_WEIGHT_SCALE_SVC  
ESP_GATT_UUID_PRI_SERVICE  
ESP_GATT_UUID_SEC_SERVICE  
ESP_GATT_UUID_INCLUDE_SERVICE  
ESP_GATT_UUID_CHAR_DECLARE  
ESP_GATT_UUID_CHAR_EXT_PROP  
ESP_GATT_UUID_CHAR_DESCRIPTION  
ESP_GATT_UUID_CHAR_CLIENT_CONFIG  
ESP_GATT_UUID_CHAR_SRVR_CONFIG  
ESP_GATT_UUID_CHAR_PRESENT_FORMAT
```

ESP_GATT_UUID_CHAR_AGG_FORMAT
ESP_GATT_UUID_CHAR_VALID_RANGE
ESP_GATT_UUID_EXT_RPT_REF_DESCR
ESP_GATT_UUID_RPT_REF_DESCR
ESP_GATT_UUID_GAP_DEVICE_NAME
ESP_GATT_UUID_GAP_ICON
ESP_GATT_UUID_GAP_PREF_CONN_PARAM
ESP_GATT_UUID_GAP_CENTRAL_ADDR_RESOL
ESP_GATT_UUID_GATT_SRV_CHGD
ESP_GATT_UUID_ALERT_LEVEL
ESP_GATT_UUID_TX_POWER_LEVEL
ESP_GATT_UUID_CURRENT_TIME
ESP_GATT_UUID_LOCAL_TIME_INFO
ESP_GATT_UUID_REF_TIME_INFO
ESP_GATT_UUID_NW_STATUS
ESP_GATT_UUID_NW_TRIGGER
ESP_GATT_UUID_ALERT_STATUS
ESP_GATT_UUID_RINGER_CP
ESP_GATT_UUID_RINGER_SETTING
ESP_GATT_UUID_GM_MEASUREMENT
ESP_GATT_UUID_GM_CONTEXT
ESP_GATT_UUID_GM_CONTROL_POINT
ESP_GATT_UUID_GM_FEATURE
ESP_GATT_UUID_SYSTEM_ID
ESP_GATT_UUID_MODEL_NUMBER_STR
ESP_GATT_UUID_SERIAL_NUMBER_STR
ESP_GATT_UUID_FW_VERSION_STR
ESP_GATT_UUID_HW_VERSION_STR
ESP_GATT_UUID_SW_VERSION_STR
ESP_GATT_UUID_MANU_NAME
ESP_GATT_UUID_IEEE_DATA
ESP_GATT_UUID_PNP_ID
ESP_GATT_UUID_HID_INFORMATION
ESP_GATT_UUID_HID_REPORT_MAP
ESP_GATT_UUID_HID_CONTROL_POINT
ESP_GATT_UUID_HID_REPORT

ESP_GATT_UUID_HID_PROTO_MODE
ESP_GATT_UUID_HID_BT_KB_INPUT
ESP_GATT_UUID_HID_BT_KB_OUTPUT
ESP_GATT_UUID_HID_BT_MOUSE_INPUT
ESP_GATT_HEART_RATE_MEAS
 Heart Rate Measurement.
ESP_GATT_BODY_SENSOR_LOCATION
 Body Sensor Location.
ESP_GATT_HEART_RATE_CNTL_POINT
 Heart Rate Control Point.
ESP_GATT_UUID_BATTERY_LEVEL
ESP_GATT_UUID_SC_CONTROL_POINT
ESP_GATT_UUID_SENSOR_LOCATION
ESP_GATT_UUID_RSC_MEASUREMENT
ESP_GATT_UUID_RSC_FEATURE
ESP_GATT_UUID_CSC_MEASUREMENT
ESP_GATT_UUID_CSC_FEATURE
ESP_GATT_UUID_SCAN_INT_WINDOW
ESP_GATT_UUID_SCAN_REFRESH
ESP_GATT_ILLEGAL_UUID
 GATT INVALID UUID.
ESP_GATT_ILLEGAL_HANDLE
 GATT INVALID HANDLE.
ESP_GATT_ATTR_HANDLE_MAX
 GATT attribute max handle.
ESP_GATT_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_ENCRYPTED_MITM = ESP_GATT_OK
ESP_GATT_ENCRYPTED_NO_MITM = 0x8d
ESP_GATT_NOT_ENCRYPTED = 0x8e
ESP_GATT_CONGESTED = 0x8f
ESP_GATT_DUP_REG = 0x90
ESP_GATT_ALREADY_OPEN = 0x91
ESP_GATT_CANCEL = 0x92
ESP_GATT_STACK_RSP = 0xe0
ESP_GATT_APP_RSP = 0xe1
ESP_GATT_UNKNOWN_ERROR = 0xef
ESP_GATT_CCC_CFG_ERR = 0xfd
ESP_GATT_PRC_IN_PROGRESS = 0xfe
ESP_GATT_OUT_OF_RANGE = 0xff
```

enum esp_gatt_conn_reason_t

Gatt Connection reason enum.

Values:

ESP_GATT_CONN_UNKNOWN = 0

Gatt connection unknown

ESP_GATT_CONN_L2C_FAILURE = 1

General L2cap failure

ESP_GATT_CONN_TIMEOUT = 0x08

Connection timeout

ESP_GATT_CONN_TERMINATE_PEER_USER = 0x13

Connection terminate by peer user

ESP_GATT_CONN_TERMINATE_LOCAL_HOST = 0x16

Connection terminated by local host

ESP_GATT_CONN_FAIL_ESTABLISH = 0x3e

Connection fail to establish

ESP_GATT_CONN_LMP_TIMEOUT = 0x22

Connection fail for LMP response tout

ESP_GATT_CONN_CONN_CANCEL = 0x0100

L2CAP connection cancelled

ESP_GATT_CONN_NONE = 0x0101

No connection to cancel

enum esp_gatt_auth_req_t

Gatt authentication request type.

Values:

ESP_GATT_AUTH_REQ_NONE = 0

ESP_GATT_AUTH_REQ_NO_MITM = 1

ESP_GATT_AUTH_REQ_MITM = 2

ESP_GATT_AUTH_REQ_SIGNED_NO_MITM = 3

ESP_GATT_AUTH_REQ_SIGNED_MITM = 4

enum esp_gatt_write_type_t

Gatt write type.

Values:

ESP_GATT_WRITE_TYPE_NO_RSP = 1

Gatt write attribute need no response

ESP_GATT_WRITE_TYPE_RSP

Gatt write attribute need remote response

GATT SERVER API

Overview

Instructions

Application Example

Check [bluetooth](#) folder in ESP-IDF examples, which contains the following application:

- This is a GATT server demo. Use GATT API to create a GATT server with send advertising. This GATT server can be connected and the service can be discovered - [bluetooth/gatt_server](#)

API Reference

Header File

- [bt/bluedroid/api/include/esp_gatts_api.h](#)

Functions

`esp_err_t esp_ble_gatts_register_callback(esp_gatts_cb_t callback)`

This function is called to register application callbacks with BTA GATTS module.

Return

- ESP_OK : success
- other : failed

`esp_err_t esp_ble_gatts_app_register(uint16_t app_id)`

This function is called to register application identifier.

Return

- ESP_OK : success
- other : failed

`esp_err_t esp_ble_gatts_app_unregister(esp_gatt_if_t gatts_if)`

unregister with GATT Server.

Return

- ESP_OK : success
- other : failed

Parameters

- *gatts_if*: GATT server access interface

`esp_err_t esp_ble_gatts_create_service(esp_gatt_if_t gatts_if, esp_gatt_srvc_id_t *service_id, uint16_t num_handles)`

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 in-
                                              cluded_service_handle)
```

This function is called to add an included service. After included service is included, a callback event BTA_GATTS_ADD_INCL_SRVC_EVT is reported the included service ID.

Return

- ESP_OK : success
- other : failed

Parameters

- service_handle: service handle to which this included service is to be added.
- included_service_handle: the service ID to be included.

```
esp_err_t esp_ble_gatts_add_char(uint16_t service_handle, esp_bt_uuid_t *char_uuid,
                                 esp_gatt_perm_t perm, esp_gatt_char_prop_t property,
                                 esp_attr_value_t *char_val, esp_attr_control_t *control)
```

This function is called to add a characteristic into a service.

Return

- ESP_OK : success
- other : failed

Parameters

- service_handle: service handle to which this included service is to be added.
- char_uuid: : Characteristic UUID.
- perm: : Characteristic value declaration attribute permission.
- property: : Characteristic Properties

- `char_val`: Characteristic value
- `control`: attribute response control byte

```
esp_err_t esp_ble_gatts_add_char_descr(uint16_t service_handle, esp_bt_uuid_t *scr_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 char_uuid
Characteristic 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

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

bool is_connected
Indicate it is connected or not
```

```
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_svr_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

bool is_connected
Indicate it is connected or not
```

```
struct gatts_exec_write_evt_param
#include <esp_gatts_api.h> ESP_GATTS_EXEC_WRITE_EVT.
```

Public Members

```
uint16_t conn_id
Connection id

uint32_t trans_id
Transfer id

esp_bd_addr_t bda
The bluetooth device address which been written

uint8_t exec_write_flag
Execute write flag
```

```
struct gatts_mtu_evt_param
#include <esp_gatts_api.h> ESP_GATTS_MTU_EVT.
```

Public Members

```
uint16_t conn_id
Connection id

uint16_t mtu
MTU size

struct gatts_open_evt_param
#include <esp_gatts_api.h> ESP_GATTS_OPEN_EVT.
```

Public Members

```
esp_gatt_status_t status
Operation status

struct gatts_read_evt_param
#include <esp_gatts_api.h> ESP_GATTS_READ_EVT.
```

Public Members

```
uint16_t conn_id
Connection id

uint32_t trans_id
Transfer id

esp_bd_addr_t bda
The bluetooth device address which been read

uint16_t handle
The attribute handle

uint16_t offset
Offset of the value, if the value is too long
```

```
bool is_long
The value is too long or not
```

```
bool need_rsp
The read operation need to do response
```

```
struct gatts_reg_evt_param
#include <esp_gatts_api.h> ESP_GATTS_REG_EVT.
```

Public Members

```
esp_gatt_status_t status
Operation status
```

```
uint16_t app_id
Application id which input in register API
```

```
struct gatts_rsp_evt_param
#include <esp_gatts_api.h> ESP_GATTS_RESPONSE_EVT.
```

Public Members

```
esp_gatt_status_t status
Operation status
```

```
uint16_t handle
Attribute handle which send response
```

```
struct gatts_set_attr_val_evt_param
#include <esp_gatts_api.h> ESP_GATTS_SET_ATTR_VAL_EVT.
```

Public Members

```
uint16_t srvc_handle
The service handle
```

```
uint16_t attr_handle
The attribute handle
```

```
esp_gatt_status_t status
Operation status
```

```
struct gatts_start_evt_param
#include <esp_gatts_api.h> ESP_GATTS_START_EVT.
```

Public Members

```
esp_gatt_status_t status
Operation status
```

```
uint16_t service_handle
Service attribute handle
```

```
struct gatts_stop_evt_param
#include <esp_gatts_api.h> ESP_GATTS_STOP_EVT.
```

Public Members

```
esp_gatt_status_t status
    Operation status

uint16_t service_handle
    Service attribute handle

struct gatts_write_evt_param
#include <esp_gatts_api.h> ESP_GATTS_WRITE_EVT.
```

Public Members

```
uint16_t conn_id
    Connection id

uint32_t trans_id
    Transfer id

esp_bd_addr_t bda
    The bluetooth device address which been written

uint16_t handle
    The attribute handle

uint16_t offset
    Offset of the value, if the value is too long

bool need_rsp
    The write operation need to do response

bool is_prep
    This write operation is prepare write

uint16_t len
    The write attribute value length

uint8_t *value
    The write attribute value
```

Macros

```
ESP_GATT_PREP_WRITE_CANCEL
    Prepare write flag to indicate cancel prepare write

ESP_GATT_PREP_WRITE_EXEC
    Prepare write flag to indicate execute prepare write
```

Type Definitions

```
typedef void (*esp_gatts_cb_t)(esp_gatts_cb_event_t event, esp_gatt_if_t gatts_if,
                                         esp_ble_gatts_cb_param_t *param)
```

GATT Server callback function type.

Parameters

- *event*: : Event type

- `gatts_if`: GATT server access interface, normally different `gatts_if` correspond to different profile
- `param`: Point to callback parameter, currently is union type

Enumerations

`enum esp_gatts_cb_event_t`

GATT Server callback function events.

Values:

`ESP_GATTS_REG_EVT = 0`

When register application id, the event comes

`ESP_GATTS_READ_EVT = 1`

When gatt client request read operation, the event comes

`ESP_GATTS_WRITE_EVT = 2`

When gatt client request write operation, the event comes

`ESP_GATTS_EXEC_WRITE_EVT = 3`

When gatt client request execute write, the event comes

`ESP_GATTS_MTU_EVT = 4`

When set mtu complete, the event comes

`ESP_GATTS_CONF_EVT = 5`

When receive confirm, the event comes

`ESP_GATTS_UNREG_EVT = 6`

When unregister application id, the event comes

`ESP_GATTS_CREATE_EVT = 7`

When create service complete, the event comes

`ESP_GATTS_ADD_INCL_SRVC_EVT = 8`

When add included service complete, the event comes

`ESP_GATTS_ADD_CHAR_EVT = 9`

When add characteristic complete, the event comes

`ESP_GATTS_ADD_CHAR_DESCR_EVT = 10`

When add descriptor complete, the event comes

`ESP_GATTS_DELETE_EVT = 11`

When delete service complete, the event comes

`ESP_GATTS_START_EVT = 12`

When start service complete, the event comes

`ESP_GATTS_STOP_EVT = 13`

When stop service complete, the event comes

`ESP_GATTS_CONNECT_EVT = 14`

When gatt client connect, the event comes

`ESP_GATTS_DISCONNECT_EVT = 15`

When gatt client disconnect, the event comes

`ESP_GATTS_OPEN_EVT = 16`

When connect to peer, the event comes

ESP_GATTS_CANCEL_OPEN_EVT = 17
When disconnect from peer, the event comes

ESP_GATTS_CLOSE_EVT = 18
When gatt server close, the event comes

ESP_GATTS_LISTEN_EVT = 19
When gatt listen to be connected the event comes

ESP_GATTS_CONGEST_EVT = 20
When congest happen, the event comes

ESP_GATTS_RESPONSE_EVT = 21
When gatt send response complete, the event comes

ESP_GATTS_CREAT_ATTR_TAB_EVT = 22

ESP_GATTS_SET_ATTR_VAL_EVT = 23

GATT CLIENT API

Overview

Instructions

Application Example

Check `bluetooth` folder in ESP-IDF examples, which contains the following application:

- This is a GATT client demo. This demo can scan devices, connect to the GATT server and discover the service `bluetooth/gatt_client`

API Reference

Header File

- `bt/bluedroid/api/include/esp_gattc_api.h`

Functions

esp_err_t esp_ble_gattc_register_callback(esp_gattc_cb_t callback)
This function is called to register application callbacks with GATT 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 GATT module.

Return

- ESP_OK: success
- other: failed

Parameters

- app_id: : Application Identify (UUID), for different application

esp_err_t **esp_ble_gattc_app_unregister**(*esp_gatt_if_t gattc_if*)

This function is called to unregister an application from GATT module.

Return

- ESP_OK: success
- other: failed

Parameters

- gattc_if: Gatt client access interface.

esp_err_t **esp_ble_gattc_open**(*esp_gatt_if_t gattc_if, esp_bd_addr_t remote_bda, 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.
- is_direct: direct connection or background auto connection

esp_err_t **esp_ble_gattc_close**(*esp_gatt_if_t gattc_if, uint16_t conn_id*)

Close a virtual connection to a GATT server. gattc maybe have multiple virtual GATT server connections when multiple app_id registered, this API only close one virtual GATT server connection. if there exist other virtual GATT server connections, it does not disconnect the physical connection. if you want to disconnect the physical connection directly, you can use esp_ble_gap_disconnect(*esp_bd_addr_t remote_device*).

Return

- ESP_OK: success
- other: failed

Parameters

- gattc_if: Gatt client access interface.
- conn_id: connection ID to be closed.

esp_err_t **esp_ble_gattc_send_mtu_req**(*esp_gatt_if_t gattc_if, uint16_t conn_id*)

Configure the MTU size in the GATT channel. This can be done only once per connection. Before using, use esp_ble_gatt_set_local_mtu() to configure the local MTU size.

Return

- ESP_OK: success
- other: failed

Parameters

- gattc_if: Gatt client access interface.
- conn_id: connection ID.

```
esp_err_t esp_ble_gattc_search_service(esp_gatt_if_t gattc_if, uint16_t conn_id, esp_bt_uuid_t *filter_uuid)
```

This function is called to request a GATT service discovery on a GATT server. This function report service search result by a callback event, and followed by a service search complete event.

Return

- ESP_OK: success
- other: failed

Parameters

- gattc_if: Gatt client access interface.
- conn_id: connection ID.
- filter_uuid: a UUID of the service application is interested in. If Null, discover for all services.

```
esp_err_t esp_ble_gattc_get_characteristic(esp_gatt_if_t gattc_if, uint16_t conn_id, esp_gatt_srvc_id_t *srvc_id, esp_gatt_id_t *start_char_id)
```

This function is called to find the first characteristic of the service on the given server.

Return

- ESP_OK: success
- other: failed

Parameters

- gattc_if: Gatt client access interface.
- conn_id: connection ID which identify the server.
- srvc_id: service ID
- start_char_id: the start characteristic ID

```
esp_err_t esp_ble_gattc_get_descriptor(esp_gatt_if_t gattc_if, uint16_t conn_id, esp_gatt_srvc_id_t *srvc_id, esp_gatt_id_t *char_id, esp_gatt_id_t *start_descr_id)
```

This function is called to find the descriptor of the service on the given server.

Return

- ESP_OK: success
- other: failed

Parameters

- gattc_if: Gatt client access interface.

- conn_id: connection ID which identify the server.
- srvc_id: the service ID of which the characteristic is belonged to.
- char_id: Characteristic ID, if NULL find the first available characteristic.
- start_descr_id: the start descriptor id

```
esp_err_t esp_ble_gattc_get_included_service(esp_gatt_if_t gattc_if, uint16_t conn_id,  
                                             esp_gatt_srvc_id_t *srvc_id, esp_gatt_srvc_id_t  
                                             *start_incl_srvc_id)
```

This function is called to find the first characteristic of the service on the given server.

Return

- ESP_OK: success
- other: failed

Parameters

- gattc_if: Gatt client access interface.
- conn_id: connection ID which identify the server.
- srvc_id: the service ID of which the characteristic is belonged to.
- start_incl_srvc_id: the start include service id

```
esp_err_t esp_ble_gattc_read_char(esp_gatt_if_t gattc_if, uint16_t conn_id, esp_gatt_srvc_id_t  
                                 *srvc_id, esp_gatt_id_t *char_id, esp_gatt_auth_req_t  
                                 auth_req)
```

This function is called to read a service's characteristics of the given characteristic ID.

Return

- ESP_OK: success
- other: failed

Parameters

- gattc_if: Gatt client access interface.
- conn_id: : connection ID.
- srvc_id: : service ID.
- char_id: : characteristic ID to read.
- auth_req: : authenticate request type

```
esp_err_t esp_ble_gattc_read_char_descr(esp_gatt_if_t gattc_if, uint16_t conn_id,  
                                         esp_gatt_srvc_id_t *srvc_id, esp_gatt_id_t *char_id,  
                                         esp_gatt_id_t *descr_id, 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.
- `srvc_id`: service ID.
- `char_id`: characteristic ID to read.
- `descr_id`: characteristic descriptor ID 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, esp_gatt_srvc_id_t
                                    *srvc_id, esp_gatt_id_t *char_id, 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.
- `srvc_id`: service ID.
- `char_id`: characteristic ID 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,
                                         esp_gatt_srvc_id_t *srvc_id, esp_gatt_id_t *char_id,
                                         esp_gatt_id_t      *descr_id,      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
- `srvc_id`: service ID.
- `char_id`: characteristic ID.
- `descr_id`: characteristic descriptor ID 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,
                                         esp_gatt_srvc_id_t *srvc_id,  esp_gatt_id_t *char_id,
                                         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.
- `srvc_id`: service ID.
- `char_id`: GATT characteristic ID of the service.
- `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,
                                                 esp_gatt_srvc_id_t *srvc_id,  esp_gatt_id_t *char_id,
                                                 esp_gatt_id_t *descr_id,  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.
- `srvc_id`: service ID.
- `char_id`: GATT characteristic ID of the service.
- `descr_id`: characteristic descriptor ID to 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 or 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,
 esp_gatt_srvc_id_t *srvc_id, esp_gatt_id_t
 *char_id)`

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.
- srvc_id: : pointer to GATT service ID.
- char_id: : pointer to GATT characteristic ID.

`esp_err_t esp_ble_gattc_unregister_for_notify(esp_gatt_if_t gattc_if, esp_bd_addr_t
 server_bda, esp_gatt_srvc_id_t *srvc_id,
 esp_gatt_id_t *char_id)`

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.
- srvc_id: : pointer to GATT service ID.
- char_id: : pointer to GATT characteristic ID.

esp_err_t **esp_ble_gattc_cache_refresh** (*esp_bd_addr_t remote_bda*)

Refresh the server cache store in the gattc stack of the remote device.

Return

- ESP_OK: success
- other: failed

Parameters

- *remote_bda*: remote device BD address.

Unions

union esp_ble_gattc_cb_param_t

#include <esp_gattc_api.h> Gatt client callback parameters union.

Public Members

struct esp_ble_gattc_cb_param_t::gattc_reg_evt_param reg

Gatt client callback param of ESP_GATTC_REG_EVT

struct esp_ble_gattc_cb_param_t::gattc_open_evt_param open

Gatt client callback param of ESP_GATTC_OPEN_EVT

struct esp_ble_gattc_cb_param_t::gattc_close_evt_param close

Gatt client callback param of ESP_GATTC_CLOSE_EVT

struct esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param cfg_mtu

Gatt client callback param of ESP_GATTC_CFG_MTU_EVT

struct esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param search_cmpl

Gatt client callback param of ESP_GATTC_SEARCH_CMPL_EVT

struct esp_ble_gattc_cb_param_t::gattc_search_res_evt_param search_res

Gatt client callback param of ESP_GATTC_SEARCH_RES_EVT

struct esp_ble_gattc_cb_param_t::gattc_read_char_evt_param read

Gatt client callback param of ESP_GATTC_READ_CHAR_EVT

struct esp_ble_gattc_cb_param_t::gattc_write_evt_param write

Gatt client callback param of ESP_GATTC_WRITE_DESCR_EVT

struct esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param exec_cmpl

Gatt client callback param of ESP_GATTC_EXEC_EVT

struct esp_ble_gattc_cb_param_t::gattc_notify_evt_param notify

Gatt client callback param of ESP_GATTC_NOTIFY_EVT

struct esp_ble_gattc_cb_param_t::gattc_srvc_chg_evt_param srvc_chg

Gatt client callback param of ESP_GATTC_SRVC_CHG_EVT

struct esp_ble_gattc_cb_param_t::gattc_congest_evt_param congest

Gatt client callback param of ESP_GATTC_CONGEST_EVT

struct esp_ble_gattc_cb_param_t::gattc_get_char_evt_param get_char

Gatt client callback param of ESP_GATTC_GET_CHAR_EVT

```
struct esp_ble_gattc_cb_param_t::gattc_get_descr_evt_param get_descr
    Gatt client callback param of ESP_GATTC_GET_DESCR_EVT

struct esp_ble_gattc_cb_param_t::gattc_get_incl_srvc_evt_param get_incl_srvc
    Gatt client callback param of ESP_GATTC_GET_INCL_SRVC_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 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

```
esp_gatt_status_t status
    Operation status

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_status_t status
    Operation status

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_char_evt_param
#include <esp_gattc_api.h> ESP_GATTC_GET_CHAR_EVT.
```

Public Members

```
esp_gatt_status_t status
    Operation status

uint16_t conn_id
    Connection id

esp_gatt_srvc_id_t srvc_id
    Service id, include service uuid and other information
```

`esp_gatt_id_t char_id`

Characteristic id, include characteristic uuid and other information

`esp_gatt_char_prop_t char_prop`

Characteristic property

`struct gattc_get_descr_evt_param`

`#include <esp_gattc_api.h>` ESP_GATTC_GET_DESCR_EVT.

Public Members

`esp_gatt_status_t status`

Operation status

`uint16_t conn_id`

Connection id

`esp_gatt_svc_id_t srvc_id`

Service id, include service uuid and other information

`esp_gatt_id_t char_id`

Characteristic id, include characteristic uuid and other information

`esp_gatt_id_t descr_id`

Descriptor id, include descriptor uuid and other information

`struct gattc_get_incl_srvc_evt_param`

`#include <esp_gattc_api.h>` ESP_GATTC_GET_INCL_SRVC_EVT.

Public Members

`esp_gatt_status_t status`

Operation status

`uint16_t conn_id`

Connection id

`esp_gatt_svc_id_t srvc_id`

Service id, include service uuid and other information

`esp_gatt_svc_id_t incl_srvc_id`

Included service id, include service uuid and other information

`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

`esp_gatt_svc_id_t srvc_id`

Service id, include service uuid and other information

```
esp_gatt_id_t char_id
    Characteristic id, include characteristic uuid and other information

esp_gatt_id_t descr_id
    Descriptor id, include descriptor uuid and other information

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_GATT_C_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_read_char_evt_param
#include <esp_gattc_api.h> ESP_GATT_C_READ_CHAR_EVT, ESP_GATT_C_READ_DESCR_EVT.
```

Public Members

```
esp_gatt_status_t status
    Operation status

uint16_t conn_id
    Connection id

esp_gatt_srvc_id_t srvc_id
    Service id, include service uuid and other information

esp_gatt_id_t char_id
    Characteristic id, include characteristic uuid and other information

esp_gatt_id_t descr_id
    Descriptor id, include descriptor uuid and other information

uint8_t *value
    Characteristic value

uint16_t value_type
    Characteristic value type, there is two value for this type:
    ESP_GATT_C_READ_VALUE_TYPE_VALUE(0x0000) and ESP_GATT_C_READ_VALUE_TYPE_AGG_FORMAT(0x0001)
    If the value is ESP_GATT_C_READ_VALUE_TYPE_VALUE means it is a generally value type, and
```

if is the type of ESP_GATT_READ_VALUE_TYPE_AGG_FORMAT, the unit of the value will indicate in the Characteristic aggregate format descriptor

```
uint16_t value_len  
        Characteristic value length  
  
struct gattc_reg_evt_param  
#include <esp_gattc_api.h> ESP_GATT_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_GATT_REG_FOR_NOTIFY_EVT.
```

Public Members

```
esp_gatt_status_t status  
        Operation status  
  
esp_gatt_svc_id_t srvc_id  
        Service id, include service uuid and other information  
  
esp_gatt_id_t char_id  
        Characteristic id, include characteristic uuid and other information  
  
struct gattc_search_cmpl_evt_param  
#include <esp_gattc_api.h> ESP_GATT_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_GATT_SEARCH_RES_EVT.
```

Public Members

```
uint16_t conn_id  
        Connection id  
  
esp_gatt_svc_id_t srvc_id  
        Service id, include service uuid and other information  
  
struct gattc_srvc_chg_evt_param  
#include <esp_gattc_api.h> ESP_GATT_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_GATT_C_UNREG_FOR_NOTIFY_EVT.

Public Members

`esp_gatt_status_t status`

Operation status

`esp_gatt_srvc_id_t srvc_id`

Service id, include service uuid and other information

`esp_gatt_id_t char_id`

Characteristic id, include characteristic uuid and other information

`struct gattc_write_evt_param`

`#include <esp_gattc_api.h>` ESP_GATT_C_WRITE_CHAR_EVT, ESP_GATT_C_PREP_WRITE_EVT,
ESP_GATT_C_WRITE_DESCR_EVT.

Public Members

`esp_gatt_status_t status`

Operation status

`uint16_t conn_id`

Connection id

`esp_gatt_srvc_id_t srvc_id`

Service id, include service uuid and other information

`esp_gatt_id_t char_id`

Characteristic id, include characteristic uuid and other information

`esp_gatt_id_t descr_id`

Descriptor id, include descriptor uuid and other information

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_GET_CHAR_EVT = 35

When characteristic is got from GATT server, the event comes

ESP_GATTC_GET_DESCR_EVT = 36

When characteristic descriptor is got from GATT server, the event comes

ESP_GATTC_GET_INCL_SRVC_EVT = 37

When included service is got from GATT server, 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

BLUFI API

Overview

BLUFI is a profile based GATT to config ESP32 WIFI to connect/disconnect AP or setup a softap and etc. Use should concern these things:

1. The event sent from profile. Then you need to do something as the event indicate.
2. Security reference. You can write your own Security functions such as symmetrical encryption/decryption and checksum functions. Even you can define the “Key Exchange/Negotiation” procedure.

Application Example

Check [bluetooth](#) folder in ESP-IDF examples, which contains the following application:

- This is a BLUFI demo. This demo can set ESP32’s wifi to softap/station/softap&station mode and config wifi connections - [bluetooth/blufi](#)

API Reference

Header File

- [bt/bluedroid/api/include/esp_blufi_api.h](#)

Functions

esp_err_t esp_blufi_register_callbacks (*esp_blufi_callbacks_t *callbacks*)

This function is called to receive blufi callback event.

Return ESP_OK - success, other - failed

Parameters

- callbacks: callback functions

esp_err_t esp_blufi_profile_init (void)

This function is called to initialize blufi_profile.

Return ESP_OK - success, other - failed

esp_err_t esp_blufi_profile_deinit (void)

This function is called to de-initialize blufi_profile.

Return ESP_OK - success, other - failed

```
esp_err_t esp_ble_send_wifi_conn_report(wifi_mode_t opmode, esp_ble_sto_conn_state_t
                                         sta_conn_state, uint8_t softap_conn_num,
                                         esp_ble_extra_info_t *extra_info)
```

This function is called to send wifi connection report.

Return ESP_OK - success, other - failed

Parameters

- opmode: : wifi opmode
- sta_conn_state: : station is already in connection or not
- softap_conn_num: : softap connection number
- extra_info: : extra information, such as sta_ssid, softap_ssid and etc.

```
uint16_t esp_ble_get_version(void)
```

Get BLUFI profile version.

Return Most 8bit significant is Great version, Least 8bit is Sub version

```
esp_err_t esp_ble_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_cb_param_t
#include <esp_ble_api.h> BLUFI callback parameters union.
```

Public Members

```
struct esp_ble_cb_param_t::ble_init_finish_evt_param init_finish
```

Blufi callback param of ESP_BLUFI_EVENT_INIT_FINISH

```
struct esp_ble_cb_param_t::ble_deinit_finish_evt_param deinit_finish
```

Blufi callback param of ESP_BLUFI_EVENT_DEINIT_FINISH

```
struct esp_ble_cb_param_t::ble_set_wifi_mode_evt_param wifi_mode
```

Blufi callback param of ESP_BLUFI_EVENT_INIT_FINISH

```
struct esp_ble_cb_param_t::ble_connect_evt_param connect
```

Blufi callback param of ESP_BLUFI_EVENT_CONNECT

```
struct esp_ble_cb_param_t::ble_disconnect_evt_param disconnect
```

Blufi callback param of ESP_BLUFI_EVENT_DISCONNECT

```

struct esp_bluifi_cb_param_t::bluifi_recv_sta_bssid_evt_param sta_bssid
    Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_BSSID

struct esp_bluifi_cb_param_t::bluifi_recv_sta_ssid_evt_param sta_ssid
    Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_SSID

struct esp_bluifi_cb_param_t::bluifi_recv_sta_passwd_evt_param sta_passwd
    Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_PASSWD

struct esp_bluifi_cb_param_t::bluifi_recv_softap_ssid_evt_param softap_ssid
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_SSID

struct esp_bluifi_cb_param_t::bluifi_recv_softap_passwd_evt_param softap_passwd
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD

struct esp_bluifi_cb_param_t::bluifi_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_bluifi_cb_param_t::bluifi_recv_softap_auth_mode_evt_param softap_auth_mode
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE

struct esp_bluifi_cb_param_t::bluifi_recv_softap_channel_evt_param softap_channel
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL

struct esp_bluifi_cb_param_t::bluifi_recv_username_evt_param username
    Blufi callback param of ESP_BLUFI_EVENT_RECV_USERNAME

struct esp_bluifi_cb_param_t::bluifi_recv_ca_evt_param ca
    Blufi callback param of ESP_BLUFI_EVENT_RECV_CA_CERT

struct esp_bluifi_cb_param_t::bluifi_recv_client_cert_evt_param client_cert
    Blufi callback param of ESP_BLUFI_EVENT_RECV_CLIENT_CERT

struct esp_bluifi_cb_param_t::bluifi_recv_server_cert_evt_param server_cert
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SERVER_CERT

struct esp_bluifi_cb_param_t::bluifi_recv_client_pkey_evt_param client_pkey
    Blufi callback param of ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY

struct esp_bluifi_cb_param_t::bluifi_recv_server_pkey_evt_param server_pkey
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY

struct bluifi_connect_evt_param
    #include <esp_bluifi_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 bluifi_deinit_finish_evt_param
    #include <esp_bluifi_api.h> ESP_BLUFI_EVENT_DEINIT_FINISH.

```

Public Members

```
esp_bluifi_deinit_state_t state
    De-initial status

struct blufi_disconnect_evt_param
#include <esp_bluifi_api.h> ESP_BLUFI_EVENT_DISCONNECT.
```

Public Members

```
esp_bd_addr_t remote_bda
    Blufi Remote bluetooth device address

struct blufi_init_finish_evt_param
#include <esp_bluifi_api.h> ESP_BLUFI_EVENT_INIT_FINISH.
```

Public Members

```
esp_bluifi_init_state_t state
    Initial status

struct blufi_recv_ca_evt_param
#include <esp_bluifi_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_bluifi_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_bluifi_api.h> ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY
```

Public Members

```
uint8_t *pkey
    Client Private Key point, if Client certificate not contain Key
```

```

int pkey_len
    Client Private key length

struct blufi_recv_server_cert_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SERVER_CERT

```

Public Members

```

uint8_t *cert
    Client certificate point

int cert_len
    Client certificate length

struct blufi_recv_server_pkey_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY

```

Public Members

```

uint8_t *pkey
    Client Private Key point, if Client certificate not contain Key

int pkey_len
    Client Private key length

struct blufi_recv_softap_auth_mode_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE.

```

Public Members

```

wifi_auth_mode_t auth_mode
    Authentication mode

struct blufi_recv_softap_channel_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL.

```

Public Members

```

uint8_t channel
    Authentication mode

struct blufi_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_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_bluifi_encrypt_func_t**) (uint8_t iv8, uint8_t *crypt_data, int crypt_len)
 BLUFI encrypt the data after negotiate a share key.

Return Nonnegative number is encrypted length, if error, return negative number;

Parameters

- `iv8`: : initial vector(8bit), normally, blufi core will input packet sequence number
- `crypt_data`: : plain text and encrypted data, the encrypt function must support autochthonous encrypt
- `crypt_len`: : length of plain text

typedef int (***esp_bluifi_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_bluifi_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_bluifi_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_PASWD
ESP_BLUFI_EVENT_RECV_SOFTAP_SSID
ESP_BLUFI_EVENT_RECV_SOFTAP_PASWD
ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM
ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE
ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL
ESP_BLUFI_EVENT_RECV_USERNAME
ESP_BLUFI_EVENT_RECV_CA_CERT
ESP_BLUFI_EVENT_RECV_CLIENT_CERT
ESP_BLUFI_EVENT_RECV_SERVER_CERT
ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY
ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY
ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE
```

enum esp_bluifi_sta_conn_state_t
BLUFI config status.

Values:

```
ESP_BLUFI_STA_CONN_SUCCESS = 0x00
ESP_BLUFI_STA_CONN_FAIL = 0x01
```

enum esp_bluifi_init_state_t
BLUFI init status.

Values:

```
ESP_BLUFI_INIT_OK = 0
ESP_BLUFI_INIT_FAILED = 0
```

enum esp_bluifi_deinit_state_t
BLUFI_deinit status.

Values:

```
ESP_BLUFI_DEINIT_OK = 0
ESP_BLUFI_DEINIT_FAILED = 0
```

2.2.4 CLASSIC BT

CLASSIC BLUETOOTH GAP API

Overview

Instructions

Application Example

Instructions

API Reference

Header File

- [bt/bluedroid/api/include/esp_gap_bt_api.h](#)

Functions

`esp_err_t esp_bt_gap_set_scan_mode (esp_bt_scan_mode_t mode)`

Set discoverability and connectability mode for legacy bluetooth. This function should be called after `esp_bluedroid_enable()` completes successfully.

Return

- `ESP_OK` : Succeed
- `ESP_ERR_INVALID_ARG`: if argument invalid
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- `mode`: : one of the enums of `bt_scan_mode_t`

Enumerations

`enum esp_bt_scan_mode_t`

Discoverability and Connectability mode.

Values:

`ESP_BT_SCAN_MODE_NONE` = 0

Neither discoverable nor connectable

`ESP_BT_SCAN_MODE_CONNECTABLE`

Connectable but not discoverable

`ESP_BT_SCAN_MODE_CONNECTABLE_DISCOVERABLE`

both discoverable and connectable

Bluetooth A2DP API

Overview

Instructions

Application Example

Check [bluetooth](#) folder in ESP-IDF examples, which contains the following application:

- This is a A2DP sink client demo. This demo can be discovered and connected by A2DP source device and receive the audio stream from remote device - [bluetooth/a2dp_sink](#)

API Reference

Header File

- [bt/bluedroid/api/include/esp_a2dp_api.h](#)

Functions

`esp_err_t esp_a2d_register_callback (esp_a2d_cb_t callback)`

Register application callback function to A2DP module. This function should be called only after `esp_bluedroid_enable()` completes successfully.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: if callback is a NULL function pointer

Parameters

- `callback`: A2DP sink event callback function

`esp_err_t esp_a2d_register_data_callback (esp_a2d_data_cb_t callback)`

Register A2DP sink data output function; For now the output is PCM data stream decoded from SBC format. This function should be called only after `esp_bluedroid_enable()` completes successfully.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: if callback is a NULL function pointer

Parameters

- `callback`: A2DP data callback function

`esp_err_t esp_a2d_sink_init (void)`

Initialize the bluetooth A2DP sink module. This function should be called after `esp_bluedroid_enable()` completes successfully.

Return

- `ESP_OK`: if the initialization request is sent successfully
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

`esp_err_t esp_a2d_sink_deinit (void)`

De-initialize for A2DP sink module. This function should be called only after `esp_bluetooth_enable()` completes successfully.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

`esp_err_t esp_a2d_sink_connect (esp_bd_addr_t remote_bda)`

Connect the remote bluetooth device bluetooth, must after `esp_a2d_sink_init()`

Return

- `ESP_OK`: connect request is sent to lower layer
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- `remote_bda`: remote bluetooth device address

`esp_err_t esp_a2d_sink_disconnect (esp_bd_addr_t remote_bda)`

Disconnect the remote bluetooth device.

Return

- `ESP_OK`: disconnect request is sent to lower layer
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- `remote_bda`: remote bluetooth device address

Unions

`union esp_a2d_cb_param_t`

#include <esp_a2dp_api.h> A2DP state callback parameters.

Public Members

`struct esp_a2d_cb_param_t::a2d_conn_stat_param conn_stat`

A2DP connection status

`struct esp_a2d_cb_param_t::a2d_audio_stat_param audio_stat`

audio stream playing state

`struct esp_a2d_cb_param_t::a2d_audio_cfg_param audio_cfg`

media codec configuration infomation

`struct a2d_audio_cfg_param`

#include <esp_a2dp_api.h> ESP_A2D_AUDIO_CFG_EVT.

Public Members

```
esp_bd_addr_t remote_bda  
    remote bluetooth device address  
  
esp_a2d_mcc_t mcc  
    A2DP media codec capability information
```

```
struct a2d_audio_stat_param  
#include <esp_a2dp_api.h> ESP_A2D_AUDIO_STATE_EVT.
```

Public Members

```
esp_a2d_audio_state_t state
```

one of the values from esp_a2d_audio_state_t

```
esp_bd_addr_t remote_bda  
    remote bluetooth device address
```

```
struct a2d_conn_stat_param  
#include <esp_a2dp_api.h> ESP_A2D_CONNECTION_STATE_EVT.
```

Public Members

```
esp_a2d_connection_state_t state
```

one of values from esp_a2d_connection_state_t

```
esp_bd_addr_t remote_bda  
    remote bluetooth device address
```

```
esp_a2d_disc_rsn_t disc_rsn  
    reason of disconnection for “DISCONNECTED”
```

Structures

```
struct esp_a2d_mcc_t  
A2DP media codec capabilities union.
```

Public Members

```
esp_a2d_mct_t type  
    A2DP media codec type  
  
union esp_a2d_mcc_t::@1 esp_a2d_mcc_t::cie  
    A2DP codec information element
```

Macros

```
ESP_A2D_MCT_SBC
```

Media codec types supported by A2DP.

SBC

ESP_A2D_MCT_M12
MPEG-1, 2 Audio

ESP_A2D_MCT_M24
MPEG-2, 4 AAC

ESP_A2D_MCT_ATRAC
ATRAC family

ESP_A2D_MCT_NON_A2DP

ESP_A2D_CIE_LEN_SBC

ESP_A2D_CIE_LEN_M12

ESP_A2D_CIE_LEN_M24

ESP_A2D_CIE_LEN_ATRAC

Type Definitions

typedef uint8_t esp_a2d_mct_t

typedef void (*esp_a2d_cb_t)(esp_a2d_cb_event_t event, esp_a2d_cb_param_t *param)
A2DP profile callback function type.

Parameters

- **event:** : Event type
- **param:** : Pointer to callback parameter

typedef void (*esp_a2d_data_cb_t)(const uint8_t *buf, uint32_t len)
A2DP profile data callback function.

Parameters

- **buf:** : data received from A2DP source device and is PCM format decoder from SBC decoder; buf references to a static memory block and can be overwritten by upcoming data
- **len:** : size(in bytes) in buf

Enumerations

enum esp_a2d_connection_state_t
Bluetooth A2DP connection states.

Values:

ESP_A2D_CONNECTION_STATE_DISCONNECTED = 0
connection released

ESP_A2D_CONNECTION_STATE_CONNECTING
connecting remote device

ESP_A2D_CONNECTION_STATE_CONNECTED
connection established

ESP_A2D_CONNECTION_STATE_DISCONNECTING
disconnecting remote device

enum esp_a2d_disc_rsn_t

Bluetooth A2DP disconnection reason.

Values:

ESP_A2D_DISC_RSN_NORMAL = 0

Finished disconnection that is initiated by local or remote device

ESP_A2D_DISC_RSN_ABNORMAL

Abnormal disconnection caused by signal loss

enum esp_a2d_audio_state_t

Bluetooth A2DP datapath states.

Values:

ESP_A2D_AUDIO_STATE_REMOTE_SUSPEND = 0

audio stream datapath suspended by remote device

ESP_A2D_AUDIO_STATE_STOPPED

audio stream datapath stopped

ESP_A2D_AUDIO_STATE_STARTED

audio stream datapath started

enum esp_a2d_cb_event_t

A2DP callback events.

Values:

ESP_A2D_CONNECTION_STATE_EVT = 0

connection state changed event

ESP_A2D_AUDIO_STATE_EVT = 1

audio stream transmission state changed event

ESP_A2D_AUDIO_CFG_EVT = 2

audio codec is configured

BT AVRCP APIs

Overview

Bluetooth AVRCP reference APIs.

Instructions

Application Example

Instructions

API Reference

Header File

- [bt/bluedroid/api/include/esp_avrc_api.h](#)

Functions

`esp_err_t esp_avrc_ct_register_callback(esp_avrc_ct_cb_t callback)`

Register application callbacks to AVRCP module; for now only AVRCP Controller role is supported. This function should be called after `esp_bluetooth_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_bluetooth_enable()` completes successfully.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

`esp_err_t esp_avrc_ct_deinit(void)`

De-initialize AVRCP controller module. This function should be called after `esp_bluetooth_enable()` completes successfully.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

`esp_err_t esp_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 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
```

```
uint32_t feat_mask
AVRC feature mask of remote device
```

```
esp_bd_addr_t remote_bda
remote bluetooth device address
```

```
struct avrc_ct_psth_rsp_param
#include <esp_avrc_api.h> ESP_AVRC_CT_PASSTHROUGH_RSP_EVT.
```

Public Members

```
uint8_t t1
transaction label, 0 to 15
```

```
uint8_t key_code
passthrough command code
```

```
uint8_t key_state
0 for PRESSED, 1 for RELEASED
```

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 command/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

`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_MAX_EVT

Example code for this API section is provided in [bluetooth](#) directory of ESP-IDF examples.

2.3 Ethernet API

2.3.1 ETHERNET

Application Example

Ethernet example: [ethernet/ethernet](#).

PHY Interfaces

The configured PHY model(s) are set in software by configuring the `eth_config_t` structure for the given PHY.

Headers include a default configuration structure. These default configurations will need some members 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 Should be called after esp_eth_init

Return

- ESP_OK
- ESP_FAIL

`esp_err_t esp_eth_disable(void)`
Disable ethernet interface.

Note Should be called after esp_eth_init

Return

- ESP_OK
- ESP_FAIL

`void esp_eth_get_mac(uint8_t mac[6])`
Get mac addr.

Note mac addr must be a valid unicast address

Parameters

- `mac`: start address of mac address.

`void esp_eth_smi_write(uint32_t reg_num, uint16_t value)`
Read phy reg with smi interface.

Note phy base addr must be right.

Parameters

- `reg_num`: phy reg num.
- `value`: value which write to phy reg.

`uint16_t esp_eth_smi_read(uint32_t reg_num)`
Read phy reg with smi interface.

Note phy base addr must be right.

Return value what read from phy reg

Parameters

- `reg_num`: phy reg num.

`esp_err_t esp_eth_smi_wait_value(uint32_t reg_num, uint16_t value, uint16_t value_mask, int timeout_ms)`
Continuously read a PHY register over SMI interface, wait until the register has the desired value.

Note PHY base address must be right.

Return ESP_OK if desired value matches, ESP_ERR_TIMEOUT if timed out.

Parameters

- `reg_num`: PHY register number
- `value`: Value to wait for (masked with `value_mask`)
- `value_mask`: Mask of bits to match in the register.
- `timeout_ms`: Timeout to wait for this value (milliseconds). 0 means never timeout.

`static esp_err_t esp_eth_smi_wait_set(uint32_t reg_num, uint16_t value_mask, int timeout_ms)`
Continuously read a PHY register over SMI interface, wait until the register has all bits in a mask set.

Note PHY base address must be right.

Return ESP_OK if desired value matches, ESP_ERR_TIMEOUT if timed out.

Parameters

- reg_num: PHY register number
- value_mask: Value mask to wait for (all bits in this mask must be set)
- timeout_ms: Timeout to wait for this value (milliseconds). 0 means never timeout.

```
void esp_eth_free_rx_buf (void *buf)
Free emac rx buf.
```

Note buf can not be null, and it is tcpip input buf.

Parameters

- buf: start address of receive packet data.

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_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_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.

2.4 Peripherals API

2.4.1 Analog to Digital Converter

Overview

ESP32 integrates two 12-bit SAR (“Successive Approximation Register”) ADCs (Analog to Digital Converters) and supports measurements on 18 channels (analog enabled pins). Some of these pins can be used to build a programmable gain amplifier which is used for the measurement of small analog signals.

The ADC driver API currently only supports ADC1 (9 channels, attached to GPIOs 32-39).

Taking an ADC reading involves configuring the ADC with the desired precision and attenuation settings, and then calling `adc1_get_raw()` to read the channel.

It is also possible to read the internal hall effect sensor via ADC1.

Application Example

A full example using the ADC driver and the `esp_adc_cal` is available in esp-idf: [peripherals/adc](#)

Reading voltage on ADC1 channel 0 (GPIO 36):

```
#include <driver/adc.h>

...
adc1_config_width(ADC_WIDTH_12Bit);
```

```
adc1_config_channel_atten(ADC1_CHANNEL_0, ADC_ATTEN_0db);  
int val = adc1_get_raw(ADC1_CHANNEL_0);
```

Reading the internal hall effect sensor:

```
#include <driver/adc.h>  
  
...  
  
adc1_config_width(ADC_WIDTH_12Bit);  
int val = hall_sensor_read();
```

The value read in both these examples is 12 bits wide (range 0-4095).

API Reference

Header File

- [driver/include/driver/adc.h](#)

Functions

`esp_err_t adc1_config_width(adc_bits_width_t width_bit)`

Configure ADC1 capture width.

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 adc1_config_channel_atten(adc1_channel_t channel, adc_atten_t atten)`

Configure the ADC1 channel, including setting attenuation.

The default ADC full-scale voltage is 1.1V. To read higher voltages (up to the pin maximum voltage, usually 3.3V) requires setting >0dB signal attenuation for that ADC channel.

Note This function also configures the input GPIO pin mux to connect it to the ADC1 channel. It must be called before calling `adc1_get_raw()` for this channel.

When VDD_A is 3.3V:

- 0dB 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 ADC1 configured bit width, this value is: 4095 for 12-bits, 2047 for 11-bits, 1023 for 10-bits, 511 for 9 bits.)

Note At 11dB attenuation the maximum voltage is limited by VDD_A, not the full scale voltage.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- channel: ADC1 channel to configure
- atten: Attenuation level

```
int adc1_get_raw(adc1_channel_t channel)
```

Take an ADC1 reading on a single channel.

Note Call adc1_config_width() before the first time this function is called.

Note For a given channel, adc1_config_channel_atten(channel) must be called before the first time this function is called.

Return

- -1: Parameter error
- Other: ADC1 channel reading.

Parameters

- channel: ADC1 channel to read

```
void adc1_ulp_enable()
```

Configure ADC1 to be usable by the ULP.

This function reconfigures ADC1 to be controlled by the ULP. Effect of this function can be reverted using adc1_get_raw function.

Note that adc1_config_channel_atten, adc1_config_width functions need to be called to configure ADC1 channels, before ADC1 is used by the ULP.

```
int hall_sensor_read()
```

Read Hall Sensor.

Note The Hall Sensor uses channels 0 and 3 of ADC1. Do not configure these channels for use as ADC channels.

Note The ADC1 module must be enabled by calling adc1_config_width() before calling hall_sensor_read(). ADC1 should be configured for 12 bit readings, as the hall sensor readings are low values and do not cover the full range of the ADC.

Return The hall sensor reading.

```
esp_err_t adc2_vref_to_gpio(gpio_num_t gpio)
```

Output ADC2 reference voltage to gpio 25 or 26 or 27.

This function utilizes the testing mux exclusive to ADC 2 to route the reference voltage one of ADC2's channels. Supported gpios are gpios 25, 26, and 27. This reference voltage can be manually read from the pin and used in the esp_adc_cal component.

Return

- ESP_OK: v_ref successfully routed to selected gpio

- **ESP_ERR_INVALID_ARG**: Unsupported gpio

Parameters

- **gpio**: GPIO number (gpios 25,26,27 supported)

Enumerations

enum adc_atten_t

Values:

ADC_ATTEN_0db = 0

The input voltage of ADC will be reduced to about 1/1

ADC_ATTEN_2_5db = 1

The input voltage of ADC will be reduced to about 1/1.34

ADC_ATTEN_6db = 2

The input voltage of ADC will be reduced to about 1/2

ADC_ATTEN_11db = 3

The input voltage of ADC will be reduced to about 1/3.6

enum adc_bits_width_t

Values:

ADC_WIDTH_9Bit = 0

ADC capture width is 9Bit

ADC_WIDTH_10Bit = 1

ADC capture width is 10Bit

ADC_WIDTH_11Bit = 2

ADC capture width is 11Bit

ADC_WIDTH_12Bit = 3

ADC capture width is 12Bit

enum adc1_channel1_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

GPIO Lookup Macros

Some useful macros can be used to specified 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).

Header File

- `soc/esp32/include/soc/adc_channel.h`

Macros

ADC1_GPIO36_CHANNEL

ADC1_CHANNEL_0_GPIO_NUM

ADC1_GPIO37_CHANNEL

ADC1_CHANNEL_1_GPIO_NUM
ADC1_GPIO38_CHANNEL
ADC1_CHANNEL_2_GPIO_NUM
ADC1_GPIO39_CHANNEL
ADC1_CHANNEL_3_GPIO_NUM
ADC1_GPIO32_CHANNEL
ADC1_CHANNEL_4_GPIO_NUM
ADC1_GPIO33_CHANNEL
ADC1_CHANNEL_5_GPIO_NUM
ADC1_GPIO34_CHANNEL
ADC1_CHANNEL_6_GPIO_NUM
ADC1_GPIO35_CHANNEL
ADC1_CHANNEL_7_GPIO_NUM
ADC2_GPIO4_CHANNEL
ADC2_CHANNEL_0_GPIO_NUM
ADC2_GPIO0_CHANNEL
ADC2_CHANNEL_1_GPIO_NUM
ADC2_GPIO2_CHANNEL
ADC2_CHANNEL_2_GPIO_NUM
ADC2_GPIO15_CHANNEL
ADC2_CHANNEL_3_GPIO_NUM
ADC2_GPIO13_CHANNEL
ADC2_CHANNEL_4_GPIO_NUM
ADC2_GPIO12_CHANNEL
ADC2_CHANNEL_5_GPIO_NUM
ADC2_GPIO14_CHANNEL
ADC2_CHANNEL_6_GPIO_NUM
ADC2_GPIO27_CHANNEL
ADC2_CHANNEL_7_GPIO_NUM
ADC2_GPIO25_CHANNEL
ADC2_CHANNEL_8_GPIO_NUM
ADC2_GPIO26_CHANNEL
ADC2_CHANNEL_9_GPIO_NUM

2.4.2 ADC Calibration

Overview

The esp_adc_cal API provides functions to correct for differences in measured voltages caused by non-ideal ADC reference voltages in ESP32s. The ideal ADC reference voltage is 1100mV however the reference voltage of different ESP32s can range from 1000mV to 1200mV.

Correcting the measured voltage using the esp_adc_cal API involves referencing a lookup table of voltages. The voltage obtained from the lookup table is the scaled and shifted by a gain and offset factor that is based on the ADC's reference voltage.

The reference voltage of the ADCs can be routed to certain GPIOs and measured manually using the ADC driver's adc2_vref_to_gpio() function.

Application Example

Reading the ADC and obtaining a result in mV:

```
#include <driver/adc.h>
#include <esp_adc_cal.h>

...
#define V_REF 1100 //ADC reference voltage

//Config ADC and characteristics
adc1_config_width(ADC_WIDTH_12Bit);
adc1_config_channel_atten(ADC1_CHANNEL_6, ADC_ATTEN_11db);

//Calculate ADC characteristics i.e. gain and offset factors
esp_adc_cal_characteristics_t characteristics;
esp_adc_cal_get_characteristics(V_REF, ADC_ATTEN_11db, ADC_WIDTH_12Bit, &
characteristics);

//Read ADC and obtain result in mV
uint32_t voltage = adc1_to_voltage(ADC1_CHANNEL_6, &characteristics);
printf("%d mV\n", voltage);
```

Routing ADC reference voltage to GPIO:

```
#include <driver/adc.h>
#include <driver/gpio.h>
#include <esp_err.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");
}
```

API Reference

Header File

- [esp_adc_cal/include/esp_adc_cal.h](#)

Functions

```
void esp_adc_cal_get_characteristics(uint32_t v_ref, adc_atten_t atten, adc_bits_width_t bit_width, esp_adc_cal_characteristics_t *chars)
```

Calculate characteristics of ADC.

This function will calculate the gain and offset factors based on the reference voltage parameter and the Gain and Offset curve provided in the LUT.

Note reference voltage of the ADCs can be routed to GPIO using adc2_vref_to_gpio() from the ADC driver

Note The LUT members have been bit shifted by ADC_CAL_GAIN_SCALE or ADC_CAL_OFFSET_SCALE to make them uint32_t compatible. This bit shifting will accounted for in this function

Parameters

- v_ref: true reference voltage of the ADC in mV (1000 to 1200mV). Nominal value for reference voltage is 1100mV.
- atten: attenuation setting used to select the corresponding lookup table
- bit_width: bit width of ADC
- chars: pointer to structure used to store ADC characteristics of module

```
uint32_t esp_adc_cal_raw_to_voltage(uint32_t adc, const esp_adc_cal_characteristics_t *chars)
```

Convert raw ADC reading to voltage in mV.

This function converts a raw ADC reading to a voltage in mV. This conversion is based on the ADC's characteristics. The raw ADC reading is referenced against the LUT (pointed to inside characteristics struct) to obtain a voltage. Gain and offset factors are then applied to the voltage in order to obtain the final result.

Return Calculated voltage in mV

Note characteristics structure must be initialized using esp_adc_cal_get_characteristics() before this function is used

Parameters

- adc: ADC reading (different bit widths will be handled)
- chars: pointer to structure containing ADC characteristics of the module. Structure also contains pointer to the corresponding LUT

```
uint32_t adc1_to_voltage(adc1_channel_t channel, const esp_adc_cal_characteristics_t *chars)
```

Reads ADC1 and returns voltage in mV.

This function reads the ADC1 using adc1_get_raw() to obtain a raw ADC reading. The reading is then converted into a voltage value using esp_adc_cal_raw_to_voltage().

Return voltage Calculated voltage in mV

Note ADC must be initialized using adc1_config_width() and adc1_config_channel_atten() before this function is used

Note characteristics structure must be initialized using esp_adc_cal_get_characteristics() before this function is used

Parameters

- channel: Channel of ADC1 to measure
- chars: Pointer to ADC characteristics struct

Structures

struct esp_adc_cal_lookup_table_t

Structure storing Lookup Table.

The Lookup Tables (LUT) of a given attenuation contains 33 equally spaced points. The Gain and Offset curves are used to find the appropriate gain and offset factor given a reference voltage v_ref.

Note A separate LUT is provided for each attenuation and are defined in esp_adc_cal_lookup_tables.c

Public Members

uint32_t **gain_m**

Gradient of Gain Curve

uint32_t **gain_c**

Offset of Gain Curve

uint32_t **offset_m**

Gradient of Offset Curve

uint32_t **offset_c**

Offset of Offset Curve

uint32_t **bit_shift**

Bit shift used to find corresponding LUT points given an ADC reading

uint32_t **voltage[]**

Array of voltages in mV representing the ADC-Voltage curve

struct esp_adc_cal_characteristics_t

Structure storing ADC characteristics of given v_ref.

The ADC Characteristics structure stores the gain and offset factors of an ESP32 module's ADC. These factors are calculated using the reference voltage, and the Gain and Offset curves provided in the lookup tables.

Note Call esp_adc_cal_get_characteristics() to initialize the structure

Public Members

uint32_t **v_ref**

Reference Voltage of current ESP32 Module in mV

uint32_t **gain**

Scaling factor used to correct LUT voltages to current v_ref. Bit shifted by << ADC_CAL_GAIN_SCALE for uint32 arithmetic

```
uint32_t offset
    Offset in mV used to correct LUT Voltages to current v_ref

uint32_t ideal_offset
    Offset in mV at the ideal reference voltage

adc_bits_width_t bit_width
    Bit width of ADC e.g. ADC_WIDTH_12Bit

const esp_adc_cal_lookup_table_t *table
    Pointer to LUT
```

2.4.3 Digital To Analog Converter

Overview

ESP32 has two 8-bit DAC (digital to analog converter) channels, connected to GPIO25 (Channel 1) and GPIO26 (Channel 2).

The DAC driver allows these channels to be set to arbitrary voltages.

The DAC channels can also be driven with DMA-style written sample data, via the [I2S driver](#) when using the “built-in DAC mode”.

For other analog output options, see the [Sigma-delta Modulation module](#) and the [LED Control module](#). Both these modules produce high frequency PWM output, which can be hardware low-pass filtered in order to generate a lower frequency analog output.

Application Example

Setting DAC channel 1 (GPIO 25) voltage to approx 0.78 of VDD_A voltage (VDD * 200 / 255). For VDD_A 3.3V, this is 2.59V:

```
#include <driver/dac.h>

...
dac_output_enable(DAC_CHANNEL_1);
dac_output_voltage(DAC_CHANNEL_1, 200);
```

API Reference

Header File

- [driver/include/driver/dac.h](#)

Functions

```
esp_err_t dac_output_voltage (dac_channel_t channel, uint8_t dac_value)
    Set DAC output voltage.
```

DAC output is 8-bit. Maximum (255) corresponds to VDD.

Note Need to configure DAC pad before calling this function. DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- channel: DAC channel
- dac_value: DAC output value

`esp_err_t dac_output_enable (dac_channel_t channel)`

DAC pad output enable.

Note DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26 I2S left channel will be mapped to DAC channel 2 I2S right channel will be mapped to DAC channel 1

Parameters

- channel: DAC channel

`esp_err_t dac_output_disable (dac_channel_t channel)`

DAC pad output disable.

Note DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26

Parameters

- channel: DAC channel

`esp_err_t dac_i2s_enable ()`

Enable DAC output data from I2S.

`esp_err_t dac_i2s_disable ()`

Disable DAC output data from I2S.

Enumerations

`enum dac_channel_t`

Values:

`DAC_CHANNEL_1 = 1`
DAC channel 1 is GPIO25

`DAC_CHANNEL_2`
DAC channel 2 is GPIO26

`DAC_CHANNEL_MAX`

GPIO Lookup Macros

Some useful macros can be used to specified the GPIO number of a DAC channel, or vice versa. e.g.

1. `DAC_CHANNEL_1_GPIO_NUM` is the GPIO number of channel 1 (25);
2. `DAC_GPIO26_CHANNEL` is the channel number of GPIO 26 (channel 2).

Header File

- [soc/esp32/include/soc/dac_channel.h](#)

Macros

DAC_GPIO25_CHANNEL

DAC_CHANNEL_1_GPIO_NUM

DAC_GPIO26_CHANNEL

DAC_CHANNEL_2_GPIO_NUM

2.4.4 GPIO & RTC GPIO

Overview

The ESP32 chip features 40 physical GPIO pads. Some GPIO pads cannot be used or do not have the corresponding pin on the chip package(refer to technical reference manual). Each pad can be used as a general purpose I/O or can be connected to an internal peripheral signal.

- Note that GPIO6-11 are usually used for SPI flash.
- GPIO34-39 can only be set as input mode and do not have software pullup or pulldown functions.

There is also separate “RTC GPIO” support, which functions when GPIOs are routed to the “RTC” low-power and analog subsystem. These pin functions can be used when in deep sleep, when the *Ultra Low Power co-processor* is running, or when analog functions such as ADC/DAC/etc are in use.

Application Example

GPIO output and input interrupt example: [peripherals/gpio](#).

API Reference - Normal GPIO

Header File

- [driver/include/driver/gpio.h](#)

Functions

esp_err_t gpio_config([const gpio_config_t](#) **pGPIOConfig*)
GPIO common configuration.

Configure GPIO's Mode,pull-up,PullDown,IntrType

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `pGPIOConfig`: Pointer to GPIO configure struct

`esp_err_t gpio_set_intr_type (gpio_num_t gpio_num, gpio_int_type_t intr_type)`
GPIO set interrupt trigger type.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to set the trigger type of e.g. of GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `intr_type`: Interrupt type, select from `gpio_int_type_t`

`esp_err_t gpio_intr_enable (gpio_num_t gpio_num)`
Enable GPIO module interrupt signal.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to enable an interrupt on e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);

`esp_err_t gpio_intr_disable (gpio_num_t gpio_num)`
Disable GPIO module interrupt signal.

Return

- `ESP_OK` success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to disable the interrupt of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);

`esp_err_t gpio_set_level (gpio_num_t gpio_num, uint32_t level)`
GPIO set output level.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` GPIO number error

Parameters

- `gpio_num`: GPIO number. If you want to set the output level of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `level`: Output level. 0: low ; 1: high

```
int gpio_get_level (gpio_num_t gpio_num)
GPIO get input level.
```

Return

- 0 the GPIO input level is 0
- 1 the GPIO input level is 1

Parameters

- `gpio_num`: GPIO number. If you want to get the logic level of e.g. pin GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);

```
esp_err_t gpio_set_direction (gpio_num_t gpio_num, gpio_mode_t mode)
```

GPIO set direction.

Configure GPIO direction,such as output_only,input_only,output_and_input

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` GPIO error

Parameters

- `gpio_num`: Configure GPIO pins number, it should be GPIO number. If you want to set direction of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `mode`: GPIO direction

```
esp_err_t gpio_set_pull_mode (gpio_num_t gpio_num, gpio_pull_mode_t pull)
```

Configure GPIO pull-up/pull-down resistors.

Only pins that support both input & output have integrated pull-up and pull-down resistors. Input-only GPIOs 34-39 do not.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` : Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to set pull up or down mode for e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `pull`: GPIO pull up/down mode.

```
esp_err_t gpio_wakeup_enable (gpio_num_t gpio_num, gpio_int_type_t intr_type)
```

Enable GPIO wake-up function.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `gpio_num`: GPIO number.

- `intr_type`: GPIO wake-up type. Only `GPIO_INTR_LOW_LEVEL` or `GPIO_INTR_HIGH_LEVEL` can be used.

`esp_err_t gpio_wakeup_disable (gpio_num_t gpio_num)`
Disable GPIO wake-up function.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `gpio_num`: GPIO number

`esp_err_t gpio_isr_register (void (*fn)) void *`,
`void *arg, int intr_alloc_flags, gpio_isr_handle_t *handle` Register GPIO interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

This ISR function is called whenever any GPIO interrupt occurs. See the alternative `gpio_install_isr_service()` and `gpio_isr_handler_add()` API in order to have the driver support per-GPIO ISRs.

To disable or remove the ISR, pass the returned handle to the *interrupt allocation functions*.

Parameters

- `fn`: Interrupt handler function.
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `arg`: Parameter for handler function
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

Return

- `ESP_OK` Success ;
- `ESP_ERR_INVALID_ARG` GPIO error

`esp_err_t gpio_pullup_en (gpio_num_t gpio_num)`
Enable pull-up on GPIO.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `gpio_num`: GPIO number

`esp_err_t gpio_pullup_dis (gpio_num_t gpio_num)`
Disable pull-up on GPIO.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `gpio_num`: GPIO number

`esp_err_t 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_FAIL` Operation fail
- `ESP_ERR_NO_MEM` No memory to install this service

Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.

`void gpio_uninstall_isr_service()`

Uninstall the driver's GPIO ISR service, freeing related resources.

`esp_err_t gpio_isr_handler_add(gpio_num_t gpio_num, gpio_isr_t isr_handler, void *args)`

Add ISR handler for the corresponding GPIO pin.

Call this function after using `gpio_install_isr_service()` to install the driver's GPIO ISR handler service.

The pin ISR handlers no longer need to be declared with `IRAM_ATTR`, unless you pass the `ESP_INTR_FLAG_IRAM` flag when allocating the ISR in `gpio_install_isr_service()`.

This ISR handler will be called from an ISR. So there is a stack size limit (configurable as “ISR stack size” in menuconfig). This limit is smaller compared to a global GPIO interrupt handler due to the additional level of indirection.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- gpio_num: GPIO number
- isr_handler: ISR handler function for the corresponding GPIO number.
- args: parameter for ISR handler.

`esp_err_t gpio_isr_handler_remove (gpio_num_t gpio_num)`

Remove ISR handler for the corresponding GPIO pin.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- gpio_num: GPIO number

`esp_err_t gpio_set_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t strength)`

Set GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- gpio_num: GPIO number, only support output GPIOs
- strength: Drive capability of the pad

`esp_err_t gpio_get_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t *strength)`

Get GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- gpio_num: GPIO number, only support output GPIOs
- strength: Pointer to accept drive capability of the pad

Structures

`struct gpio_config_t`

Configuration parameters of GPIO pad for gpio_config function.

Public Members

`uint64_t pin_bit_mask`

GPIO pin: set with bit mask, each bit maps to a GPIO

`gpio_mode_t mode`

GPIO mode: set input/output mode

`gpio_pullup_t pull_up_en`

GPIO pull-up

`gpio_pulldown_t pull_down_en`

GPIO pull-down

`gpio_int_type_t intr_type`

GPIO interrupt type

Macros

`GPIO_SEL_0`

Pin 0 selected

`GPIO_SEL_1`

Pin 1 selected

`GPIO_SEL_2`

Pin 2 selected

Note There are more macros like that up to pin 39, excluding pins 20, 24 and 28..31. They are not shown here to reduce redundant information.

`GPIO_IS_VALID_GPIO(gpio_num)`

Check whether it is a valid GPIO number

`GPIO_IS_VALID_OUTPUT_GPIO(gpio_num)`

Check whether it can be a valid GPIO number of output mode

Type Definitions

`typedef void (*gpio_isr_t)(void *)`

`typedef intr_handle_t gpio_isr_handle_t`

Enumerations

`enum gpio_num_t`

Values:

`GPIO_NUM_0 = 0`

GPIO0, input and output

```
GPIO_NUM_1 = 1
    GPIO1, input and output
```

```
GPIO_NUM_2 = 2
    GPIO2, input and output
```

Note There are more enumerations like that up to GPIO39, excluding GPIO20, GPIO24 and GPIO28..31.
They are not shown here to reduce redundant information.

Note GPIO34..39 are input mode only.

```
enum gpio_int_type_t
```

Values:

```
GPIO_INTR_DISABLE = 0
    Disable GPIO interrupt
```

```
GPIO_INTR_POSEdge = 1
    GPIO interrupt type : rising edge
```

```
GPIO_INTR_NEGEDGE = 2
    GPIO interrupt type : falling edge
```

```
GPIO_INTR_ANYEDGE = 3
    GPIO interrupt type : both rising and falling edge
```

```
GPIO_INTR_LOW_LEVEL = 4
    GPIO interrupt type : input low level trigger
```

```
GPIO_INTR_HIGH_LEVEL = 5
    GPIO interrupt type : input high level trigger
```

```
GPIO_INTR_MAX
```

```
enum gpio_mode_t
```

Values:

```
GPIO_MODE_INPUT = GPIO_MODE_DEF_INPUT
    GPIO mode : input only
```

```
GPIO_MODE_OUTPUT = GPIO_MODE_DEF_OUTPUT
    GPIO mode : output only mode
```

```
GPIO_MODE_OUTPUT_OD = ((GPIO_MODE_DEF_OUTPUT)|(GPIO_MODE_DEF_OD))
    GPIO mode : output only with open-drain mode
```

```
GPIO_MODE_INPUT_OUTPUT_OD = ((GPIO_MODE_DEF_INPUT)|(GPIO_MODE_DEF_OUTPUT)|(GPIO_MODE_DEF_OD))
    GPIO mode : output and input with open-drain mode
```

```
GPIO_MODE_INPUT_OUTPUT = ((GPIO_MODE_DEF_INPUT)|(GPIO_MODE_DEF_OUTPUT))
    GPIO mode : output and input mode
```

```
enum gpio_pullup_t
```

Values:

```
GPIO_PULLUP_DISABLE = 0x0
    Disable GPIO pull-up resistor
```

```
GPIO_PULLUP_ENABLE = 0x1
    Enable GPIO pull-up resistor
```

```
enum gpio_pulldown_t
```

Values:

```
GPIO_PULLDOWN_DISABLE = 0x0
    Disable GPIO pull-down resistor
```

```
GPIO_PULLDOWN_ENABLE = 0x1
    Enable GPIO pull-down resistor
```

```
enum gpio_pull_mode_t
```

Values:

```
GPIO_PULLUP_ONLY
    Pad pull up
```

```
GPIO_PULLDOWN_ONLY
    Pad pull down
```

```
GPIO_PULLUP_PULLDOWN
    Pad pull up + pull down
```

```
GPIO_FLOATING
    Pad floating
```

```
enum gpio_drive_cap_t
```

Values:

```
GPIO_DRIVE_CAP_0 = 0
    Pad drive capability: weak
```

```
GPIO_DRIVE_CAP_1 = 1
    Pad drive capability: stronger
```

```
GPIO_DRIVE_CAP_2 = 2
    Pad drive capability: default value
```

```
GPIO_DRIVE_CAP_DEFAULT = 2
    Pad drive capability: default value
```

```
GPIO_DRIVE_CAP_3 = 3
    Pad drive capability: strongest
```

```
GPIO_DRIVE_CAP_MAX
```

API Reference - RTC GPIO

Header File

- [driver/include/driver/rtc_io.h](#)

Functions

```
static bool rtc_gpio_is_valid_gpio(gpio_num_t gpio_num)
```

Determine if the specified GPIO is a valid RTC GPIO.

Return true if GPIO is valid for RTC GPIO use. false otherwise.

Parameters

- `gpio_num`: GPIO number

```
esp_err_t rtc_gpio_init (gpio_num_t gpio_num)
```

Init a GPIO as RTC GPIO.

This function must be called when initializing a pad for an analog function.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- *gpio_num*: GPIO number (e.g. GPIO_NUM_12)

```
esp_err_t rtc_gpio_deinit (gpio_num_t gpio_num)
```

Init a GPIO as digital GPIO.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- *gpio_num*: GPIO number (e.g. GPIO_NUM_12)

```
uint32_t rtc_gpio_get_level (gpio_num_t gpio_num)
```

Get the RTC IO input level.

Return

- 1 High level
- 0 Low level
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- *gpio_num*: GPIO number (e.g. GPIO_NUM_12)

```
esp_err_t rtc_gpio_set_level (gpio_num_t gpio_num, uint32_t level)
```

Set the RTC IO output level.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- *gpio_num*: GPIO number (e.g. GPIO_NUM_12)
- *level*: output level

```
esp_err_t rtc_gpio_set_direction (gpio_num_t gpio_num, rtc_gpio_mode_t mode)
```

RTC GPIO set direction.

Configure RTC GPIO direction, such as output only, input only, output and input.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)
- mode: GPIO direction

`esp_err_t rtc_gpio_pullup_en (gpio_num_t gpio_num)`
RTC GPIO pullup enable.

This function only works for RTC IOs. In general, call `gpio_pullup_en`, which will work both for normal GPIOs and RTC IOs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

`esp_err_t rtc_gpio_pulldown_en (gpio_num_t gpio_num)`
RTC GPIO pulldown enable.

This function only works for RTC IOs. In general, call `gpio_pulldown_en`, which will work both for normal GPIOs and RTC IOs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

`esp_err_t rtc_gpio_pullup_dis (gpio_num_t gpio_num)`
RTC GPIO pullup disable.

This function only works for RTC IOs. In general, call `gpio_pullup_dis`, which will work both for normal GPIOs and RTC IOs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

`esp_err_t rtc_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)

`void rtc_gpio_force_hold_dis_all ()`

Disable force hold signal for all RTC IOs.

Each RTC pad has a “force hold” input signal from the RTC controller. If this signal is set, pad latches current values of input enable, function, output enable, and other signals which come from the RTC mux. Force hold signal is enabled before going into deep sleep for pins which are used for EXT1 wakeup.

`esp_err_t rtc_gpio_set_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t strength)`

Set RTC GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- gpio_num: GPIO number, only support output GPIOs
- strength: Drive capability of the pad

esp_err_t **rtc_gpio_get_drive_capability**(*gpio_num_t gpio_num, gpio_drive_cap_t *strength*)
Get RTC GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number, only support output GPIOs
- *strength*: Pointer to accept drive capability of the pad

Structures

struct rtc_gpio_desc_t

Pin function information for a single GPIO pad's RTC functions.

This is an internal function of the driver, and is not usually useful for external use.

Public Members

uint32_t reg

Register of RTC pad, or 0 if not an RTC GPIO

uint32_t mux

Bit mask for selecting digital pad or RTC pad

uint32_t func

Shift of pad function (FUN_SEL) field

uint32_t ie

Mask of input enable

uint32_t pullup

Mask of pullup enable

uint32_t pulldown

Mask of pulldown enable

uint32_t slpsel

If slpsel bit is set, slpie will be used as pad input enabled signal in sleep mode

uint32_t slpie

Mask of input enable in sleep mode

uint32_t hold

Mask of hold enable

uint32_t hold_force

Mask of hold_force bit for RTC IO in RTC_CNTL_HOLD_FORCE_REG

uint32_t drv_v

Mask of drive capability

uint32_t drv_s

Offset of drive capability

```
int rtc_num
    RTC IO number, or -1 if not an RTC GPIO
```

Macros

`RTC_GPIO_IS_VALID_GPIO(gpio_num)`

Enumerations

```
enum rtc_gpio_mode_t
    Values:
        RTC_GPIO_MODE_INPUT_ONLY
            Pad output
        RTC_GPIO_MODE_OUTPUT_ONLY
            Pad input
        RTC_GPIO_MODE_INPUT_OUTPUT
            Pad pull output + input
        RTC_GPIO_MODE_DISABLED
            Pad (output + input) disable
```

2.4.5 I2C

Overview

ESP32 has two I2C controllers which can be set as master mode or slave mode.

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.

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, gpio_num_t sda_io_num, gpio_num_t 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

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- cmd_handle: I2C cmd link
- data: data to send
- 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*, *int 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

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- cmd_handle: I2C cmd link
- data: pointer accept the data byte
- 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*, *int 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

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
- 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, port-
```

```
BASE_TYPE 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, portBASE_TYPE
```

```
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, portBASE_TYPE
```

```
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_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_data_mode(i2c_port_t i2c_num, i2c_trans_mode_t tx_trans_mode, i2c_trans_mode_t
                            rx_trans_mode)
    set I2C data transfer mode
```

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2c_num`: I2C port number
- `tx_trans_mode`: I2C sending data mode
- `rx_trans_mode`: I2C receiving data mode

```
esp_err_t i2c_get_data_mode(i2c_port_t      i2c_num,      i2c_trans_mode_t      *tx_trans_mode,
                            i2c_trans_mode_t *rx_trans_mode)
    get I2C data transfer mode
```

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2c_num`: I2C port number
- `tx_trans_mode`: pointer to get I2C sending data mode
- `rx_trans_mode`: pointer to get I2C receiving data mode

Structures

```
struct i2c_config_t
    I2C initialization parameters.
```

Public Members

`i2c_mode_t mode`
I2C mode

`gpio_num_t sda_io_num`
GPIO number for I2C sda signal

`gpio_pullup_t sda_pullup_en`
Internal GPIO pull mode for I2C sda signal

`gpio_num_t scl_io_num`
GPIO number for I2C scl signal

`gpio_pullup_t scl_pullup_en`
Internal GPIO pull mode for I2C scl signal

`uint32_t clk_speed`
I2C clock frequency for master mode, (no higher than 1MHz for now)

`uint8_t addr_10bit_en`
I2C 10bit address mode enable for slave mode

`uint16_t slave_addr`
I2C address for slave mode

Macros

`I2C_APB_CLK_FREQ`
I2C source clock is APB clock, 80MHz

`I2C_FIFO_LEN`
I2C hardware fifo length

Type Definitions

`typedef void *i2c_cmd_handle_t`
I2C command handle

Enumerations

`enum i2c_mode_t`
Values:

`I2C_MODE_SLAVE` = 0
I2C slave mode

`I2C_MODE_MASTER`
I2C master mode

`I2C_MODE_MAX`

`enum i2c_rw_t`
Values:

`I2C_MASTER_WRITE` = 0
I2C write data

```

I2C_MASTER_READ
    I2C read data

enum i2c_trans_mode_t
    Values:

        I2C_DATA_MODE_MSB_FIRST = 0
            I2C data msb first

        I2C_DATA_MODE_LSB_FIRST = 1
            I2C data lsb first

I2C_DATA_MODE_MAX

enum i2c_opmode_t
    Values:

        I2C_CMD_RESTART = 0
            I2C restart command

        I2C_CMD_WRITE
            I2C write command

        I2C_CMD_READ
            I2C read command

        I2C_CMD_STOP
            I2C stop command

        I2C_CMD_END
            I2C end command

enum i2c_port_t
    Values:

        I2C_NUM_0 = 0
            I2C port 0

        I2C_NUM_1
            I2C port 1

I2C_NUM_MAX

enum i2c_addr_mode_t
    Values:

        I2C_ADDR_BIT_7 = 0
            I2C 7bit address for slave mode

        I2C_ADDR_BIT_10
            I2C 10bit address for slave mode

        I2C_ADDR_BIT_MAX

```

2.4.6 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.

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 = ESP_INTR_FLAG_LEVEL1, // high interrupt priority
    .dma_buf_count = 8,
    .dma_buf_len = 64
};

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,
.intr_alloc_flags = ESP_INTR_FLAG_LEVEL1, // high interrupt priority
.dma_buf_count = 8,
.dma_buf_len = 64
};

...
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_FAIL` Parameter 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_FAIL Parameter error

```
esp_err_t i2s_driver_uninstall (i2s_port_t i2s_num)
```

Uninstall I2S driver.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `i2s_num`: I2S_NUM_0, I2S_NUM_1

```
int i2s_write_bytes (i2s_port_t i2s_num, const char *src, size_t size, TickType_t ticks_to_wait)
```

Write data to I2S DMA transmit buffer.

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
- `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 Number of bytes written, or ESP_FAIL (-1) for parameter error. If a timeout occurred, bytes written will be less than total size.

```
int i2s_read_bytes (i2s_port_t i2s_num, char *dest, size_t size, TickType_t ticks_to_wait)
Read data from I2S DMA receive buffer.
```

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
- dest: Destination address to read into
- size: Size of data in bytes
- 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.

Return Number of bytes read, or ESP_FAIL (-1) for parameter error. If a timeout occurred, bytes read will be less than total size.

```
int i2s_push_sample (i2s_port_t i2s_num, const char *sample, TickType_t ticks_to_wait)
Push (write) a single sample to the I2S DMA TX buffer.
```

Size of the sample is determined by the channel_format (mono or stereo)) & bits_per_sample configuration (see [i2s_config_t](#)).

Return Number of bytes successfully pushed to DMA buffer, or ESP_FAIL (-1) for parameter error. Will be either zero or the size of configured sample buffer.

Parameters

- i2s_num: I2S_NUM_0, I2S_NUM_1
- sample: Pointer to buffer containing sample to write. Size of buffer (in bytes) = (number of channels) * bits_per_sample / 8.
- ticks_to_wait: Push timeout in RTOS ticks. If space is not available in the DMA TX buffer within this period, no data is written and function returns 0.

```
int i2s_pop_sample (i2s_port_t i2s_num, char *sample, TickType_t ticks_to_wait)
Pop (read) a single sample from the I2S DMA RX buffer.
```

Size of the sample is determined by the channel_format (mono or stereo)) & bits_per_sample configuration (see [i2s_config_t](#)).

Return Number of bytes successfully read from DMA buffer, or ESP_FAIL (-1) for parameter error. Byte count will be either zero or the size of the configured sample buffer.

Parameters

- i2s_num: I2S_NUM_0, I2S_NUM_1
- sample: Buffer sample data will be read into. Size of buffer (in bytes) = (number of channels) * bits_per_sample / 8.
- ticks_to_wait: Pop 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_FAIL Parameter error

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_FAIL 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_FAIL 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_FAIL 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_FAIL` Parameter error

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)

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

struct i2s_event_t

Event structure used in I2S event queue.

Public Members

```
i2s_event_type_t type  
    I2S event type  
  
size_t size  
    I2S data size for I2S_DATA event  
  
struct i2s_pin_config_t  
    I2S pin number for i2s_set_pin.
```

Public Members

```
int bck_io_num  
    BCK in out pin  
  
int ws_io_num  
    WS in out pin  
  
int data_out_num  
    DATA out pin  
  
int data_in_num  
    DATA in pin
```

Macros

I2S_PIN_NO_CHANGE
Use in *i2s_pin_config_t* for pins which should not be changed

Type Definitions

```
typedef intr_handle_t i2s_isr_handle_t
```

Enumerations

```
enum i2s_bits_per_sample_t  
    I2S bit width per sample.  
  
    Values:  
  
    I2S_BITS_PER_SAMPLE_8BIT = 8  
        I2S bits per sample: 8-bits  
  
    I2S_BITS_PER_SAMPLE_16BIT = 16  
        I2S bits per sample: 16-bits  
  
    I2S_BITS_PER_SAMPLE_24BIT = 24  
        I2S bits per sample: 24-bits  
  
    I2S_BITS_PER_SAMPLE_32BIT = 32  
        I2S bits per sample: 32-bits
```

enum i2s_channel_t

I2S channel.

*Values:***I2S_CHANNEL_MONO = 1**

I2S 1 channel (mono)

I2S_CHANNEL_STEREO = 2

I2S 2 channel (stereo)

enum i2s_comm_format_t

I2S communication standard format.

*Values:***I2S_COMM_FORMAT_I2S = 0x01**

I2S communication format I2S

I2S_COMM_FORMAT_I2S_MSB = 0x02

I2S format MSB

I2S_COMM_FORMAT_I2S_LSB = 0x04

I2S format LSB

I2S_COMM_FORMAT_PCM = 0x08

I2S communication format PCM

I2S_COMM_FORMAT_PCM_SHORT = 0x10

PCM Short

I2S_COMM_FORMAT_PCM_LONG = 0x20

PCM Long

enum i2s_channel_fmt_t

I2S channel format type.

*Values:***I2S_CHANNEL_FMT_RIGHT_LEFT = 0x00****I2S_CHANNEL_FMT_ALL_RIGHT****I2S_CHANNEL_FMT_ALL_LEFT****I2S_CHANNEL_FMT_ONLY_RIGHT****I2S_CHANNEL_FMT_ONLY_LEFT****enum pdm_sample_rate_ratio_t**

PDM sample rate ratio, measured in Hz.

*Values:***PDM_SAMPLE_RATE_RATIO_64****PDM_SAMPLE_RATE_RATIO_128****enum pdm_pcm_conv_t**

PDM PCM convter enable/disable.

*Values:***PDM_PCM_CONV_ENABLE****PDM_PCM_CONV_DISABLE**

enum i2s_port_t

I2S Peripheral, 0 & 1.

Values:

I2S_NUM_0 = 0x0

I2S 0

I2S_NUM_1 = 0x1

I2S 1

I2S_NUM_MAX

enum i2s_mode_t

I2S Mode, default is I2S_MODE_MASTER | I2S_MODE_TX.

Note PDM and built-in DAC functions are only supported on I2S0 for current ESP32 chip.

Values:

I2S_MODE_MASTER = 1

I2S_MODE_SLAVE = 2

I2S_MODE_TX = 4

I2S_MODE_RX = 8

I2S_MODE_DAC_BUILT_IN = 16

Output I2S data to built-in DAC, no matter the data format is 16bit or 32 bit, the DAC module will only take the 8bits from MSB

I2S_MODE_PDM = 64

enum i2s_event_type_t

I2S event types.

Values:

I2S_EVENT_DMA_ERROR

I2S_EVENT_TX_DONE

I2S DMA finish sent 1 buffer

I2S_EVENT_RX_DONE

I2S DMA finish received 1 buffer

I2S_EVENT_MAX

I2S event max index

enum i2s_dac_mode_t

I2S DAC mode for i2s_set_dac_mode.

Note PDM and built-in DAC functions are only supported on I2S0 for current ESP32 chip.

Values:

I2S_DAC_CHANNEL_DISABLE = 0

Disable I2S built-in DAC signals

I2S_DAC_CHANNEL_RIGHT_EN = 1

Enable I2S built-in DAC right channel, maps to DAC channel 1 on GPIO25

I2S_DAC_CHANNEL_LEFT_EN = 2
 Enable I2S built-in DAC left channel, maps to DAC channel 2 on GPIO26

I2S_DAC_CHANNEL_BOTH_EN = 0x3
 Enable both of the I2S built-in DAC channels.

I2S_DAC_CHANNEL_MAX = 0x4
 I2S built-in DAC mode max index

2.4.7 LED Control

Overview

The LED control 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. For maximum flexibility, the high-speed as well as the low-speed channels can be driven from one of four high-speed/low-speed timers. The PWM controller also has the ability to automatically increase or decrease the duty cycle gradually, allowing for fades without any processor interference.

Application Example

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 depth.

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)/bit_num.

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 bit_num.

Parameters

- timer_conf: Pointer of LEDC timer configure struct

`esp_err_t ledc_update_duty(ledc_mode_t speed_mode, ledc_channel_t channel)`

LEDC update channel parameters Call this function to activate the LEDC updated parameters. After ledc_set_duty, ledc_set_fade, we need to call this function to update the settings.

Return

- **ESP_OK** Success
- **ESP_ERR_INVALID_ARG** Parameter error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- channel: LEDC channel(0-7), select from ledc_channel_t

`esp_err_t ledc_stop(ledc_mode_t speed_mode, ledc_channel_t channel, uint32_t idle_level)`

LEDC stop. Disable LEDC output, and set idle level.

Return

- **ESP_OK** Success
- **ESP_ERR_INVALID_ARG** Parameter error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode
- channel: LEDC channel(0-7), select from ledc_channel_t
- idle_level: Set output idle level after LEDC stops.

`esp_err_t ledc_set_freq(ledc_mode_t speed_mode, ledc_timer_t timer_num, uint32_t freq_hz)`

LEDC set channel frequency(Hz)

Return

- **ESP_OK** Success
- **ESP_ERR_INVALID_ARG** Parameter error
- **ESP_FAIL** Can not find a proper pre-divider number base on the given frequency and the current bit_num.

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode
- timer_num: LEDC timer index(0-3), select from ledc_timer_t
- freq_hz: Set the LEDC frequency

`uint32_t ledc_get_freq(ledc_mode_t speed_mode, ledc_timer_t timer_num)`

LEDC get channel frequency(Hz)

Return

- 0 error
- Others Current LEDC frequency

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode
- timer_num: LEDC timer index(0-3), select from ledc_timer_t

`esp_err_t ledc_set_duty (ledc_mode_t speed_mode, ledc_channel_t channel, uint32_t duty)`
LEDC set duty Only after calling ledc_update_duty will the duty update.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode
- channel: LEDC channel(0-7), select from ledc_channel_t
- duty: Set the LEDC duty, the duty range is [0, (2**bit_num) - 1]

`int ledc_get_duty (ledc_mode_t speed_mode, ledc_channel_t channel)`
LEDC get duty.

Return

- (-1) parameter error
- Others Current LEDC duty

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode
- channel: LEDC channel(0-7), select from ledc_channel_t

`esp_err_t ledc_set_fade (ledc_mode_t speed_mode, ledc_channel_t channel, uint32_t duty, ledc_duty_direction_t gradule_direction, uint32_t step_num, uint32_t duty_cyle_num, uint32_t duty_scale)`

LEDC set gradient Set LEDC gradient, After the function calls the ledc_update_duty function, the function can take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode
- channel: LEDC channel(0-7), select from ledc_channel_t
- duty: Set the start of the gradient duty, the duty range is [0, (2**bit_num) - 1]
- gradule_direction: Set the direction of the gradient
- step_num: Set the number of the gradient

- `duty_cycle_num`: Set how many LEDC tick each time the gradient lasts
- `duty_scale`: Set gradient change amplitude

`esp_err_t ledc_isr_register(void (*fn)) void *`
, void *`arg`, int `intr_alloc_flags`, `ledc_isr_handle_t` *`handle` Register LEDC interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Function pointer error.

Parameters

- `fn`: Interrupt handler function.
- `arg`: User-supplied argument passed to the handler function.
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `arg`: Parameter for handler function
- `handle`: Pointer to return handle. If non-NUL, 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 div_num, uint32_t bit_num, 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
- `div_num`: Timer clock divide number, the timer clock is divided from the selected clock source
- `bit_num`: The count number of one period, counter range is 0 ~ ((2 ** bit_num) - 1)
- `clk_src`: Select LEDC source clock.

`esp_err_t ledc_timer_rst(ledc_mode_t speed_mode, uint32_t timer_sel)`
Reset LEDC timer.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index(0-3), select from `ledc_timer_t`

`esp_err_t ledc_timer_pause (ledc_mode_t speed_mode, uint32_t timer_sel)`
Pause LEDC timer counter.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index(0-3), select from ledc_timer_t

`esp_err_t ledc_timer_resume (ledc_mode_t speed_mode, uint32_t timer_sel)`
Resume LEDC timer.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index(0-3), select from ledc_timer_t

`esp_err_t ledc_bind_channel_timer (ledc_mode_t speed_mode, uint32_t channel, uint32_t timer_idx)`
Bind LEDC channel with the selected timer.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode
- `channel`: LEDC channel index(0-7), select from ledc_channel_t
- `timer_idx`: LEDC timer index(0-3), select from ledc_timer_t

`esp_err_t ledc_set_fade_with_step (ledc_mode_t speed_mode, ledc_channel_t channel, int target_duty, int scale, int cycle_num)`

Set LEDC fade function. Should call ledc_fade_func_install() before calling this function. Call ledc_fade_start() after this to start fading.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- `speed_mode`: Select the LEDC speed_mode, high-speed mode and low-speed mode,

- channel: LEDC channel index(0-7), select from ledc_channel_t
- target_duty: Target duty of fading.(0 - (2 ** bit_num - 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, int tar-  
get_duty, int max_fade_time_ms)
```

Set LEDC fade function, with a limited time. Should call ledc_fade_func_install() before calling this function. Call ledc_fade_start() after this to start fading.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- channel: LEDC channel index(0-7), select from ledc_channel_t
- target_duty: Target duty of fading.(0 - (2 ** bit_num - 1))
- max_fade_time_ms: The maximum time of the fading (ms).

```
esp_err_t ledc_fade_func_install(int intr_alloc_flags)
```

Install ledc fade function. This function will occupy interrupt of LEDC module.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function already installed.

Parameters

- intr_alloc_flags: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.

```
void ledc_fade_func_uninstall()
```

Uninstall LEDC fade function.

```
esp_err_t ledc_fade_start(ledc_mode_t speed_mode, ledc_channel_t channel, ledc_fade_mode_t  
wait_done)
```

Start LEDC fading.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_ERR_INVALID_ARG Parameter error.

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode

- channel: LEDC channel number
- wait_done: Whether to block until fading done.

Structures

`struct ledc_channel_config_t`

Configuration parameters of LEDC channel for ledc_channel_config function.

Public Members

`int gpio_num`

the LEDC output gpio_num, if you want to use gpio16, gpio_num = 16

`ledc_mode_t speed_mode`

LEDC speed speed_mode, high-speed mode or low-speed mode

`ledc_channel_t channel`

LEDC channel(0 - 7)

`ledc_intr_type_t intr_type`

configure interrupt, Fade interrupt enable or Fade interrupt disable

`ledc_timer_t timer_sel`

Select the timer source of channel (0 - 3)

`uint32_t duty`

LEDC channel duty, the duty range is [0, (2**bit_num) - 1],

`struct ledc_timer_config_t`

Configuration parameters of LEDC Timer timer for ledc_timer_config function.

Public Members

`ledc_mode_t speed_mode`

LEDC speed speed_mode, high-speed mode or low-speed mode

`ledc_timer_bit_t bit_num`

LEDC channel duty depth

`ledc_timer_t timer_num`

The timer source of channel (0 - 3)

`uint32_t freq_hz`

LEDC timer frequency(Hz)

Macros

`LED_C_APB_CLK_HZ`

`LED_C_REF_CLK_HZ`

Type Definitions

`typedef intr_handle_t ledc_isr_handle_t`

Enumerations

`enum ledc_mode_t`

Values:

`LEDC_HIGH_SPEED_MODE` = 0
LEDC high speed speed_mode
`LEDC_LOW_SPEED_MODE`
LEDC low speed speed_mode
`LEDC_SPEED_MODE_MAX`
LEDC speed limit

`enum ledc_intr_type_t`

Values:

`LEDC_INTR_DISABLE` = 0
Disable LEDC interrupt
`LEDC_INTR_FADE_END`
Enable LEDC interrupt

`enum ledc_duty_direction_t`

Values:

`LEDC_DUTY_DIR_DECREASE` = 0
LEDC duty decrease direction
`LEDC_DUTY_DIR_INCREASE` = 1
LEDC duty increase direction

`enum ledc_clk_src_t`

Values:

`LEDC_REF_TICK` = 0
LEDC timer clock divided from reference tick(1Mhz)
`LEDC_APB_CLK`
LEDC timer clock divided from APB clock(80Mhz)

`enum ledc_timer_t`

Values:

`LEDC_TIMER_0` = 0
LEDC source timer TIMER0
`LEDC_TIMER_1`
LEDC source timer TIMER1
`LEDC_TIMER_2`
LEDC source timer TIMER2
`LEDC_TIMER_3`
LEDC source timer TIMER3

`enum ledc_channel_t`

Values:

`LEDC_CHANNEL_0` = 0
LEDC channel 0
`LEDC_CHANNEL_1`
LEDC channel 1

```

LEDC_CHANNEL_2
    LEDC channel 2

LEDC_CHANNEL_3
    LEDC channel 3

LEDC_CHANNEL_4
    LEDC channel 4

LEDC_CHANNEL_5
    LEDC channel 5

LEDC_CHANNEL_6
    LEDC channel 6

LEDC_CHANNEL_7
    LEDC channel 7

LEDC_CHANNEL_MAX

enum ledc_timer_bit_t
Values:

LEDC_TIMER_10_BIT = 10
    LEDC PWM depth 10Bit

LEDC_TIMER_11_BIT = 11
    LEDC PWM depth 11Bit

LEDC_TIMER_12_BIT = 12
    LEDC PWM depth 12Bit

LEDC_TIMER_13_BIT = 13
    LEDC PWM depth 13Bit

LEDC_TIMER_14_BIT = 14
    LEDC PWM depth 14Bit

LEDC_TIMER_15_BIT = 15
    LEDC PWM depth 15Bit

enum ledc_fade_mode_t
Values:

LEDC_FADE_NO_WAIT = 0
    LEDC fade function will return immediately

LEDC_FADE_WAIT_DONE
    LEDC fade function will block until fading to the target duty

LEDC_FADE_MAX

```

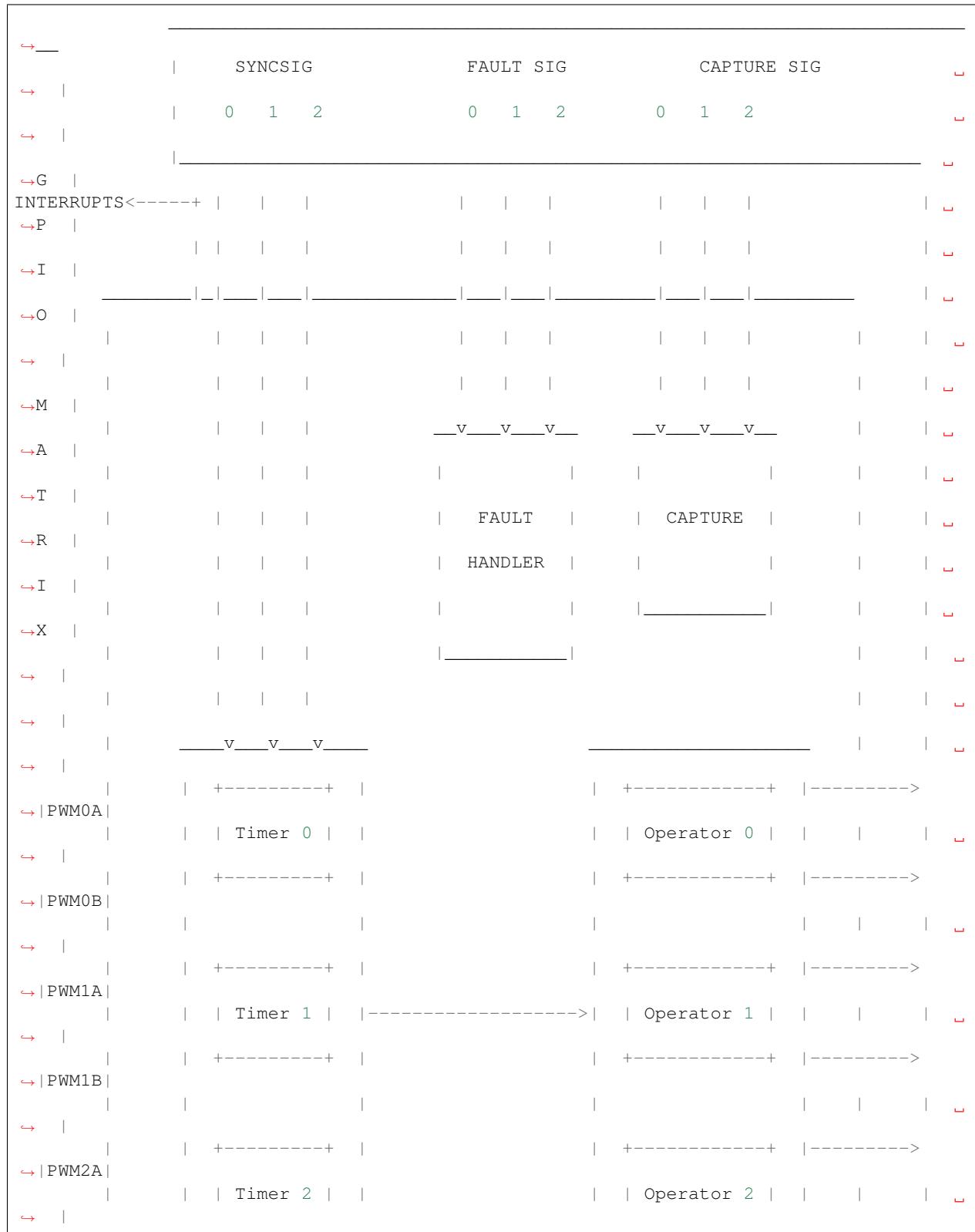
2.4.8 MCPWM

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.





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.

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 than 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 input_level, mcpwm_fault_signal_t fault_sig)`

Initialize fault submodule, currently low level triggering not supported.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- input_level: set fault signal level, which will cause fault to occur
- fault_sig: set the fault Pin, which needs to be enabled

```
esp_err_t mcpwm_fault_set_oneshot_mode (mcpwm_unit_t      mcpwm_num,      mcpwm_timer_t
                                         timer_num,      mcpwm_fault_signal_t    fault_sig,
                                         mcpwm_action_on_pwmxa_t   action_on_pwmxa,
                                         mcpwm_action_on_pwmxb_t   action_on_pwmxb)
```

Set oneshot mode on fault detection, once fault occur in oneshot mode reset is required to resume MCPWM signals.

Note currently low level triggering not supported

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- fault_sig: set the fault Pin, which needs to be enabled for oneshot mode
- action_on_pwmxa: action to be taken on MCPWMXA when fault occurs, either no change or high or low or toggle
- action_on_pwmxb: action to be taken on MCPWMXB when fault occurs, either no change or high or low or toggle

```
esp_err_t mcpwm_fault_set_cyc_mode (mcpwm_unit_t  mcpwm_num,  mcpwm_timer_t  timer_num,
                                     mcpwm_fault_signal_t fault_sig, mcpwm_action_on_pwmxa_t
                                     action_on_pwmxa,      mcpwm_action_on_pwmxb_t      ac-
                                     tion_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-NUL, 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 than 8(increment every 12.5%)

uint8_t pulse_width_in_os
    Set pulse width of first pulse in one shot mode = (carrier period)*(pulse_width_in_os + 1), should be less
    then 16

mcpwm_carrier_os_t carrier_os_mode
    Enable or disable carrier oneshot mode

mcpwm_carrier_out_ivt_t carrier_ivt_mode
    Invert output of carrier
```

Enumerations

```
enum mcpwm_io_signals_t
    IO signals for MCPWM 6 MCPWM output pins that generate PWM signals 3 MCPWM fault input pins to detect
    faults like overcurrent, overvoltage, etc 3 MCPWM sync input pins to synchronize MCPWM outputs signals 3
    MCPWM capture input pin to capture hall sell signal to measure time.
```

Values:

MCPWM0A = 0
PWM0A output pin

MCPWM0B
PWM0B output pin

MCPWM1A
PWM1A output pin

MCPWM1B
PWM1B output pin

MCPWM2A
PWM2A output pin

MCPWM2B
PWM2B output pin

```
MCPWM_SYNC_0
    SYNC0 input pin

MCPWM_SYNC_1
    SYNC1 input pin

MCPWM_SYNC_2
    SYNC2 input pin

MCPWM_FAULT_0
    FAULT0 input pin

MCPWM_FAULT_1
    FAULT1 input pin

MCPWM_FAULT_2
    FAULT2 input pin

MCPWM_CAP_0 = 84
    CAP0 input pin

MCPWM_CAP_1
    CAP1 input pin

MCPWM_CAP_2
    CAP2 input pin

enum mcpwm_unit_t
Select MCPWM unit.

    Values:

    MCPWM_UNIT_0 = 0
        MCPWM unit0 selected

    MCPWM_UNIT_1
        MCPWM unit1 selected

    MCPWM_UNIT_MAX
        Num of MCPWM units on ESP32

enum mcpwm_timer_t
Select MCPWM timer.

    Values:

    MCPWM_TIMER_0 = 0
        Select MCPWM timer0

    MCPWM_TIMER_1
        Select MCPWM timer1

    MCPWM_TIMER_2
        Select MCPWM timer2

    MCPWM_TIMER_MAX
        Num of MCPWM timers on ESP32

enum mcpwm_operator_t
Select MCPWM operator.

    Values:

    MCPWM_OPR_A = 0
        Select MCPWMXA, where 'X' is timer number
```

MCPWM_OPR_B

Select MCPWMXB, where 'X' is timer number

MCPWM_OPR_MAX

Num of operators to each timer of MCPWM

enum mcpwm_counter_type_t

Select type of MCPWM counter.

Values:

MCPWM_UP_COUNTER = 1

For asymmetric MCPWM

MCPWM_DOWN_COUNTER

For asymmetric MCPWM

MCPWM_UP_DOWN_COUNTER

For symmetric MCPWM, frequency is half of MCPWM frequency set

MCPWM_COUNTER_MAX

Maximum counter mode

enum mcpwm_duty_type_t

Select type of MCPWM duty cycle mode.

Values:

MCPWM_DUTY_MODE_0 = 0

Active high duty, i.e. duty cycle proportional to high time for asymmetric MCPWM

MCPWM_DUTY_MODE_1

Active low duty, i.e. duty cycle proportional to low time for asymmetric MCPWM, out of phase(inverted) MCPWM

MCPWM_DUTY_MODE_MAX

Num of duty cycle modes

enum mcpwm_carrier_os_t

MCPWM carrier oneshot mode, in this mode the width of the first pulse of carrier can be programmed.

Values:

MCPWM_ONESHOT_MODE_DIS = 0

Enable oneshot mode

MCPWM_ONESHOT_MODE_EN

Disable oneshot mode

enum mcpwm_carrier_out_ivt_t

MCPWM carrier output inversion, high frequency carrier signal active with MCPWM signal is high.

Values:

MCPWM_CARRIER_OUT_IVT_DIS = 0

Enable carrier output inversion

MCPWM_CARRIER_OUT_IVT_EN

Disable carrier output inversion

enum mcpwm_sync_signal_t

MCPWM select sync signal input.

Values:

MCPWM_SELECT_SYNC0 = 4

Select SYNC0 as input

MCPWM_SELECT_SYNC1

Select SYNC1 as input

MCPWM_SELECT_SYNC2

Select SYNC2 as input

enum mcpwm_fault_signal_t

MCPWM select fault signal input.

Values:

MCPWM_SELECT_F0 = 0

Select F0 as input

MCPWM_SELECT_F1

Select F1 as input

MCPWM_SELECT_F2

Select F2 as input

enum mcpwm_fault_input_level_t

MCPWM select triggering level of fault signal.

Values:

MCPWM_LOW_LEVEL_TGR = 0

Fault condition occurs when fault input signal goes from high to low, currently not supported

MCPWM_HIGH_LEVEL_TGR

Fault condition occurs when fault input signal goes low to high

enum mcpwm_action_on_pwmxa_t

MCPWM select action to be taken on MCPWMXA when fault occurs.

Values:

MCPWM_NO_CHANGE_IN_MCPWMXA = 0

No change in MCPWMXA output

MCPWM_FORCE_MCPWMXA_LOW

Make MCPWMXA output low

MCPWM_FORCE_MCPWMXA_HIGH

Make MCPWMXA output high

MCPWM_TOG_MCPWMXA

Make MCPWMXA output toggle

enum mcpwm_action_on_pwmxb_t

MCPWM select action to be taken on MCPWMXB when fault occurs.

Values:

MCPWM_NO_CHANGE_IN_MCPWMXB = 0

No change in MCPWMXB output

MCPWM_FORCE_MCPWMXB_LOW

Make MCPWMXB output low

MCPWM_FORCE_MCPWMXB_HIGH

Make MCPWMXB output high

MCPWM_TOG_MCPWMXB

Make MCPWMXB output toggle

enum mcpwm_capture_signal_t

MCPWM select capture signal input.

Values:

MCPWM_SELECT_CAP0 = 0

Select CAP0 as input

MCPWM_SELECT_CAP1

Select CAP1 as input

MCPWM_SELECT_CAP2

Select CAP2 as input

enum mcpwm_capture_on_edge_t

MCPWM select capture starts from which edge.

Values:

MCPWM_NEG_EDGE = 0

Capture starts from negative edge

MCPWM_POS_EDGE

Capture starts from positive edge

enum mcpwm_deadtime_type_t

MCPWM deadtime types, used to generate deadtime, RED refers to rising edge delay and FED refers to falling edge delay.

Values:

MCPWM_BYPASS_RED = 0

MCPWMXA = no change, MCPWMXB = falling edge delay

MCPWM_BYPASS_FED

MCPWMXA = rising edge delay, MCPWMXB = no change

MCPWM_ACTIVE_HIGH_MODE

MCPWMXA = rising edge delay, MCPWMXB = falling edge delay

MCPWM_ACTIVE_LOW_MODE

MCPWMXA = compliment of rising edge delay, MCPWMXB = compliment of falling edge delay

MCPWM_ACTIVE_HIGH_COMPLIMENT_MODE

MCPWMXA = rising edge delay, MCPWMXB = compliment of falling edge delay

MCPWM_ACTIVE_LOW_COMPLIMENT_MODE

MCPWMXA = compliment of rising edge delay, MCPWMXB = falling edge delay

MCPWM_ACTIVE_RED_FED_FROM_PWMXA

MCPWMXA = MCPWMXB = rising edge delay as well as falling edge delay, generated from MCPWMXA

MCPWM_ACTIVE_RED_FED_FROM_PWMXB

MCPWMXA = MCPWMXB = rising edge delay as well as falling edge delay, generated from MCPWMXB

MCPWM_DEADTIME_TYPE_MAX

2.4.9 Pulse Counter

Overview

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.

Application Example

Pulse counter with control signal and event interrupt example: [peripherals/pcnt](#).

API Reference

Header File

- `driver/include/driver/pcnt.h`

Functions

`esp_err_t pcnt_unit_config(const pcnt_config_t *pcnt_config)`

Configure Pulse Counter unit.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `pcnt_config`: Pointer of Pulse Counter unit configure parameter

`esp_err_t pcnt_get_counter_value(pcnt_unit_t pcnt_unit, int16_t *count)`

Get pulse counter value.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `pcnt_unit`: Pulse Counter unit number
- `count`: Pointer to accept counter value

`esp_err_t pcnt_counter_pause(pcnt_unit_t pcnt_unit)`

Pause PCNT counter of PCNT unit.

Return

- `ESP_OK` Success

- ESP_ERR_INVALID_ARG Parameter error

Parameters

- pcnt_unit: PCNT unit number

`esp_err_t pcnt_counter_resume (pcnt_unit_t pcnt_unit)`

Resume counting for PCNT counter.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- pcnt_unit: PCNT unit number, select from pcnt_unit_t

`esp_err_t pcnt_counter_clear (pcnt_unit_t pcnt_unit)`

Clear and reset PCNT counter value to zero.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- pcnt_unit: PCNT unit number, select from pcnt_unit_t

`esp_err_t pcnt_intr_enable (pcnt_unit_t pcnt_unit)`

Enable PCNT interrupt for PCNT unit.

Note Each Pulse counter unit has five watch point events that share the same interrupt. Configure events with `pcnt_event_enable()` and `pcnt_event_disable()`

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- pcnt_unit: PCNT unit number

`esp_err_t pcnt_intr_disable (pcnt_unit_t pcnt_unit)`

Disable PCNT interrupt for PCNT uint.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- pcnt_unit: PCNT unit number

`esp_err_t pcnt_event_enable (pcnt_unit_t unit, pcnt_evt_type_t evt_type)`

Enable PCNT event of PCNT unit.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number
- evt_type: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).

`esp_err_t pcnt_event_disable (pcnt_unit_t unit, pcnt_evt_type_t evt_type)`

Disable PCNT event of PCNT unit.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number
- evt_type: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).

`esp_err_t pcnt_set_event_value (pcnt_unit_t unit, pcnt_evt_type_t evt_type, int16_t value)`

Set PCNT event value of PCNT unit.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number
- evt_type: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).
- value: Counter value for PCNT event

`esp_err_t pcnt_get_event_value (pcnt_unit_t unit, pcnt_evt_type_t evt_type, int16_t *value)`

Get PCNT event value of PCNT unit.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number
- evt_type: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).
- value: Pointer to accept counter value for PCNT event

```
esp_err_t pcnt_isr_register(void (*fn)) void *  
    , void *arg, int intr_alloc_flags, pcnt_isr_handle_t *handle
```

Register PCNT interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Function pointer error.

Parameters

- fn: Interrupt handler function.
- arg: Parameter for handler function
- intr_alloc_flags: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.
- handle: Pointer to return handle. If non-NUL, a handle for the interrupt will be returned here.

```
esp_err_t pcnt_set_pin(pcnt_unit_t unit, pcnt_channel_t channel, int pulse_io, int ctrl_io)
```

Configure PCNT pulse signal input pin and control input pin.

Note Set to PCNT_PIN_NOT_USED if unused.

Note Set 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

Parameters

- ctrl_io: Control signal input GPIO

```
esp_err_t pcnt_filter_enable(pcnt_unit_t unit)
```

Enable PCNT input filter.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number

```
esp_err_t pcnt_filter_disable(pcnt_unit_t unit)
```

Disable PCNT input filter.

Return

- ESP_OK Success

- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number

`esp_err_t pcnt_set_filter_value (pcnt_unit_t unit, uint16_t filter_val)`

Set PCNT filter value.

Note filter_val is a 10-bit value, so the maximum filter_val should be limited to 1023.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number
- filter_val: PCNT signal filter value, counter in APB_CLK cycles. Any pulses lasting shorter than this will be ignored when the filter is enabled.

`esp_err_t pcnt_get_filter_value (pcnt_unit_t unit, uint16_t *filter_val)`

Get PCNT filter value.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number
- filter_val: Pointer to accept PCNT filter value.

`esp_err_t pcnt_set_mode (pcnt_unit_t unit, pcnt_channel_t channel, pcnt_count_mode_t pos_mode, pcnt_count_mode_t neg_mode, pcnt_ctrl_mode_t hctrl_mode, pcnt_ctrl_mode_t lctrl_mode)`

Set PCNT counter mode.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- unit: PCNT unit number
- channel: PCNT channel number
- pos_mode: Counter mode when detecting positive edge
- neg_mode: Counter mode when detecting negative edge
- hctrl_mode: Counter mode when control signal is high level
- lctrl_mode: Counter mode when control signal is low level

Structures

```
struct pcnt_config_t
    Pulse Counter configure struct.
```

Public Members

```
int pulse_gpio_num
    Pulse input gpio_num, if you want to use gpio16, pulse_gpio_num = 16, a negative value will be ignored

int ctrl_gpio_num
    Control signal input gpio_num, 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
    Pin are not used
```

Type Definitions

```
typedef intr_handle_t pcnt_isr_handle_t
```

Enumerations

```
enum pcnt_ctrl_mode_t
    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
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
Values:
PCNT_UNIT_0 = 0
PCNT unit0

PCNT_UNIT_1 = 1
PCNT unit1

PCNT_UNIT_2 = 2
PCNT unit2

PCNT_UNIT_3 = 3
PCNT unit3

PCNT_UNIT_4 = 4
PCNT unit4

PCNT_UNIT_5 = 5
PCNT unit5

PCNT_UNIT_6 = 6
PCNT unit6

PCNT_UNIT_7 = 7
PCNT unit7

PCNT_UNIT_MAX

enum pcnt_channel_t
Values:
PCNT_CHANNEL_0 = 0x00
PCNT channel0

PCNT_CHANNEL_1 = 0x01
PCNT channel1

PCNT_CHANNEL_MAX

enum pcnt_evt_type_t
Values:
```

PCNT_EVT_L_LIM = 0
PCNT watch point event: Minimum counter value

PCNT_EVT_H_LIM = 1
PCNT watch point event: Maximum counter value

PCNT_EVT_THRES_0 = 2
PCNT watch point event: threshold0 value event

PCNT_EVT_THRES_1 = 3
PCNT watch point event: threshold1 value event

PCNT_EVT_ZERO = 4
PCNT watch point event: counter value zero event

PCNT_EVT_MAX

2.4.10 RMT

Overview

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 many other types of signals.

Application Example

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. Channel0 can use at most 8 blocks of memory, accordingly channel7 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 channel0-7.

```
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: To enable RMT transmitter loop sending mode.

If `set` true, transmitter will `continue` sending `from the` first ↗
data to the last data `in` channel0-7 again `and` again.

`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, channel0-7 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 source clock:

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 source clock:

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 channel0-7 in idle state.

`esp_err_t rmt_get_status (rmt_channel_t channel, uint32_t *status)`
Get RMT status.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel (0-7)
- `status`: Pointer to accept channel status.

`void rmt_set_intr_enable_mask (uint32_t mask)`
Set mask value to RMT interrupt enable register.

Parameters

- `mask`: Bit mask to set to the register

`void rmt_clr_intr_enable_mask (uint32_t mask)`
Clear mask value to RMT interrupt enable register.

Parameters

- mask: Bit mask to clear the register

`esp_err_t rmt_set_rx_intr_en (rmt_channel_t channel, bool en)`
Set RMT RX interrupt enable.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel (0 - 7)
- en: enable or disable RX interrupt.

`esp_err_t rmt_set_err_intr_en (rmt_channel_t channel, bool en)`
Set RMT RX error interrupt enable.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel (0 - 7)
- en: enable or disable RX err interrupt.

`esp_err_t rmt_set_tx_intr_en (rmt_channel_t channel, bool en)`
Set RMT TX interrupt enable.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel (0 - 7)
- en: enable or disable TX interrupt.

`esp_err_t rmt_set_tx_thr_intr_en (rmt_channel_t channel, bool en, uint16_t evt_thresh)`
Set RMT TX threshold event interrupt enable.

Causes an interrupt when a threshold number of items have been transmitted.

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 pins.

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 structor

`esp_err_t rmt_isr_register (void (*fn)) void *`
, void *arg, int intr_alloc_flags, `rmt_isr_handle_t` *handleregister 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 handler function
- intr_alloc_flags: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See esp_intr_alloc.h for more info.
- handle: If non-zero, a handle to later clean up the ISR gets stored here.

`esp_err_t rmt_isr_deregister (rmt_isr_handle_t handle)`
Deregister previously registered RMT interrupt handler.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Handle invalid

Parameters

- handle: Handle obtained from rmt_isr_register

`esp_err_t rmt_fill_tx_items (rmt_channel_t channel, const rmt_item32_t *item, uint16_t item_num, uint16_t mem_offset)`

Fill memory data of channel with given RMT items.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel (0 - 7)
- item: Pointer of items.
- item_num: RMT sending items number.
- mem_offset: Index offset of memory.

`esp_err_t rmt_driver_install (rmt_channel_t channel, size_t rx_buf_size, int intr_alloc_flags)`

Initialize RMT driver.

Return

- ESP_ERR_INVALID_STATE Driver is already installed, call rmt_driver_uninstall first.
- ESP_ERR_NO_MEM Memory allocation failure
- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel (0 - 7)
- rx_buf_size: Size of RMT RX ringbuffer. Can be 0 if the RX ringbuffer is not used.
- intr_alloc_flags: Flags for the RMT driver interrupt handler. Pass 0 for default flags. See esp_intr_alloc.h for details.

`esp_err_t rmt_driver_uninstall (rmt_channel_t channel)`

Uninstall RMT driver.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel (0 - 7)

`esp_err_t rmt_write_items (rmt_channel_t channel, const rmt_item32_t *rmt_item, int item_num, bool wait_tx_done)`

RMT send waveform from rmt_item array.

This API allows user to send waveform with any length.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel (0 - 7)
- rmt_item: head point of RMT items array.
- item_num: RMT data item number.
- wait_tx_done: If set 1, it will block the task and wait for sending done.

If `set 0`, it will **not** wait **and return** immediately.

@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.**

`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 Crossed 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 to wait for transmission to be complete

`esp_err_t rmt_get_ringbuf_handle (rmt_channel_t channel, RingbufHandle_t *buf_handle)`
 Get ringbuffer from UART.

Users can get the RMT RX ringbuffer handler, 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 handler to accept RX ringbuffer handler.

Structures

`struct rmt_tx_config_t`

Data struct of RMT TX configure parameters.

Public Members

`bool loop_en`

RMT loop output mode

`uint32_t carrier_freq_hz`

RMT carrier frequency

`uint8_t carrier_duty_percent`

RMT carrier duty (%)

`rmt_carrier_level_t carrier_level`

RMT carrier level

`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 filer enable

`uint8_t filter_ticks_thresh`

RMT filter tick number

`uint16_t idle_threshold`

RMT RX idle threshold

`struct rmt_config_t`

Data struct of RMT configure parameters.

Public Members

`rmt_mode_t rmt_mode`

RMT mode: transmitter or receiver

`rmt_channel_t channel`

RMT channel

```
uint8_t clk_div
    RMT channel counter divider

gpio_num_t gpio_num
    RMT GPIO number

uint8_t mem_block_num
    RMT memory block number

rmt_tx_config_t tx_config
    RMT TX parameter

rmt_rx_config_t rx_config
    RMT RX parameter
```

Macros

```
RMT_MEM_BLOCK_BYTE_NUM
RMT_MEM_ITEM_NUM
```

Type Definitions

```
typedef intr_handle_t rmt_isr_handle_t
```

Enumerations

```
enum rmt_channel_t
    Values:
        RMT_CHANNEL_0 =0
            RMT Channel0
        RMT_CHANNEL_1
            RMT Channel1
        RMT_CHANNEL_2
            RMT Channel2
        RMT_CHANNEL_3
            RMT Channel3
        RMT_CHANNEL_4
            RMT Channel4
        RMT_CHANNEL_5
            RMT Channel5
        RMT_CHANNEL_6
            RMT Channel6
        RMT_CHANNEL_7
            RMT Channel7
        RMT_CHANNEL_MAX
```

```
enum rmt_mem_owner_t
    Values:
```

RMT_MEM_OWNER_TX = 0
RMT RX mode, RMT transmitter owns the memory block

RMT_MEM_OWNER_RX = 1
RMT RX mode, RMT receiver owns the memory block

RMT_MEM_OWNER_MAX

enum rmt_source_clk_t

Values:

RMT_BASECLK_REF = 0
RMT source clock system reference tick, 1MHz by default(Not supported in this version)

RMT_BASECLK_APB

RMT source clock is APB CLK, 80Mhz by default

RMT_BASECLK_MAX

enum rmt_data_mode_t

Values:

RMT_DATA_MODE_FIFO = 0

RMT_DATA_MODE_MEM = 1

RMT_DATA_MODE_MAX

enum rmt_mode_t

Values:

RMT_MODE_TX = 0

RMT TX mode

RMT_MODE_RX

RMT RX mode

RMT_MODE_MAX

enum rmt_idle_level_t

Values:

RMT_IDLE_LEVEL_LOW = 0

RMT TX idle level: low Level

RMT_IDLE_LEVEL_HIGH

RMT TX idle level: high Level

RMT_IDLE_LEVEL_MAX

enum rmt_carrier_level_t

Values:

RMT_CARRIER_LEVEL_LOW = 0

RMT carrier wave is modulated for low Level output

RMT_CARRIER_LEVEL_HIGH

RMT carrier wave is modulated for high Level output

RMT_CARRIER_LEVEL_MAX

2.4.11 SDMMC Host Peripheral

Overview

SDMMC peripheral supports SD and MMC memory cards and SDIO cards. SDMMC software builds on top of SDMMC driver and consists of the following parts:

1. SDMMC host driver (`driver/sdmmc_host.h`) — this driver provides APIs to send commands to the slave device(s), send and receive data, and handling error conditions on the bus.
2. SDMMC protocol layer (`sdmmc_cmd.h`) — this component handles specifics of SD protocol such as card initialization and data transfer commands. Despite the name, only SD (SDSC/SDHC/SDXC) cards are supported at the moment. Support for MCC/eMMC cards can be added in the future.

Protocol layer works with the host via `sdmmc_host_t` structure. This structure contains pointers to various functions of the host.

In addition to SDMMC Host peripheral, ESP32 has SPI peripherals which can also be used to work with SD cards. This is supported using a variant of the host driver, `driver/sdspi_host.h`. This driver has the same interface as SDMMC host driver, and the protocol layer can use either of two.

Application Example

An example which combines SDMMC driver with FATFS library is provided in `examples/storage/sd_card` directory. This example initializes the card, writes and reads data from it using POSIX and C library APIs. See `README.md` file in the example directory for more information.

Protocol layer APIs

Protocol layer is given `sdmmc_host_t` structure which describes the SD/MMC host driver, lists its capabilities, and provides pointers to functions of the driver. Protocol layer stores card-specific information in `sdmmc_card_t` structure. When sending commands to the SD/MMC host driver, protocol layer uses `sdmmc_command_t` structure to describe the command, argument, expected return value, and data to transfer, if any.

Normal usage of the protocol layer is as follows:

1. Call the host driver functions to initialize the host (e.g. `sdmmc_host_init`, `sdmmc_host_init_slot`).
2. Call `sdmmc_card_init` to initialize the card, passing it host driver information (`host`) and a pointer to `sdmmc_card_t` structure which will be filled in (`card`).
3. To read and write sectors of the card, use `sdmmc_read_sectors` and `sdmmc_write_sectors`, passing the pointer to card information structure (`card`).
4. When card is not used anymore, call the host driver function to disable SDMMC host peripheral and free resources allocated by the driver (e.g. `sdmmc_host_deinit`).

Most applications need to use the protocol layer only in one task; therefore the protocol layer doesn't implement any kind of locking on the `sdmmc_card_t` structure, or when accessing SDMMC host driver. Such locking has to be implemented in the higher layer, if necessary (e.g. in the filesystem driver).

struct 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

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

SDMMC_HOST_FLAG_1BIT
    host supports 1-line SD and MMC protocol

SDMMC_HOST_FLAG_4BIT
    host supports 4-line SD and MMC protocol

SDMMC_HOST_FLAG_8BIT
    host supports 8-line MMC protocol

SDMMC_HOST_FLAG_SPI
    host supports SPI protocol

SDMMC_FREQ_DEFAULT
    SD/MMC Default speed (limited by clock divider)

SDMMC_FREQ_HIGHSPEED
    SD High speed (limited by clock divider)

SDMMC_FREQ_PROBING
    SD/MMC probing speed

struct sdmmc_command_t
    SD/MMC command information

```

Public Members

```

uint32_t opcode
    SD or MMC command index

uint32_t arg
    SD/MMC command argument

```

```
sdmmc_response_t response
    response buffer

void *data
    buffer to send or read into

size_t datalen
    length of data buffer

size_t blklen
    block length

int flags
    see below

esp_err_t error
    error returned from transfer

struct sdmmc_card_t
    SD/MMC card information structure
```

Public Members

```
sdmmc_host_t host
    Host with which the card is associated

uint32_t ocr
    OCR (Operation Conditions Register) value

sdmmc_cid_t cid
    decoded CID (Card IDentification) register value

sdmmc_csd_t csd
    decoded CSD (Card-Specific Data) register value

sdmmc_scr_t scr
    decoded SCR (SD card Configuration Register) value

uint16_t rca
    RCA (Relative Card Address)

struct sdmmc_csd_t
    Decoded values from SD card Card Specific Data register
```

Public Members

```
int csd_ver
    CSD structure format

int mmc_ver
    MMC version (for CID format)

int capacity
    total number of sectors

int sector_size
    sector size in bytes

int read_block_len
    block length for reads
```

```

int card_command_class
    Card Command Class for SD

int tr_speed
    Max transfer speed

struct sdmmc_cid_t
    Decoded values from SD card Card IDentification register

```

Public Members

```

int mfg_id
    manufacturer identification number

int oem_id
    OEM/product identification number

char name[8]
    product name (MMC v1 has the longest)

int revision
    product revision

int serial
    product serial number

int date
    manufacturing date

struct sdmmc_scr_t
    Decoded values from SD Configuration Register

```

Public Members

```

int sd_spec
    SD Physical layer specification version, reported by card

int bus_width
    bus widths supported by card: BIT(0) — 1-bit bus, BIT(2) — 4-bit bus

esp_err_t sdmmc_card_init (const sdmmc_host_t *host, sdmmc_card_t *out_card)
    Probe and initialize SD/MMC card using given host

```

Note Only SD cards (SDSC and SDHC/SDXC) are supported now. Support for MMC/eMMC cards will be added later.

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- host: pointer to structure defining host controller
- out_card: pointer to structure which will receive information about the card when the function completes

```
esp_err_t sdmmc_write_sectors (sdmmc_card_t *card, const void *src, size_t start_sector, size_t sector_count)
```

Write given number of sectors to SD/MMC card

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- *card*: pointer to card information structure previously initialized using `sdmmc_card_init`
- *src*: pointer to data buffer to read data from; data size must be equal to *sector_count* * *card->csd.sector_size*
- *start_sector*: sector where to start writing
- *sector_count*: number of sectors to write

```
esp_err_t sdmmc_read_sectors (sdmmc_card_t *card, void *dst, size_t start_sector, size_t sector_count)
```

Write given number of sectors to SD/MMC card

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- *card*: pointer to card information structure previously initialized using `sdmmc_card_init`
- *dst*: pointer to data buffer to write into; buffer size must be at least *sector_count* * *card->csd.sector_size*
- *start_sector*: sector where to start reading
- *sector_count*: number of sectors to read

SDMMC host driver APIs

On the ESP32, SDMMC host peripheral has two slots:

- Slot 0 (`SDMMC_HOST_SLOT_0`) is an 8-bit slot. It uses `HS1_*` signals in the PIN MUX.
- Slot 1 (`SDMMC_HOST_SLOT_1`) is a 4-bit slot. It uses `HS2_*` signals in the PIN MUX.

Card Detect and Write Protect signals can be routed to arbitrary pins using GPIO matrix. To use these pins, set `gpio_cd` and `gpio_wp` members of `sdmmc_slot_config_t` structure when calling `sdmmc_host_init_slot`.

Of all the functions listed below, only `sdmmc_host_init`, `sdmmc_host_init_slot`, and `sdmmc_host_deinit` will be used directly by most applications. Other functions, such as `sdmmc_host_set_bus_width`, `sdmmc_host_set_card_clk`, and `sdmmc_host_do_transaction` will be called by the SD/MMC protocol layer via function pointers in `sdmmc_host_t` structure.

```
esp_err_t sdmmc_host_init ()
```

Initialize SDMMC host peripheral.

Note This function is not thread safe

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if sdmmc_host_init was already called
- ESP_ERR_NO_MEM if memory can not be allocated

SDMMC_HOST_SLOT_0

SDMMC slot 0.

SDMMC_HOST_SLOT_1

SDMMC slot 1.

SDMMC_HOST_DEFAULT

Default *sdmmc_host_t* structure initializer for SDMMC peripheral.

Uses SDMMC peripheral, with 4-bit mode enabled, and max frequency set to 20MHz

SDMMC_SLOT_WIDTH_DEFAULT

use the default width for the slot (8 for slot 0, 4 for slot 1)

`esp_err_t sdmmc_host_init_slot(int slot, const sdmmc_slot_config_t *slot_config)`

Initialize given slot of SDMMC peripheral.

On the ESP32, SDMMC peripheral has two slots:

- Slot 0: 8-bit wide, maps to HS1_* signals in PIN MUX
- Slot 1: 4-bit wide, maps to HS2_* signals in PIN MUX

Card detect and write protect signals can be routed to arbitrary GPIOs using GPIO matrix.

Note This function is not thread safe

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if host has not been initialized using sdmmc_host_init

Parameters

- slot: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- slot_config: additional configuration for the slot

struct sdmmc_slot_config_t

Extra configuration for SDMMC peripheral slot

Public Members***gpio_num_t gpio_cd***

GPIO number of card detect signal.

gpio_num_t gpio_wp

GPIO number of write protect signal.

uint8_t width

Bus width used by the slot (might be less than the max width supported)

SDMMC_SLOT_NO_CD

indicates that card detect line is not used

SDMMC_SLOT_NO_WP

indicates that write protect line is not used

SDMMC_SLOT_CONFIG_DEFAULT

Macro defining default configuration of SDMMC host slot

`esp_err_t sdmmc_host_set_bus_width (int slot, size_t width)`

Select bus width to be used for data transfer.

SD/MMC card must be initialized prior to this command, and a command to set bus width has to be sent to the card (e.g. SD_APP_SET_BUS_WIDTH)

Note This function is not thread safe

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if slot number or width is not valid

Parameters

- slot: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- width: bus width (1, 4, or 8 for slot 0; 1 or 4 for slot 1)

`esp_err_t sdmmc_host_set_card_clk (int slot, uint32_t freq_khz)`

Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

Note This function is not thread safe

Return

- ESP_OK on success
- other error codes may be returned in the future

Parameters

- slot: slot number (SDMMC_HOST_SLOT_0 or SDMMC_HOST_SLOT_1)
- freq_khz: card clock frequency, in kHz

`esp_err_t sdmmc_host_do_transaction (int slot, sdmmc_command_t *cmdinfo)`

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

Note This function is not thread safe w.r.t. init/deinit functions, and bus width/clock speed configuration functions. Multiple tasks can call sdmmc_host_do_transaction as long as other sdmmc_host_* functions are not called.

Attention Data buffer passed in cmdinfo->data must be in DMA capable memory

Return

- ESP_OK on success
- ESP_ERR_TIMEOUT if response or data transfer has timed out
- ESP_ERR_INVALID_CRC if response or data transfer CRC check has failed

- `ESP_ERR_INVALID_RESPONSE` if the card has sent an invalid response
- `ESP_ERR_INVALID_SIZE` if the size of data transfer is not valid in SD protocol
- `ESP_ERR_INVALID_ARG` if the data buffer is not in DMA capable memory

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)
- `cmdinfo`: pointer to structure describing command and data to transfer

`esp_err_t sdmmc_host_deinit()`

Disable SDMMC host and release allocated resources.

Note This function is not thread safe**Return**

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `sdmmc_host_init` function has not been called

SD SPI driver APIs

SPI controllers accessible via `spi_master` driver (HSPI, VSPI) can be used to work with SD cards. In SPI mode, SD driver has lower throughput than in 1-line SD mode. However SPI mode makes pin selection more flexible, as SPI peripheral can be connected to any ESP32 pins using GPIO Matrix. SD SPI driver uses software controlled CS signal. Currently SD SPI driver assumes that it can use the SPI controller exclusively, so applications which need to share SPI bus between SD cards and other peripherals need to make sure that SD card and other devices are not used at the same time from different tasks.

SD SPI driver is represented using an `sdmmc_host_t` structure initialized using `SDSPI_HOST_DEFAULT` macro. For slot initialization, `SDSPI_SLOT_CONFIG_DEFAULT` can be used to fill in default pin mapping, which is the same as the pin mapping in SD mode.

SD SPI driver APIs are very similar to SDMMC host APIs. As with the SDMMC host driver, only `sdspi_host_init`, `sdspi_host_init_slot`, and `sdspi_host_deinit` functions are normally used by the applications. Other functions are called by the protocol level driver via function pointers in `sdmmc_host_t` structure.

`esp_err_t sdspi_host_init()`

Initialize SD SPI driver.

Note This function is not thread safe**Return**

- `ESP_OK` on success
- other error codes may be returned in future versions

SDSPI_HOST_DEFAULTDefault `sdmmc_host_t` structure initializer for SD over SPI driver.

Uses SPI mode and max frequency set to 20MHz

'slot' can be set to one of `HSPI_HOST`, `VSPI_HOST`.`esp_err_t 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

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)

SDSPI_SLOT_NO_CD

indicates that card detect line is not used

SDSPI_SLOT_NO_WP

indicates that write protect line is not used

SDSPI_SLOT_CONFIG_DEFAULT

Macro defining default configuration of SPI host

esp_err_t sdspi_host_set_card_clk (int slot, uint32_t freq_khz)
Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

Note This function is not thread safe

Return

- ESP_OK on success
- other error codes may be returned in the future

Parameters

- slot: SPI controller (HSPI_HOST or VSPI_HOST)
- freq_khz: card clock frequency, in kHz

`esp_err_t sdspi_host_do_transaction (int slot, sdmmc_command_t *cmdinfo)`

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

Note This function is not thread safe w.r.t. init/deinit functions, and bus width/clock speed configuration functions. Multiple tasks can call sdspi_host_do_transaction as long as other sdspi_host_* functions are not called.

Return

- ESP_OK on success
- ESP_ERR_TIMEOUT if response or data transfer has timed out
- ESP_ERR_INVALID_CRC if response or data transfer CRC check has failed
- ESP_ERR_INVALID_RESPONSE if the card has sent an invalid response

Parameters

- slot: SPI controller (HSPI_HOST or VSPI_HOST)
- cmdinfo: pointer to structure describing command and data to transfer

`esp_err_t sdspi_host_deinit ()`

Release resources allocated using sdspi_host_init.

Note This function is not thread safe

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if sdspi_host_init function has not been called

2.4.12 Sigma-delta Modulation

Overview

ESP32 has a second-order sigma-delta modulation module. This driver configures the channels of the sigma-delta module.

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 $V_{dc} = V_{DDIO} / 256 * duty + V_{DDIO}/2$, V_{DDIO} is 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 channel0

```
SIGMADELTA_CHANNEL_1 = 1
```

Sigma-delta channel1

```
SIGMADELTA_CHANNEL_2 = 2
```

Sigma-delta channel2

```
SIGMADELTA_CHANNEL_3 = 3
```

Sigma-delta channel3

```
SIGMADELTA_CHANNEL_4 = 4
```

Sigma-delta channel4

```
SIGMADELTA_CHANNEL_5 = 5
```

Sigma-delta channel5

```
SIGMADELTA_CHANNEL_6 = 6
```

Sigma-delta channel6

```
SIGMADELTA_CHANNEL_7 = 7
```

Sigma-delta channel7

SIGMADELTA_CHANNEL_MAX

2.4.13 SPI Master driver

Overview

The ESP32 has four SPI peripheral devices, called SPI0, SPI1, HSPI and VSPI. SPI0 is entirely dedicated to the flash cache the ESP32 uses to map the SPI flash device it is connected to into memory. SPI1 is connected to the same hardware lines as SPI0 and is used to write to the flash chip. HSPI and VSPI are free to use. SPI1, HSPI and VSPI all have three chip select lines, allowing them to drive up to three SPI devices each as a master.

The `spi_master` driver

The `spi_master` driver allows easy communicating with SPI slave devices, even in a multithreaded environment. It fully transparently handles DMA transfers to read and write data and automatically takes care of multiplexing between different SPI slaves on the same master

Terminology

The `spi_master` driver uses the following terms:

- Host: The SPI peripheral inside the ESP32 initiating the SPI transmissions. One of SPI, HSPI or VSPI. (For now, only HSPI or VSPI are actually supported in the driver; it will support all 3 peripherals somewhere in the future.)
- Bus: The SPI bus, common to all SPI devices connected to one host. In general the bus consists of the miso, mosi, sclk and optionally quadwp and quadhd signals. The SPI slaves are connected to these signals in parallel.
 - miso - Also known as q, this is the input of the serial stream into the ESP32
 - mosi - Also known as d, this is the output of the serial stream from the ESP32
 - sclk - Clock signal. Each data bit is clocked out or in on the positive or negative edge of this signal
 - quadwp - Write Protect signal. Only used for 4-bit (qio/qout) transactions.
 - quadhd - Hold signal. Only used for 4-bit (qio/qout) transactions.
- Device: A SPI slave. Each SPI slave has its own chip select (CS) line, which is made active when a transmission to/from the SPI slave occurs.
- Transaction: One instance of CS going active, data transfer from and/or to a device happening, and CS going inactive again. Transactions are atomic, as in they will never be interrupted by another transaction.

SPI transactions

A transaction on the SPI bus consists of five phases, any of which may be skipped:

- The command phase. In this phase, a command (0-16 bit) is clocked out.
- The address phase. In this phase, an address (0-64 bit) is clocked out.
- The write phase. The master sends data to the slave.
- The dummy phase. The phase is configurable, used to meet the timing requirements.
- The read phase. The slave sends data to the master.

In full duplex, the read and write phases are combined, causing the SPI host to read and write data simultaneously. The total transaction length is decided by `dev_conf.command_bits + dev_conf.address_bits + trans_conf.length`, while the `trans_conf.rx_length` only determines length of data received into the buffer.

In half duplex, the length of write phase and read phase are decided by `trans_conf.length` and `trans_conf.rx_length` respectively. ** Note that a half duplex transaction with both a read and write phase is not supported when using DMA. ** If such transaction is needed, you have to use one of the alternative solutions:

1. use full-duplex mode instead.
2. disable the DMA by set the last parameter to 0 in bus initialization function just as belows:
`ret=spi_bus_initialize(VSPI_HOST, &buscfg, 0);`
 this may prohibit you from transmitting and receiving data longer than 32 bytes.
3. try to use command and address field to replace the write phase.

The command and address phase are optional in that not every SPI device will need to be sent a command and/or address. This is reflected in the device configuration: when the `command_bits` or `address_bits` fields are set to zero, no command or address phase is done.

Something similar is true for the read and write phase: not every transaction needs both data to be written as well as data to be read. When `rx_buffer` is NULL (and `SPI_USE_RXDATA` is not set) the read phase is skipped. When `tx_buffer` is NULL (and `SPI_USE_TXDATA` is not set) the write phase is skipped.

Using the spi_master driver

- Initialize a SPI bus by calling `spi_bus_initialize`. Make sure to set the correct IO pins in the `bus_config` struct. Take care to set signals that are not needed to -1.
- Tell the driver about a SPI slave device connected to the bus by calling `spi_bus_add_device`. Make sure to configure any timing requirements the device has in the `dev_config` structure. You should now have a handle for the device, to be used when sending it a transaction.
- To interact with the device, fill one or more `spi_transaction_t` structure with any transaction parameters you need. Either queue all transactions by calling `spi_device_queue_trans`, later querying the result using `spi_device_get_trans_result`, or handle all requests synchronously by feeding them into `spi_device_transmit`.
- Optional: to unload the driver for a device, call `spi_bus_remove_device` with the device handle as an argument
- Optional: to remove the driver for a bus, make sure no more drivers are attached and call `spi_bus_free`.

Transaction data

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 belows:

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.

Sometimes, the amount of data is very small making it less than optimal allocating a separate buffer for it. If the data to be transferred is 32 bits or less, it can be stored in the transaction struct itself. For transmitted data, use the `tx_data` member for this and set the `SPI_USE_TXDATA` flag on the transmission. For received data, use `rx_data` and set `SPI_USE_RXDATA`. In both cases, do not touch the `tx_buffer` or `rx_buffer` members, because they use the same memory locations as `tx_data` and `rx_data`.

Application Example

Display graphics on the 320x240 LCD of WROVER-Kits: [peripherals/spi_master](#).

API Reference - SPI Common

Header File

- [driver/include/driver/spi_common.h](#)

Functions

`bool spicommon_periph_claim(spi_host_device_t host)`

Try to claim a SPI peripheral.

Call this if your driver wants to manage a SPI peripheral.

Return True if peripheral is claimed successfully; false if peripheral already is claimed.

Parameters

- `host`: Peripheral to claim

`bool spicommon_periph_free(spi_host_device_t host)`

Return the SPI peripheral so another driver can claim it.

Return True if peripheral is returned successfully; false if peripheral was free to claim already.

Parameters

- `host`: Peripheral to return

`bool spicommon_dma_chan_claim(int dma_chan)`

Try to claim a SPI DMA channel.

Call this if your driver wants to use SPI with a DMA 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, int flags, bool *is_native)
```

Connect a SPI peripheral to GPIO pins.

This routine is used to connect a SPI peripheral to the IO-pads and DMA channel given in the arguments. Depending on the IO-pads requested, the routing is done either using the IO_mux or using the GPIO matrix.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

Parameters

- `host`: SPI peripheral to be routed
- `bus_config`: Pointer to a `spi_bus_config` struct detailing the GPIO pins
- `dma_chan`: DMA-channel (1 or 2) to use, or 0 for no DMA.
- `flags`: Combination of `SPICOMMON_BUSFLAG_*` flags
- `is_native`: A value of ‘true’ will be written to this address if the GPIOs can be routed using the `IO_mux`, ‘false’ if the GPIO matrix is used.

```
esp_err_t spicommon_bus_free_io(spi_host_device_t host)
```

Free the IO used by a SPI peripheral.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

Parameters

- `host`: SPI peripheral to be freed

```
void spicommon_cs_initialize(spi_host_device_t host, int cs_io_num, int cs_num, int force_gpio_matrix)
```

Initialize a Chip Select pin for a specific SPI peripheral.

Parameters

- `host`: SPI peripheral
- `cs_io_num`: GPIO pin to route
- `cs_num`: CS id to route
- `force_gpio_matrix`: If true, CS will always be routed through the GPIO matrix. If false, if the GPIO number allows it, the routing will happen through the `IO_mux`.

```
void spicommon_cs_free(spi_host_device_t host, int cs_num)
```

Free a chip select line.

Parameters

- `host`: SPI peripheral
- `cs_num`: CS id to free

void **spicommon_setup_dma_desc_links** (lldesc_t *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 this 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* referring to these lines will be ignored and can thus safely be left uninitialized.

Public Members

```
int mosi_io_num
```

GPIO pin for Master Out Slave In (=spi_d) signal, or -1 if not used.

```
int miso_io_num
```

GPIO pin for Master In Slave Out (=spi_q) signal, or -1 if not used.

```
int sclk_io_num
```

GPIO pin for Spi CLocK signal, or -1 if not used.

```
int quadwp_io_num
```

GPIO pin for WP (Write Protect) signal which is used as D2 in 4-bit communication modes, or -1 if not used.

```
int quadhd_io_num
```

GPIO pin for HD (Hold) signal which is used as D3 in 4-bit communication modes, or -1 if not used.

```
int max_transfer_sz
```

Maximum transfer size, in bytes. Defaults to 4094 if 0.

Macros

```
SPI_MAX_DMA_LEN
```

```
SPICOMMON_BUSFLAG_SLAVE
```

Initialize I/O in slave mode.

SPICOMMON_BUSFLAG_MASTER

Initialize I/O in master mode.

SPICOMMON_BUSFLAG_QUAD

Also initialize WP/HD pins, if specified.

Type Definitions

```
typedef void (*dmaworkaround_cb_t) (void *arg)  
Callback, to be called when a DMA engine reset is completed
```

Enumerations

enum spi_host_device_t

Enum with the three SPI peripherals that are software-accessible in it.

Values:

SPI_HOST =0

SPI1, SPI.

HSPI_HOST =1

SPI2, HSPI.

VSPI_HOST =2

SPI3, VSPI.

API Reference - SPI Master

Header File

- driver/include/driver/spi_master.h

Functions

```
esp_err_t spi_bus_initialize (spi_host_device_t host, const spi_bus_config_t *bus_config, int  
dma_chan)
```

Initialize a SPI bus.

Warning For now, only supports HSPI and VSPI.

Warning If a DMA channel is selected, any transmit and receive buffer used should be allocated in DMA-capable memory.

Return

- ESP_ERR_INVALID_ARG if configuration is invalid
- ESP_ERR_INVALID_STATE if host already is in use
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

Parameters

- host: SPI peripheral that controls this bus

- **bus_config:** Pointer to a `spi_bus_config_t` struct specifying how the host should be initialized
- **dma_chan:** Either channel 1 or 2, or 0 in the case when no DMA is required. Selecting a DMA channel for a SPI bus allows transfers on the bus to have sizes only limited by the amount of internal memory. Selecting no DMA channel (by passing the value 0) limits the amount of bytes transferred to a maximum of 32.

`esp_err_t spi_bus_free (spi_host_device_t host)`

Free a SPI bus.

Warning In order for this to succeed, all devices have to be removed first.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_INVALID_STATE` if not all devices on the bus are freed
- `ESP_OK` on success

Parameters

- `host:` SPI peripheral to free

`esp_err_t spi_bus_add_device (spi_host_device_t host, spi_device_interface_config_t *dev_config, spi_device_handle_t *handle)`

Allocate a device on a SPI bus.

This initializes the internal structures for a device, plus allocates a CS pin on the indicated SPI master peripheral and routes it to the indicated GPIO. All SPI master devices have three CS pins and can thus control up to three devices.

Note While in general, speeds up to 80MHz on the dedicated SPI pins and 40MHz on GPIO-matrix-routed pins are supported, full-duplex transfers routed over the GPIO matrix only support speeds up to 26MHz.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_NOT_FOUND` if host doesn't have any free CS slots
- `ESP_ERR_NO_MEM` if out of memory
- `ESP_OK` on success

Parameters

- `host:` SPI peripheral to allocate device on
- `dev_config:` SPI interface protocol config for the device
- `handle:` Pointer to variable to hold the device handle

`esp_err_t spi_bus_remove_device (spi_device_handle_t handle)`

Remove a device from the SPI bus.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_INVALID_STATE` if device already is freed
- `ESP_OK` on success

Parameters

- handle: Device handle to free

```
esp_err_t spi_device_queue_trans(spi_device_handle_t handle, spi_transaction_t *trans_desc, TickType_t ticks_to_wait)
```

Queue a SPI transaction for execution.

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
- ticks_to_wait: Ticks to wait until there's room in the queue; use portMAX_DELAY to never time out.

```
esp_err_t spi_device_get_trans_result(spi_device_handle_t handle, spi_transaction_t **trans_desc, TickType_t ticks_to_wait)
```

Get the result of a SPI transaction queued earlier.

This routine will wait until a transaction to the given device (queued earlier with spi_device_queue_trans) has successfully completed. It will then return the description of the completed transaction so software can inspect the result and e.g. free the memory or re-use the buffers.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Parameters

- handle: Device handle obtained using spi_host_add_dev
- trans_desc: Pointer to variable able to contain a pointer to the description of the transaction that is executed. The descriptor should not be modified until the descriptor is returned by spi_device_get_trans_result.
- ticks_to_wait: Ticks to wait until there's a returned item; use portMAX_DELAY to never time out.

```
esp_err_t spi_device_transmit(spi_device_handle_t handle, spi_transaction_t *trans_desc)
```

Do a SPI transaction.

Essentially does the same as spi_device_queue_trans followed by spi_device_get_trans_result. Do not use this when there is still a transaction queued that hasn't been finalized using spi_device_get_trans_result.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Parameters

- handle: Device handle obtained using spi_host_add_dev
- trans_desc: Description of transaction to execute

Structures

`struct spi_device_interface_config_t`

This is a configuration for a SPI slave device that is connected to one of the SPI buses.

Public Members

`uint8_t command_bits`

Amount of bits in command phase (0-16)

`uint8_t address_bits`

Amount of bits in address phase (0-64)

`uint8_t dummy_bits`

Amount of dummy bits to insert between address and data phase.

`uint8_t mode`

SPI mode (0-3)

`uint8_t duty_cycle_pos`

Duty cycle of positive clock, in 1/256th increments (128 = 50%/50% duty). Setting this to 0 (=not setting it) is equivalent to setting this to 128.

`uint8_t cs_ena_pretrans`

Amount of SPI bit-cycles the cs should be activated before the transmission (0-16). This only works on half-duplex transactions.

`uint8_t cs_ena_posttrans`

Amount of SPI bit-cycles the cs should stay active after the transmission (0-16)

`int clock_speed_hz`

Clock speed, in Hz.

`int spics_io_num`

CS GPIO pin for this device, or -1 if not used.

`uint32_t flags`

Bitwise OR of SPI_DEVICE_* flags.

`int queue_size`

Transaction queue size. This sets how many transactions can be ‘in the air’ (queued using `spi_device_queue_trans` but not yet finished using `spi_device_get_trans_result`) at the same time.

`transaction_cb_t pre_cb`

Callback to be called before a transmission is started. This callback is called within interrupt context.

`transaction_cb_t post_cb`

Callback to be called after a transmission has completed. This callback is called within interrupt context.

`struct spi_transaction_t`

This structure describes one SPI transaction. The descriptor should not be modified until the transaction finishes.

Public Members

`uint32_t flags`

Bitwise OR of SPI_TRANS_* flags.

uint16_t cmd

Command data, of which the length is set in the `command_bits` of `spi_device_interface_config_t`.
NOTE: this field, used to be “command” in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF 3.0.

- Example: write 0x0123 and command_bits=12 to send command 0x12, 0x3_ (in previous version, you may have to write 0x3_12).

uint64_t addr

Address data, of which the length is set in the `address_bits` of `spi_device_interface_config_t`.
NOTE: this field, used to be “address” in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF3.0.

- Example: write 0x123400 and address_bits=24 to send address of 0x12, 0x34, 0x00 (in previous version, you may have to write 0x12340000).

size_t length

Total data length, in bits.

size_t rxlength

Total data length received, should be not greater than `length` in full-duplex mode (0 defaults this to the value of `length`).

void *user

User-defined variable. Can be used to store eg transaction ID.

const void *tx_buffer

Pointer to transmit buffer, or NULL for no MOSI phase.

uint8_t tx_data[4]

If SPI_USE_TXDATA is set, data set here is sent directly from this variable.

void *rx_buffer

Pointer to receive buffer, or NULL for no MISO phase. Written by 4 bytes-unit if DMA is used.

uint8_t rx_data[4]

If SPI_USE_RXDATA is set, data is received directly to this variable.

Macros

SPI_DEVICE_TXBIT_LSBFIRST

Transmit command/address/data LSB first instead of the default MSB first.

SPI_DEVICE_RXBIT_LSBFIRST

Receive data LSB first instead of the default MSB first.

SPI_DEVICE_BIT_LSBFIRST

Transmit and receive LSB first.

SPI_DEVICE_3WIRE

Use MOSI (=spid) for both sending and receiving data.

SPI_DEVICE_POSITIVE_CS

Make CS positive during a transaction instead of negative.

SPI_DEVICE_HALFDUPLEX

Transmit data before receiving it, instead of simultaneously.

SPI_DEVICE_CLK_AS_CS

Output clock on CS line if CS is active.

SPI_TRANS_MODE_DIO

Transmit/receive data in 2-bit mode.

SPI_TRANS_MODE_QIO

Transmit/receive data in 4-bit mode.

SPI_TRANS_MODE_DIOQIO_ADDR

Also transmit address in mode selected by SPI_MODE_DIO/SPI_MODE_QIO.

SPI_TRANS_USE_RXDATA

Receive into rx_data member of *spi_transaction_t* instead into memory at rx_buffer.

SPI_TRANS_USE_TXDATA

Transmit tx_data member of *spi_transaction_t* instead of data at tx_buffer. Do not set tx_buffer when using this.

Type Definitions

```
typedef struct spi_transaction_t spi_transaction_t
typedef void (*transaction_cb_t)(spi_transaction_t *trans)
typedef struct spi_device_t *spi_device_handle_t
Handle for a device on a SPI bus.
```

2.4.14 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 querying the result using `spi_slave_get_trans_result`, or handle all requests synchronously by feeding them into `spi_slave_transmit`. The latter two functions will block until the master has initiated and finished a transaction, causing the queued data to be sent and received.
- Optional: to unload the SPI slave driver, call `spi_slave_free`.

Transaction data and master/slave length mismatches

Normally, data to be transferred to or from a device will be read from or written to a chunk of memory indicated by the `rx_buffer` and `tx_buffer` members of the transaction structure. The SPI driver may decide to use DMA for transfers, so these buffers should be allocated in DMA-capable memory using `pvPortMallocCaps(size, MALLOC_CAP_DMA)`.

The amount of data written to the buffers is limited by the `length` member of the transaction structure: the driver will never read/write more data than indicated there. The `length` cannot define the actual length of the SPI transaction; this is determined by the master as it drives the clock and CS lines. In case the length of the transmission is larger than the buffer length, only the start of the transmission will be sent and received. In case the transmission length is shorter than the buffer length, only data up to the length of the buffer will be exchanged.

Warning: Due to a design peculiarity in the ESP32, if the amount of bytes sent by the master or the length of the transmission queues in the slave driver, in bytes, is not both larger than eight and dividable by four, the SPI hardware can fail to write the last one to seven bytes to the receive buffer.

Application Example

Slave/master communication: [peripherals/spi_slave](#).

API Reference

Header File

- [driver/include/driver/spi_slave.h](#)

Functions

```
esp_err_t spi_slave_initialize(spi_host_device_t host, const spi_bus_config_t *bus_config, const
                               spi_slave_interface_config_t *slave_config, int dma_chan)
```

Initialize a SPI bus as a slave interface.

Warning For now, only supports HSPI and VSPI.

Warning If a DMA channel is selected, any transmit and receive buffer used should be allocated in DMA-capable memory.

Return

- ESP_ERR_INVALID_ARG if configuration is invalid
- ESP_ERR_INVALID_STATE if host already is in use
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

Parameters

- host: SPI peripheral to use as a SPI slave interface
- bus_config: Pointer to a *spi_bus_config_t* struct specifying how the host should be initialized
- slave_config: Pointer to a *spi_slave_interface_config_t* struct specifying the details for the slave interface
- dma_chan: Either 1 or 2. A SPI bus used by this driver must have a DMA channel associated with it. The SPI hardware has two DMA channels to share. This parameter indicates which one to use.

```
esp_err_t spi_slave_free(spi_host_device_t host)
```

Free a SPI bus claimed as a SPI slave interface.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if not all devices on the bus are freed
- ESP_OK on success

Parameters

- host: SPI peripheral to free

```
esp_err_t spi_slave_queue_trans(spi_host_device_t host, const spi_slave_transaction_t
                                *trans_desc, TickType_t ticks_to_wait)
```

Queue a SPI transaction for execution.

Queues a SPI transaction to be executed by this slave device. (The transaction queue size was specified when the slave device was initialised via `spi_slave_initialize`.) This function may block if the queue is full (depending on the `ticks_to_wait` parameter). No SPI operation is directly initiated by this function, the next queued transaction will happen when the master initiates a SPI transaction by pulling down CS and sending out clock signals.

This function hands over ownership of the buffers in `trans_desc` to the SPI slave driver; the application is not to access this memory until `spi_slave_queue_trans` is called to hand ownership back to the application.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid

- ESP_OK on success

Parameters

- host: SPI peripheral that is acting as a slave
- trans_desc: Description of transaction to execute. Not const because we may want to write status back into the transaction description.
- ticks_to_wait: Ticks to wait until there's room in the queue; use portMAX_DELAY to never time out.

```
esp_err_t spi_slave_get_trans_result(spi_host_device_t      host,          spi_slave_transaction_t
                                      **trans_desc, TickType_t ticks_to_wait)
```

Get the result of a SPI transaction queued earlier.

This routine will wait until a transaction to the given device (queued earlier with spi_slave_queue_trans) has successfully completed. It will then return the description of the completed transaction so software can inspect the result and e.g. free the memory or re-use the buffers.

It is mandatory to eventually use this function for any transaction queued by spi_slave_queue_trans.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Parameters

- host: SPI peripheral to that is acting as a slave
- trans_desc: Pointer to variable able to contain a pointer to the description of the transaction that is executed
- ticks_to_wait: Ticks to wait until there's a returned item; use portMAX_DELAY to never time out.

```
esp_err_t spi_slave_transmit(spi_host_device_t host, spi_slave_transaction_t *trans_desc, TickType_t
                             ticks_to_wait)
```

Do a SPI transaction.

Essentially does the same as spi_slave_queue_trans followed by spi_slave_get_trans_result. Do not use this when there is still a transaction queued that hasn't been finalized using spi_slave_get_trans_result.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Parameters

- host: SPI peripheral to that is acting as a slave
- trans_desc: Pointer to variable able to contain a pointer to the description of the transaction that is executed. Not const because we may want to write status back into the transaction description.
- ticks_to_wait: Ticks to wait until there's a returned item; use portMAX_DELAY to never time out.

Structures

`struct spi_slave_interface_config_t`

This is a configuration for a SPI host acting as a slave device.

Public Members

`int spics_io_num`

CS GPIO pin for this device.

`uint32_t flags`

Bitwise OR of SPI_SLAVE_* flags.

`int queue_size`

Transaction queue size. This sets how many transactions can be ‘in the air’ (queued using `spi_slave_queue_trans` but not yet finished using `spi_slave_get_trans_result`) at the same time.

`uint8_t mode`

SPI mode (0-3)

`slave_transaction_cb_t post_setup_cb`

Callback called after the SPI registers are loaded with new data.

`slave_transaction_cb_t post_trans_cb`

Callback called after a transaction is done.

`struct spi_slave_transaction_t`

This structure describes one SPI transaction

Public Members

`size_t length`

Total data length, in bits.

`const void *tx_buffer`

Pointer to transmit buffer, or NULL for no MOSI phase.

`void *rx_buffer`

Pointer to receive buffer, or NULL for no MISO phase.

`void *user`

User-defined variable. Can be used to store eg transaction ID.

Macros

`SPI_SLAVE_TXBIT_LSBFIRST`

Transmit command/address/data LSB first instead of the default MSB first.

`SPI_SLAVE_RXBIT_LSBFIRST`

Receive data LSB first instead of the default MSB first.

`SPI_SLAVE_BIT_LSBFIRST`

Transmit and receive LSB first.

Type Definitions

```
typedef struct spi_slave_transaction_t spi_slave_transaction_t  
typedef void (*slave_transaction_cb_t)(spi_slave_transaction_t *trans)
```

2.4.15 TIMER

Overview

ESP32 chip contains two hardware timer groups, each containing 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.

Application Example

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, uint16_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.

```
esp_err_t timer_set_alarm_value(timer_group_t group_num, timer_idx_t timer_num, uint64_t  
                                alarm_value)
```

Set timer alarm value.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `alarm_value`: A 64-bit value to set the alarm value.

```
esp_err_t timer_get_alarm_value(timer_group_t group_num, timer_idx_t timer_num, uint64_t  
                                *alarm_value)
```

Get timer alarm value.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- group_num: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- timer_num: Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- alarm_value: Pointer of A 64-bit value to accept the alarm value.

```
esp_err_t timer_set_alarm(timer_group_t group_num, timer_idx_t timer_num, timer_alarm_t alarm_en)
```

Get timer alarm value.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- group_num: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- timer_num: Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- alarm_en: To enable or disable timer alarm function.

```
esp_err_t timer_isr_register(timer_group_t group_num, timer_idx_t timer_num, void (*fn) void *, void *arg, int intr_alloc_flags, timer_isr_handle_t *handle)
```

register Timer interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

Note In case the this is called with the INIRAM flag, code inside the handler function can only call functions in IRAM, so it cannot call other timer APIs. Use direct register access to access timers from inside the ISR in this case.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Function pointer error.

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.

Parameters

- 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`
timer configure struct

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

uint16_t divider
    Counter clock divider
```

Macros

`TIMER_BASE_CLK`

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.hgllcnt.low

TIMER_COUNT_UP = 1

Ascending Count from Zero

TIMER_COUNT_MAX

enum timer_start_t

Decides whether timer is on or paused.

Values:

TIMER_PAUSE = 0

Pause timer counter

TIMER_START = 1

Start timer counter

enum timer_alarm_t

Decides whether to enable alarm mode.

Values:

TIMER_ALARM_DIS = 0

Disable timer alarm

TIMER_ALARM_EN = 1

Enable timer alarm

TIMER_ALARM_MAX**enum timer_intr_mode_t**

Select interrupt type if running in alarm mode.

*Values:***TIMER_INTR_LEVEL** = 0

Interrupt mode: level mode

TIMER_INTR_MAX**enum timer_autoreload_t**

Select if Alarm needs to be loaded by software or automatically reload by hardware.

*Values:***TIMER_AUTORELOAD_DIS** = 0

Disable auto-reload: hardware will not load counter value after an alarm event

TIMER_AUTORELOAD_EN = 1

Enable auto-reload: hardware will load counter value after an alarm event

TIMER_AUTORELOAD_MAX

2.4.16 Touch Sensor

Introduction

A touch-sensor system is built on a substrate which carries electrodes and relevant connections under a protective flat surface. When a user touches the surface, the capacitance variation is triggered and a binary signal is generated to indicate whether the touch is valid.

ESP32 can provide up to 10 capacitive touch pads / GPIOs. The sensing pads can be arranged in different combinations (e.g. matrix, slider), so that a larger area or more points can be detected. The touch pad sensing process is under the control of a hardware-implemented finite-state machine (FSM) which is initiated by software or a dedicated hardware timer.

Design, operation and control registers of touch sensor are discussed in [ESP32 Technical Reference Manual \(PDF\)](#). Please refer to it for additional details how this subsystem works.

Functionality Overview

Description of API is broken down into groups of functions to provide quick overview of features like:

- Initialization of touch pad driver
- Configuration of touch pad GPIO pins
- Taking measurements
- Adjusting parameters of measurements
- Filtering measurements
- Touch detection methods

- Setting up interrupts to report touch detection
- Waking up from sleep mode on interrupt

For detailed description of particular function please go to section [API Reference](#). Practical implementation of this API is covered in section [Application Examples](#).

Initialization

Touch pad driver should be initialized before use by calling function `touch_pad_init()`. This function sets several `.._DEFAULT` driver parameters listed in [API Reference](#) under “Macros”. It also clears information what pads have been touched before (if any) and disables interrupts.

If not required anymore, driver can be disabled by calling `touch_pad_deinit()`.

Configuration

Enabling of touch sensor functionality for particular GPIO is done with `touch_pad_config()`.

The function `touch_pad_set_fsm_mode()` is used to select whether touch pad measurement (operated by FSM) is started automatically by hardware timer, or by software. If software mode is selected, then use `touch_pad_sw_start()` to start of the FSM.

Touch State Measurements

The following two functions come handy to read raw or filtered measurements from the sensor:

- `touch_pad_read()`
- `touch_pad_read_filtered()`

They may be used to characterize particular touch pad design by checking the range of sensor readings when a pad is touched or released. This information can be then used to establish the touch threshold.

Note: Start and configure filter before using `touch_pad_read_filtered()` by calling specific filter functions described down below.

To see how to use both read functions check [peripherals/touch_pad_read](#) application example.

Optimization of Measurements

Touch sensor has several configurable parameters to match characteristics of particular touch pad design. For instance, to sense smaller capacity changes, it is possible to narrow the reference voltage range within which the touch pads are charged / discharged. The high and low reference voltages are set using function `touch_pad_set_voltage()`. A positive side effect, besides ability to discern smaller capacity changes, will be reduction of power consumption for low power applications. A likely negative effect will be increase of measurement noise. If dynamic 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.

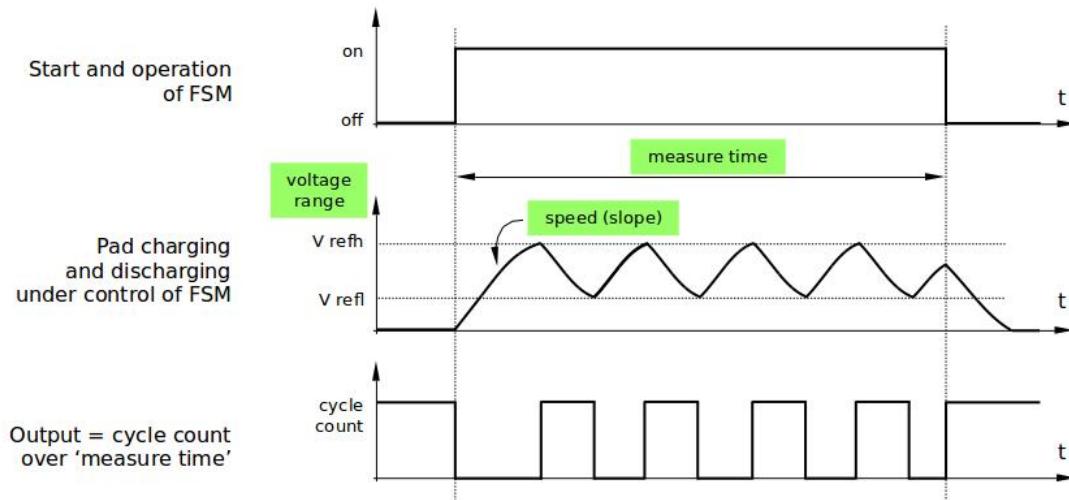


Fig. 2.1: Touch Pad - relationship between measurement parameters

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()`.

The filter type is IIR (Infinite Impulse Response) and it has configurable period that can be set with function `touch_pad_set_filter_period()`.

You can stop the filter with `touch_pad_filter_stop()`. If not required anymore, the filter may be deleted by invoking `touch_pad_filter_delete()`.

Touch Detection

Touch detection is implemented in ESP32’s hardware basing on user configured threshold and raw measurements executed by FSM. Use function `touch_pad_get_status()` to check what pads have been touched and `touch_pad_clear_status()` to clear the touch status information.

Hardware touch detection may be also wired to interrupts and this is described in next section.

If measurements are noisy and capacity changes small, then hardware touch detection may be not reliable. To resolve this issue, instead of using hardware detection / provided interrupts, implement measurement filtering and perform

touch detection in your own application. See [peripherals/touch_pad_interrupt](#) for sample implementation of both methods of touch detection.

Touch Triggered Interrupts

Before enabling an interrupt on touch detection, user should establish touch detection threshold. Use functions described above to read and display sensor measurements when pad is touched and released. Apply a filter when measurements are noisy and relative changes are small. Depending on your application and environmental conditions, test the influence of temperature and power supply voltage changes on measured values.

Once detection threshold is established, it may be set on initialization with `touch_pad_config()` or at the runtime with `touch_pad_set_thresh()`.

In next step configure how interrupts are triggered. They may be triggered below or above threshold and this is set with function `touch_pad_set_trigger_mode()`.

Finally configure and manage interrupt calls using the following functions:

- `touch_pad_isr_register()` / `touch_pad_isr_deregister()`
- `touch_pad_intr_enable()` / `touch_pad_intr_disable()`

When interrupts are operational, you can obtain information what particular pad triggered interrupt by invoking `touch_pad_get_status()` and clear pad status with `touch_pad_clear_status()`.

Note: Interrupts on touch detection operate on raw / unfiltered measurements checked against user established threshold and are implemented in hardware. Enabling software filtering API (see [Filtering of Measurements](#)) does not affect this process.

Wakeup from Sleep Mode

If touch pad interrupts are used to wakeup the chip from a sleep mode, then user can select certain configuration of pads (SET1 or both SET1 and SET2), that should be touched to trigger the interrupt and cause subsequent wakeup. To do so, use function `touch_pad_set_trigger_source()`.

Configuration of required bit patterns of pads may be managed for each ‘SET’ with:

- `touch_pad_set_group_mask()` / `touch_pad_get_group_mask()`
- `touch_pad_clear_group_mask()`

Application Examples

- Touch sensor read example: [peripherals/touch_pad_read](#).
- Touch sensor interrupt example: [peripherals/touch_pad_interrupt](#).

API Reference

Header File

- [driver/include/driver/touch_pad.h](#)

Functions

`esp_err_t touch_pad_init()`

Initialize touch module.

Return

- ESP_OK Success
- ESP_FAIL Touch pad init error

`esp_err_t touch_pad_deinit()`

Un-install touch pad driver.

Return

- ESP_OK Success
- ESP_FAIL Touch pad driver not initialized

`esp_err_t touch_pad_config(touch_pad_t touch_num, uint16_t threshold)`

Configure touch pad interrupt threshold.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG if argument wrong
- ESP_FAIL if touch pad not initialized

Parameters

- `touch_num`: touch pad index
- `threshold`: interrupt threshold,

`esp_err_t touch_pad_read(touch_pad_t touch_num, uint16_t *touch_value)`

get touch sensor counter value. Each touch sensor has a counter to count the number of charge/discharge cycles. When the pad is not ‘touched’, we can get a number of the counter. When the pad is ‘touched’, the value in counter will get smaller because of the larger equivalent capacitance. User can use this function to determine the interrupt trigger threshold.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch pad error
- ESP_FAIL Touch pad not initialized

Parameters

- `touch_num`: touch pad index
- `touch_value`: pointer to accept touch sensor value

`esp_err_t touch_pad_read_filtered(touch_pad_t touch_num, uint16_t *touch_value)`

get filtered touch sensor counter value by IIR filter.

Note `touch_pad_filter_start` has to be called before calling `touch_pad_read_filtered`. This function can be called from ISR

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch pad error
- ESP_FAIL Touch pad not initialized

Parameters

- touch_num: touch pad index
- touch_value: pointer to accept touch sensor value

```
esp_err_t touch_pad_isr_handler_register(void (*fn) void *  
, void *arg, int unused, intr_handler_t *handle_unusedRegister touch-pad ISR.,
```

Note Deprecated function, users should replace this with touch_pad_isr_register, because RTC modules share a same interrupt index.

Return

- ESP_OK Success ;
- ESP_ERR_INVALID_ARG GPIO error

Parameters

- fn: Pointer to ISR handler
- arg: Parameter for ISR
- unused: Reserved, not used
- handle_unused: Reserved, not used

```
esp_err_t touch_pad_isr_register(intr_handler_t fn, void *arg)
```

Register touch-pad ISR. The handler will be attached to the same CPU core that this function is running on.

Return

- ESP_OK Success ;
- ESP_ERR_INVALID_ARG GPIO error

Parameters

- fn: Pointer to ISR handler
- arg: Parameter for ISR

```
esp_err_t touch_pad_isr_deregister(void (*fn) void *
```

, void *arg)Deregister the handler previously registered using touch_pad_isr_handler_register.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if a handler matching both fn and arg isn't registered

Parameters

- fn: handler function to call (as passed to touch_pad_isr_handler_register)
- arg: argument of the handler (as passed to touch_pad_isr_handler_register)

`esp_err_t touch_pad_set_meas_time(uint16_t sleep_cycle, uint16_t meas_cycle)`
Set touch sensor measurement and sleep time.

Return

- ESP_OK on success

Parameters

- `sleep_cycle`: The touch sensor will sleep after each measurement. `sleep_cycle` decide the interval between each measurement. $t_{sleep} = sleep_cycle / (RTC_SLOW_CLK \text{ frequency})$. The approximate frequency value of RTC_SLOW_CLK can be obtained using `rtc_clk_slow_freq_get_hz` function.
- `meas_cycle`: The duration of the touch sensor measurement. $t_{meas} = meas_cycle / 8M$, the maximum measure time is `0xffff / 8M = 8.19 ms`

`esp_err_t touch_pad_get_meas_time(uint16_t *sleep_cycle, uint16_t *meas_cycle)`
Get touch sensor measurement and sleep time.

Return

- ESP_OK on success

Parameters

- `sleep_cycle`: Pointer to accept sleep cycle number
- `meas_cycle`: Pointer to accept measurement cycle count.

`esp_err_t touch_pad_set_voltage(touch_high_volt_t refh, touch_low_volt_t refl, touch_volt_atten_t atten)`

Set touch sensor reference voltage, if the voltage gap between high and low reference voltage get less, the charging and discharging time would be faster, accordingly, the counter value would be larger. In the case of detecting very slight change of capacitance, we can narrow down the gap so as to increase the sensitivity. On the other hand, narrow voltage gap would also introduce more noise, but we can use a software filter to pre-process the counter value.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

Parameters

- `refh`: the value of DREFH
- `refl`: the value of DREFL
- `atten`: the attenuation on DREFH

`esp_err_t touch_pad_get_voltage(touch_high_volt_t *refh, touch_low_volt_t *refl, touch_volt_atten_t *atten)`

Get touch sensor reference voltage,.

Return

- ESP_OK on success

Parameters

- `refh`: pointer to accept DREFH value

- `refl`: pointer to accept DREFL value
- `atten`: pointer to accept the attenuation on DREFH

`esp_err_t touch_pad_set_cnt_mode (touch_pad_t touch_num, touch_cnt_slope_t slope, touch_tie_opt_t opt)`

Set touch sensor charge/discharge speed for each pad. If the slope is 0, the counter would always be zero. If the slope is 1, the charging and discharging would be slow, accordingly, the counter value would be small. If the slope is set 7, which is the maximum value, the charging and discharging would be fast, accordingly, the counter value would be larger.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

Parameters

- `touch_num`: touch pad index
- `slope`: touch pad charge/discharge speed
- `opt`: the initial voltage

`esp_err_t touch_pad_get_cnt_mode (touch_pad_t touch_num, touch_cnt_slope_t *slope, touch_tie_opt_t *opt)`

Get touch sensor charge/discharge speed for each pad.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

Parameters

- `touch_num`: touch pad index
- `slope`: pointer to accept touch pad charge/discharge slope
- `opt`: pointer to accept the initial voltage

`esp_err_t touch_pad_io_init (touch_pad_t touch_num)`

Initialize touch pad GPIO.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

Parameters

- `touch_num`: touch pad index

`esp_err_t touch_pad_set_fsm_mode (touch_fsm_mode_t mode)`

Set touch sensor FSM mode, the test action can be triggered by the timer, as well as by the software.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

Parameters

- mode: FSM mode

`esp_err_t touch_pad_get_fsm_mode (touch_fsm_mode_t *mode)`

Get touch sensor FSM mode.

Return

- ESP_OK on success

Parameters

- mode: pointer to accept FSM mode

`esp_err_t touch_pad_sw_start ()`

Trigger a touch sensor measurement, only support in SW mode of FSM.

Return

- ESP_OK on success

`esp_err_t touch_pad_set_thresh (touch_pad_t touch_num, uint16_t threshold)`

Set touch sensor interrupt threshold.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

Parameters

- touch_num: touch pad index
- threshold: threshold of touchpad count, refer to touch_pad_set_trigger_mode to see how to set trigger mode.

`esp_err_t touch_pad_get_thresh (touch_pad_t touch_num, uint16_t *threshold)`

Get touch sensor interrupt threshold.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

Parameters

- touch_num: touch pad index
- threshold: pointer to accept threshold

`esp_err_t touch_pad_set_trigger_mode (touch_trigger_mode_t mode)`

Set touch sensor interrupt trigger mode. Interrupt can be triggered either when counter result is less than threshold or when counter result is more than threshold.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

Parameters

- mode: touch sensor interrupt trigger mode

`esp_err_t touch_pad_get_trigger_mode (touch_trigger_mode_t *mode)`

Get touch sensor interrupt trigger mode.

Return

- ESP_OK on success

Parameters

- mode: pointer to accept touch sensor interrupt trigger mode

`esp_err_t touch_pad_set_trigger_source (touch_trigger_src_t src)`

Set touch sensor interrupt trigger source. There are two sets of touch signals. Set1 and set2 can be mapped to several touch signals. Either set will be triggered if at least one of its touch signal is ‘touched’. The interrupt can be configured to be generated if set1 is triggered, or only if both sets are triggered.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

Parameters

- src: touch sensor interrupt trigger source

`esp_err_t touch_pad_get_trigger_source (touch_trigger_src_t *src)`

Get touch sensor interrupt trigger source.

Return

- ESP_OK on success

Parameters

- src: pointer to accept touch sensor interrupt trigger source

`esp_err_t touch_pad_set_group_mask (uint16_t set1_mask, uint16_t set2_mask, uint16_t en_mask)`

Set touch sensor group mask. Touch pad module has two sets of signals, ‘Touched’ signal is triggered only if at least one of touch pad in this group is “touched”. This function will set the register bits according to the given bitmask.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

Parameters

- set1_mask: bitmask of touch sensor signal group1, it’s a 10-bit value
- set2_mask: bitmask of touch sensor signal group2, it’s a 10-bit value
- en_mask: bitmask of touch sensor work enable, it’s a 10-bit value

`esp_err_t touch_pad_get_group_mask (uint16_t *set1_mask, uint16_t *set2_mask, uint16_t *en_mask)`

Get touch sensor group mask.

Return

- ESP_OK on success

Parameters

- set1_mask: pointer to accept bitmask of touch sensor signal group1, it's a 10-bit value
- set2_mask: pointer to accept bitmask of touch sensor signal group2, it's a 10-bit value
- en_mask: pointer to accept bitmask of touch sensor work enable, it's a 10-bit value

`esp_err_t touch_pad_clear_group_mask (uint16_t set1_mask, uint16_t set2_mask, uint16_t en_mask)`

Clear touch sensor group mask. Touch pad module has two sets of signals, Interrupt is triggered only if at least one of touch pad in this group is “touched”. This function will clear the register bits according to the given bitmask.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

Parameters

- set1_mask: bitmask touch sensor signal group1, it's a 10-bit value
- set2_mask: bitmask touch sensor signal group2, it's a 10-bit value
- en_mask: bitmask of touch sensor work enable, it's a 10-bit value

`esp_err_t touch_pad_clear_status ()`

To clear the touch status register, usually use this function in touch ISR to clear status.

Return

- ESP_OK on success

`uint32_t touch_pad_get_status ()`

Get the touch sensor status, usually used in ISR to decide which pads are ‘touched’.

Return

- touch status

`esp_err_t touch_pad_intr_enable ()`

To enable touch pad interrupt.

Return

- ESP_OK on success

`esp_err_t touch_pad_intr_disable ()`

To disable touch pad interrupt.

Return

- ESP_OK on success

`esp_err_t touch_pad_set_filter_period (uint32_t new_period_ms)`

set touch pad filter calibration period, in ms. Need to call touch_pad_filter_start before all touch filter APIs

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE driver state error
- ESP_ERR_INVALID_ARG parameter error

Parameters

- new_period_ms: filter period, in ms

esp_err_t **touch_pad_get_filter_period** (uint32_t **p_period_ms*)

get touch pad filter calibration period, in ms Need to call touch_pad_filter_start before all touch filter APIs

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE driver state error
- ESP_ERR_INVALID_ARG parameter error

Parameters

- p_period_ms: pointer to accept period

esp_err_t **touch_pad_filter_start** (uint32_t *filter_period_ms*)

start touch pad filter function This API will start a filter to process the noise in order to prevent false triggering when detecting slight change of capacitance. Need to call touch_pad_filter_start before all touch filter APIs

If filter is not initialized, this API will initialize the filter with given period. If filter is already initialized, this API will update the filter period.

Note This filter uses FreeRTOS timer, which is dispatched from a task with priority 1 by default on CPU 0. So if some application task with higher priority takes a lot of CPU0 time, then the quality of data obtained from this filter will be affected. You can adjust FreeRTOS timer task priority in menuconfig.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG parameter error
- ESP_ERR_NO_MEM No memory for driver
- ESP_ERR_INVALID_STATE driver state error

Parameters

- filter_period_ms: filter calibration period, in ms

esp_err_t **touch_pad_filter_stop** ()

stop touch pad filter function Need to call touch_pad_filter_start before all touch filter APIs

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE driver state error

esp_err_t **touch_pad_filter_delete** ()

delete touch pad filter driver and release the memory Need to call touch_pad_filter_start before all touch filter APIs

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE driver state error

Macros**TOUCH_PAD_SLEEP_CYCLE_DEFAULT**

The timer frequency is RTC_SLOW_CLK (can be 150k or 32k depending on the options), max value is 0xfffff

TOUCH_PAD_MEASURE_CYCLE_DEFAULT

The timer frequency is 8Mhz, the max value is 0xfffff

TOUCH_FSM_MODE_DEFAULT

The touch FSM may be started by the software or timer

TOUCH_TRIGGER_MODE_DEFAULT

Interrupts can be triggered if sensor value gets below or above threshold

TOUCH_TRIGGER_SOURCE_DEFAULT

The wakeup trigger source can be SET1 or both SET1 and SET2

TOUCH_PAD_BIT_MASK_MAX**Type Definitions**

```
typedef intr_handle_t touch_isr_handle_t
```

Enumerations**enum touch_pad_t**

Values:

TOUCH_PAD_NUM0 = 0

Touch pad channel 0 is GPIO4

TOUCH_PAD_NUM1

Touch pad channel 1 is GPIO0

TOUCH_PAD_NUM2

Touch pad channel 2 is GPIO2

TOUCH_PAD_NUM3

Touch pad channel 3 is GPIO15

TOUCH_PAD_NUM4

Touch pad channel 4 is GPIO13

TOUCH_PAD_NUM5

Touch pad channel 5 is GPIO12

TOUCH_PAD_NUM6

Touch pad channel 6 is GPIO14

TOUCH_PAD_NUM7

Touch pad channel 7 is GPIO27

TOUCH_PAD_NUM8

Touch pad channel 8 is GPIO32

TOUCH_PAD_NUM9

Touch pad channel 9 is GPIO33

TOUCH_PAD_MAX

enum touch_high_volt_t

Values:

TOUCH_HVOLT_KEEP = -1

Touch sensor high reference voltage, no change

TOUCH_HVOLT_2V4 = 0

Touch sensor high reference voltage, 2.4V

TOUCH_HVOLT_2V5

Touch sensor high reference voltage, 2.5V

TOUCH_HVOLT_2V6

Touch sensor high reference voltage, 2.6V

TOUCH_HVOLT_2V7

Touch sensor high reference voltage, 2.7V

TOUCH_HVOLT_MAX

enum touch_low_volt_t

Values:

TOUCH_LVOLT_KEEP = -1

Touch sensor low reference voltage, no change

TOUCH_LVOLT_0V5 = 0

Touch sensor low reference voltage, 0.5V

TOUCH_LVOLT_0V6

Touch sensor low reference voltage, 0.6V

TOUCH_LVOLT_0V7

Touch sensor low reference voltage, 0.7V

TOUCH_LVOLT_0V8

Touch sensor low reference voltage, 0.8V

TOUCH_LVOLT_MAX

enum touch_volt_atten_t

Values:

TOUCH_HVOLT_ATTEN_KEEP = -1

Touch sensor high reference voltage attenuation, no change

TOUCH_HVOLT_ATTEN_1V5 = 0

Touch sensor high reference voltage attenuation, 1.5V attenuation

TOUCH_HVOLT_ATTEN_1V

Touch sensor high reference voltage attenuation, 1.0V attenuation

TOUCH_HVOLT_ATTEN_0V5

Touch sensor high reference voltage attenuation, 0.5V attenuation

TOUCH_HVOLT_ATTEN_0V

Touch sensor high reference voltage attenuation, 0V attenuation

TOUCH_HVOLT_ATTEN_MAX**enum touch_cnt_slope_t**

Values:

TOUCH_PAD_SLOPE_0 = 0

Touch sensor charge / discharge speed, always zero

TOUCH_PAD_SLOPE_1 = 1

Touch sensor charge / discharge speed, slowest

TOUCH_PAD_SLOPE_2 = 2

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_3 = 3

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_4 = 4

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_5 = 5

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_6 = 6

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_7 = 7

Touch sensor charge / discharge speed, fast

TOUCH_PAD_SLOPE_MAX**enum touch_trigger_mode_t**

Values:

TOUCH_TRIGGER_BELOW = 0

Touch interrupt will happen if counter value is less than threshold.

TOUCH_TRIGGER_ABOVE = 1

Touch interrupt will happen if counter value is larger than threshold.

TOUCH_TRIGGER_MAX**enum touch_trigger_src_t**

Values:

TOUCH_TRIGGER_SOURCE_BOTH = 0

wakeup interrupt is generated if both SET1 and SET2 are “touched”

TOUCH_TRIGGER_SOURCE_SET1 = 1

wakeup interrupt is generated if SET1 is “touched”

TOUCH_TRIGGER_SOURCE_MAX**enum touch_tie_opt_t**

Values:

TOUCH_PAD_TIE_OPT_LOW = 0

Initial level of charging voltage, low level

TOUCH_PAD_TIE_OPT_HIGH = 1

Initial level of charging voltage, high level

TOUCH_PAD_TIE_OPT_MAX

```
enum touch_fsm_mode_t
```

Values:

```
TOUCH_FSM_MODE_TIMER = 0
```

To start touch FSM by timer

```
TOUCH_FSM_MODE_SW
```

To start touch FSM by software trigger

```
TOUCH_FSM_MODE_MAX
```

GPIO Lookup Macros

Some useful macros can be used to specified the GPIO number of a touchpad channel, or vice versa. e.g.

1. TOUCH_PAD_NUM5_GPIO_NUM is the GPIO number of channel 5 (12);
2. TOUCH_PAD_GPIO4_CHANNEL is the channel number of GPIO 4 (channel 0).

Header File

- [soc/esp32/include/soc/touch_channel.h](#)

Macros

```
TOUCH_PAD_GPIO4_CHANNEL
```

```
TOUCH_PAD_NUM0_GPIO_NUM
```

```
TOUCH_PAD_GPIO0_CHANNEL
```

```
TOUCH_PAD_NUM1_GPIO_NUM
```

```
TOUCH_PAD_GPIO2_CHANNEL
```

```
TOUCH_PAD_NUM2_GPIO_NUM
```

```
TOUCH_PAD_GPIO15_CHANNEL
```

```
TOUCH_PAD_NUM3_GPIO_NUM
```

```
TOUCH_PAD_GPIO13_CHANNEL
```

```
TOUCH_PAD_NUM4_GPIO_NUM
```

```
TOUCH_PAD_GPIO12_CHANNEL
```

```
TOUCH_PAD_NUM5_GPIO_NUM
```

```
TOUCH_PAD_GPIO14_CHANNEL
```

```
TOUCH_PAD_NUM6_GPIO_NUM
```

```
TOUCH_PAD_GPIO27_CHANNEL
```

```
TOUCH_PAD_NUM7_GPIO_NUM
```

```
TOUCH_PAD_GPIO33_CHANNEL
```

```
TOUCH_PAD_NUM8_GPIO_NUM
```

```
TOUCH_PAD_GPIO32_CHANNEL
```

`TOUCH_PAD_NUM9_GPIO_NUM`

2.4.17 UART

Overview

Instructions

Application Example

Configure uart settings and install uart driver to read/write using UART0 and UART1 interfaces: [peripherals/uart](#).

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*)
Set 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.

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 bit-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, combine 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_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 status that to be cleared. enable_mask should be chosen 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. enable_mask should be chosen 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. disable_mask should be chosen 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(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_enable_tx_intr(uart_port_t uart_num, int enable, int thresh)`
Enable UART TX 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
- 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.

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, if set to `UART_PIN_NO_CHANGE`, use the current pin.
- `rx_io_num`: UART RX pin GPIO number, if set to `UART_PIN_NO_CHANGE`, use the current pin.
- `rts_io_num`: UART RTS pin GPIO number, if set to `UART_PIN_NO_CHANGE`, use the current pin.
- `cts_io_num`: UART CTS pin GPIO number, if set to `UART_PIN_NO_CHANGE`, use the current pin.

`esp_err_t uart_set_rts (uart_port_t uart_num, int level)`

UART set RTS level (before inverse) UART rx hardware flow control should not be set.

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)`

UART set DTR level (before inverse)

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_param_config(uart_port_t uart_num, const uart_config_t *uart_config)`

UART parameter configure.

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)`

UART interrupt configure.

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 tx_buffer_size should be 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, rx_buffer_size should be greater than UART_FIFO_LEN.
- 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..

Parameters

- 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 UART TX FIFO 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 the space in TX FIFO, just fill the 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 data that 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 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 tx_buffer_size > 0, this function will return after copying all the data to tx ringbuffer, then, UART ISR will move data from ring buffer to TX FIFO gradually.

Return

- (-1) Parameter error
- OTHERS(>=0) The number of data that 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 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 send out, send a break signal.

Otherwise, if tx_buffer_size > 0, this function will return after copying all the data to tx ringbuffer, then, UART ISR will move data from ring buffer to TX FIFO gradually. After all data send out, send a break signal.

Return

- (-1) Parameter error
- OTHERS(>=0) The number of data that pushed to the TX FIFO

Parameters

- uart_num: UART_NUM_0, UART_NUM_1 or UART_NUM_2
- src: data buffer address
- size: data length to send
- brk_len: break signal length (unit: time of one data 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 return a char data 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)
```

UART ring buffer flush.

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 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.

`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.
- post_idle: idle time after the last pattern character, 24bit value, unit is APB(80Mhz) clock cycle.
- pre_idle: idle time before the first pattern character, 24bit value, unit is APB(80Mhz) clock cycle.

Structures

struct uart_config_t
UART configuration parameters for uart_param_config function.

Public Members

int **baud_rate**
UART baudrate

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

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 ringbuffer.

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

Some useful macros can be used to specified 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 specify the channel of a GPIO num. e.g.

1. `UART_NUM_2_TXD_DIRECT_GPIO_NUM` is the GPIO number of UART channel 2 TXD pin (17);
2. `UART_GPIO19_DIRECT_CHANNEL` is the UART channel number of GPIO 19 (channel 0);
3. `UART_CTS_GPIO19_DIRECT_CHANNEL` is the UART channel number of GPIO 19, and GPIO 19 must be a CTS pin (channel 0).

Header File

- `soc/esp32/include/soc/uart_channel.h`

Macros

UART_GPIO1_DIRECT_CHANNEL
UART_NUM_0_TXD_DIRECT_GPIO_NUM
UART_GPIO3_DIRECT_CHANNEL
UART_NUM_0_RXD_DIRECT_GPIO_NUM
UART_GPIO19_DIRECT_CHANNEL
UART_NUM_0_CTS_DIRECT_GPIO_NUM
UART_GPIO22_DIRECT_CHANNEL

UART_NUM_0_RTS_DIRECT_GPIO_NUM
UART_TXD_GPIO1_DIRECT_CHANNEL
UART_RXD_GPIO3_DIRECT_CHANNEL
UART_CTS_GPIO19_DIRECT_CHANNEL
UART_RTS_GPIO22_DIRECT_CHANNEL
UART_GPIO10_DIRECT_CHANNEL
UART_NUM_1_TXD_DIRECT_GPIO_NUM
UART_GPIO9_DIRECT_CHANNEL
UART_NUM_1_RXD_DIRECT_GPIO_NUM
UART_GPIO6_DIRECT_CHANNEL
UART_NUM_1_CTS_DIRECT_GPIO_NUM
UART_GPIO11_DIRECT_CHANNEL
UART_NUM_1_RTS_DIRECT_GPIO_NUM
UART_TXD_GPIO10_DIRECT_CHANNEL
UART_RXD_GPIO9_DIRECT_CHANNEL
UART_CTS_GPIO6_DIRECT_CHANNEL
UART_RTS_GPIO11_DIRECT_CHANNEL
UART_GPIO17_DIRECT_CHANNEL
UART_NUM_2_TXD_DIRECT_GPIO_NUM
UART_GPIO16_DIRECT_CHANNEL
UART_NUM_2_RXD_DIRECT_GPIO_NUM
UART_GPIO8_DIRECT_CHANNEL
UART_NUM_2_CTS_DIRECT_GPIO_NUM
UART_GPIO7_DIRECT_CHANNEL
UART_NUM_2_RTS_DIRECT_GPIO_NUM
UART_TXD_GPIO17_DIRECT_CHANNEL
UART_RXD_GPIO16_DIRECT_CHANNEL
UART_CTS_GPIO8_DIRECT_CHANNEL
UART_RTS_GPIO7_DIRECT_CHANNEL

Example code for this API section is provided in [peripherals](#) directory of ESP-IDF examples.

2.5 Protocols API

2.5.1 mDNS Service

Overview

mDNS is a multicast UDP service that is used to provide local network service and host discovery.

mDNS is installed by default on most operating systems or is available as separate package. On Mac OS it is installed by default and is called Bonjour. Apple releases an installer for Windows that can be found on [Apple's support page](#). On Linux, mDNS is provided by `avahi` and is usually installed by default.

mDNS Properties

- `hostname`: the hostname that the device will respond to. If not set, the `hostname` will be read from the interface. Example: `my-esp32` will resolve to `my-esp32.local`
- `default_instance`: friendly name for your device, like Jhon's ESP32 Thing. If not set, `hostname` will be used.

Example method to start mDNS for the STA interface and set `hostname` and `default_instance`:

```
mdns_server_t * mdns = NULL;

void start_mdns_service()
{
    //initialize mDNS service on STA interface
    esp_err_t err = mdns_init(TCPIP_ADAPTER_IF_STA, &mdns);
    if (err) {
        printf("MDNS Init failed: %d\n", err);
        return;
    }

    //set hostname
    mdns_set_hostname(mdns, "my-esp32");
    //set default instance
    mdns_set_instance(mdns, "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.

- `service`: (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
- `instance`: friendly name for your service, like Jhon's ESP32 Web Server. If not defined, `default_instance` will be used.
- `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(mdns, "_http", "_tcp", 80);
    mdns_service_add(mdns, "_arduino", "_tcp", 3232);
    mdns_service_add(mdns, "_myservice", "_udp", 1234);

    //NOTE: services must be added before their properties can be set
    //use custom instance for the web server
    mdns_service_instance_set(mdns, "_http", "_tcp", "Jhon's ESP32 Web Server");

    const char * arduTxtData[4] = {
        "board=esp32",
        "tcp_check=no",
        "ssh_upload=no",
        "auth_upload=no"
    };
    //set txt data for service (will free and replace current data)
    mdns_service_txt_set(mdns, "_arduino", "_tcp", 4, arduTxtData);

    //change service port
    mdns_service_port_set(mdns, "_myservice", "_udp", 4321);
}
```

mDNS Query

mDNS provides methods for browsing for services and resolving host's IP/IPv6 addresses.

Results are returned as a linked list of `mdns_result_t` objects. If the result is from host query, it will contain only `addr` and `addrv6` if found. Service queries will populate all fields in a result that were found.

Example method to resolve host IPs:

```
void resolve_mdns_host(const char * hostname)
{
    printf("mDNS Host Lookup: %s.local\n", hostname);
    //run search for 1000 ms
    if (mdns_query(mdns, hostname, NULL, 1000)) {
        //results were found
        const mdns_result_t * results = mdns_result_get(mdns, 0);
        //iterate through all results
        size_t i = 1;
        while(results) {
            //print result information
            printf(" %u: IP:" IPSTR ", IPv6:" IPV6STR "\n", i++
                   IP2STR(&results->addr), IPV62STR(results->addrv6));
            //load next result. Will be NULL if this was the last one
            results = results->next;
        }
        //free the results from memory
        mdns_result_free(mdns);
    } else {
        //host was not found
        printf(" Host Not Found\n");
    }
}
```

Example method to resolve local services:

```
void find_mdns_service(const char * service, const char * proto)
{
    printf("mDNS Service Lookup: %s.%s\n", service, proto);
    //run search for 1000 ms
    if (mdns_query(mdns, service, proto, 1000)) {
        //results were found
        const mdns_result_t * results = mdns_result_get(mdns, 0);
        //iterate through all results
        size_t i = 1;
        while(results) {
            //print result information
            printf(" %u: hostname:%s, instance:\"%s\", IP:" IPSTR ", IPv6:" IPV6STR
            ↵", port:%u, txt:%s\n", i++,
                (results->host)?results->host:"NULL", (results->instance)?results->
            ↵instance:"NULL",
                IP2STR(&results->addr), IPV62STR(results->addrv6),
                results->port, (results->txt)?results->txt:"\r");
            //load next result. Will be NULL if this was the last one
            results = results->next;
        }
        //free the results from memory
        mdns_result_free(mdns);
    } else {
        //service was not found
        printf(" Service Not Found\n");
    }
}
```

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 (tcpip_adapter_if_t tcpip_if, mdns_server_t **server)`
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`

Parameters

- `tcpip_if`: Interface that the server will listen on
- `server`: Server pointer to populate on success

`void mdns_free (mdns_server_t *server)`

Stop and free mDNS server.

Parameters

- `server`: mDNS Server to free

`esp_err_t mdns_set_hostname (mdns_server_t *server, const char *hostname)`

Set the hostname for mDNS server.

Return

- `ESP_OK` success
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_ERR_NO_MEM` memory error

Parameters

- `server`: mDNS Server
- `hostname`: Hostname to set

`esp_err_t mdns_set_instance (mdns_server_t *server, const char *instance)`

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

- server: mDNS Server
- instance: Instance name to set

```
esp_err_t mdns_service_add(mdns_server_t *server, const char *service, const char *proto, uint16_t  
                           port)
```

Add service to mDNS server.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM memory error

Parameters

- server: mDNS Server
- service: service type (_http, _ftp, etc)
- proto: service protocol (_tcp, _udp)
- port: service port

```
esp_err_t mdns_service_remove(mdns_server_t *server, const char *service, 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

- server: mDNS Server
- service: service type (_http, _ftp, etc)
- proto: service protocol (_tcp, _udp)

```
esp_err_t mdns_service_instance_set(mdns_server_t *server, const char *service, const char  
                                    *proto, const char *instance)
```

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

- server: mDNS Server
- service: service type (_http, _ftp, etc)
- proto: service protocol (_tcp, _udp)
- instance: instance name to set

```
esp_err_t mdns_service_txt_set (mdns_server_t *server, const char *service, const char *proto,  
                                uint8_t num_items, const char **txt)
```

Set TXT data 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

- server: mDNS Server
- service: service type (_http, _ftp, etc)
- proto: service protocol (_tcp, _udp)
- num_items: number of items in TXT data
- txt: string array of TXT data (eg. {"var=val","other=2"})

```
esp_err_t mdns_service_port_set (mdns_server_t *server, const char *service, 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

- server: mDNS Server
- service: service type (_http, _ftp, etc)
- proto: service protocol (_tcp, _udp)
- port: service port

```
esp_err_t mdns_service_remove_all (mdns_server_t *server)
```

Remove and free all services from mDNS server.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- server: mDNS Server

`size_t mdns_query (mdns_server_t *server, const char *service, const char *proto, uint32_t timeout)`
Query mDNS for host or service.

Return the number of results found

Parameters

- server: mDNS Server
- service: service type or host name
- proto: service protocol or NULL if searching for host
- timeout: time to wait for answers. If 0, mdns_query_end MUST be called to end the search

`size_t mdns_query_end (mdns_server_t *server)`
Stop mDNS Query started with timeout = 0.

Return the number of results found

Parameters

- server: mDNS Server

`size_t mdns_result_get_count (mdns_server_t *server)`
get the number of results currently in memory

Return the number of results

Parameters

- server: mDNS Server

`const mdns_result_t *mdns_result_get (mdns_server_t *server, size_t num)`
Get mDNS Search result with given index.

Return the result or NULL if error

Parameters

- server: mDNS Server
- num: the index of the result

`esp_err_t mdns_result_free (mdns_server_t *server)`
Remove and free all search results from mDNS server.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- server: mDNS Server

Structures

```
struct mdns_result_s
    mDNS query result structure
```

Public Members

```
const char *host
    hostname

const char *instance
    instance

const char *txt
    txt data

uint16_t priority
    service priority

uint16_t weight
    service weight

uint16_t port
    service port

struct ip4_addr addr
    ip4 address

struct ip6_addr addrv6
    ip6 address

const struct mdns_result_s *next
    next result, or NULL for the last result in the list
```

Type Definitions

```
typedef struct mdns_server_s mdns_server_t
typedef struct mdns_result_s mdns_result_t
    mDNS query result structure
```

Example code for this API section is provided in [protocols](#) directory of ESP-IDF examples.

2.6 Storage API

2.6.1 SPI Flash APIs

Overview

The `spi_flash` component contains APIs related to reading, writing, erasing, memory mapping data in the external SPI flash. It also has higher-level APIs which work with partitions defined in the [*partition table*](#).

Note that all the functionality is limited to the “main” SPI flash chip, the same SPI flash chip from which program runs. For `spi_flash_*` functions, this is a software limitation. The underlying ROM functions which work with SPI flash do not have provisions for working with flash chips attached to SPI peripherals other than SPI0.

SPI flash access APIs

This is the set of APIs for working with data in flash:

- `spi_flash_read` used to read data from flash to RAM
- `spi_flash_write` used to write data from RAM to flash
- `spi_flash_erase_sector` used to erase individual sectors of flash
- `spi_flash_erase_range` used to erase range of addresses in flash
- `spi_flash_get_chip_size` returns flash chip size, in bytes, as configured in menuconfig

Generally, try to avoid using the raw SPI flash functions in favour of partition-specific functions.

SPI Flash Size

The SPI flash size is configured by writing a field in the software bootloader image header, flashed at offset 0x1000.

By default, the SPI flash size is detected by `esptool.py` when this bootloader is written to flash, and the header is updated with the correct size. Alternatively, it is possible to generate a fixed flash size by disabling detection in `make menuconfig` (under Serial Flasher Config).

If it is necessary to override the configured flash size at runtime, it is possible to set the `chip_size` member of `g_rom_flashchip` structure. This size is used by `spi_flash_*` functions (in both software & ROM) for bounds checking.

Concurrency Constraints

Because the SPI flash is also used for firmware execution (via the instruction & data caches), these caches must be disabled while reading/writing/erasing. This means that both CPUs must be running code from IRAM and only reading data from DRAM while flash write operations occur.

Refer to the [application memory layout](#) documentation for an explanation of the differences between IRAM, DRAM and flash cache.

To avoid reading flash cache accidentally, when one CPU commences a flash write or erase operation the other CPU is put into a blocked state and all non-IRAM-safe interrupts are disabled on both CPUs, until the flash operation completes.

IRAM-Safe Interrupt Handlers

If you have an interrupt handler that you want to execute even when a flash operation is in progress (for example, for low latency operations), set the `ESP_INTR_FLAG_IRAM` flag when the [interrupt handler is registered](#).

You must ensure all data and functions accessed by these interrupt handlers are located in IRAM or DRAM. This includes any functions that the handler calls.

Use the `IRAM_ATTR` attribute for functions:

```
#include "esp_attr.h"

void IRAM_ATTR gpio_isr_handler(void* arg)
{
    // ...
}
```

Use the DRAM_ATTR and DRAM_STR attributes for constant data:

```
void IRAM_ATTR gpio_isr_handler(void* arg)
{
    const static DRAM_ATTR uint8_t INDEX_DATA[] = { 45, 33, 12, 0 };
    const static char *MSG = DRAM_STR("I am a string stored in RAM");
}
```

Note that knowing which data should be marked with DRAM_ATTR can be hard, the compiler will sometimes recognise that a variable or expression is constant (even if it is not marked `const`) and optimise it into flash, unless it is marked with DRAM_ATTR.

If a function or symbol is not correctly put into IRAM/DRAM and the interrupt handler reads from the flash cache during a flash operation, it will cause a crash due to Illegal Instruction exception (for code which should be in IRAM) or garbage data to be read (for constant data which should be in DRAM).

Partition table APIs

ESP-IDF projects use a partition table to maintain information about various regions of SPI flash memory (bootloader, various application binaries, data, filesystems). More information about partition tables can be found [here](#).

This component provides APIs to enumerate partitions found in the partition table and perform operations on them. These functions are declared in `esp_partition.h`:

- `esp_partition_find` used to search partition table for entries with specific type, returns an opaque iterator
- `esp_partition_get` returns a structure describing the partition, for the given iterator
- `esp_partition_next` advances iterator to the next partition found
- `esp_partition_iterator_release` releases iterator returned by `esp_partition_find`
- `esp_partition_find_first` is a convenience function which returns structure describing the first partition found by `esp_partition_find`
- `esp_partition_read`, `esp_partition_write`, `esp_partition_erase_range` are equivalent to `spi_flash_read`, `spi_flash_write`, `spi_flash_erase_range`, but operate within partition boundaries

Most application code should use `esp_partition_*` APIs instead of lower level `spi_flash_*` APIs. Partition APIs do bounds checking and calculate correct offsets in flash based on data stored in partition table.

SPI Flash Encryption

It is possible to encrypt SPI flash contents, and have it transparently decrypted by hardware.

Refer to the [Flash Encryption documentation](#) for more details.

Memory mapping APIs

ESP32 features memory hardware which allows regions of flash memory to be mapped into instruction and data address spaces. This mapping works only for read operations, it is not possible to modify contents of flash memory by writing to mapped memory region. Mapping happens in 64KB pages. Memory mapping hardware can map up to 4 megabytes of flash into data address space, and up to 16 megabytes of flash into instruction address space. See the technical reference manual for more details about memory mapping hardware.

Note that some number of 64KB pages is used to map the application itself into memory, so the actual number of available 64KB pages may be less.

Reading data from flash using a memory mapped region is the only way to decrypt contents of flash when [flash encryption](#) is enabled. Decryption is performed at hardware level.

Memory mapping APIs are declared in `esp_spi_flash.h` and `esp_partition.h`:

- `spi_flash_mmap` maps a region of physical flash addresses into instruction space or data space of the CPU
- `spi_flash_munmap` unmaps previously mapped region
- `esp_partition_mmap` maps part of a partition into the instruction space or data space of the CPU

Differences between `spi_flash_mmap` and `esp_partition_mmap` are as follows:

- `spi_flash_mmap` must be given a 64KB aligned physical address
- `esp_partition_mmap` may be given an arbitrary offset within the partition, it will adjust returned pointer to mapped memory as necessary

Note that because memory mapping happens in 64KB blocks, it may be possible to read data outside of the partition provided to `esp_partition_mmap`.

See also

- [Partition Table documentation](#)
- [Over The Air Update \(OTA\) API](#) provides high-level API for updating app firmware stored in flash.
- [Non-Volatile Storage \(NVS\) API](#) provides a structured API for storing small items of data in SPI flash.

Implementation details

In order to perform some flash operations, we need to make sure both CPUs are not running any code from flash for the duration of the flash operation. In a single-core setup this is easy: we disable interrupts/scheduler and do the flash operation. In the dual-core setup this is slightly more complicated. We need to make sure that the other CPU doesn't run any code from flash.

When SPI flash API is called on CPU A (can be PRO or APP), we start `spi_flash_op_block_func` function on CPU B using `esp_ipc_call` API. This API wakes up high priority task on CPU B and tells it to execute given function, in this case `spi_flash_op_block_func`. This function disables cache on CPU B and signals that cache is disabled by setting `s_flash_op_can_start` flag. Then the task on CPU A disables cache as well, and proceeds to execute flash operation.

While flash operation is running, interrupts can still run on CPUs A and B. We assume that all interrupt code is placed into RAM. Once interrupt allocation API is added, we should add a flag to request interrupt to be disabled for the duration of flash operations.

Once flash operation is complete, function on CPU A sets another flag, `s_flash_op_complete`, to let the task on CPU B know that it can re-enable cache and release the CPU. Then the function on CPU A re-enables the cache on CPU A as well and returns control to the calling code.

Additionally, all API functions are protected with a mutex (`s_flash_op_mutex`).

In a single core environment (`CONFIG_FREERTOS_UNICORE` enabled), we simply disable both caches, no inter-CPU communication takes place.

API Reference - SPI Flash

Header File

- `spi_flash/include/esp_spi_flash.h`

Functions

`void spi_flash_init()`

Initialize SPI flash access driver.

This function must be called exactly once, before any other `spi_flash_*` functions are called. Currently this function is called from startup code. There is no need to call it from application code.

`size_t spi_flash_get_chip_size()`

Get flash chip size, as set in binary image header.

Note This value does not necessarily match real flash size.

Return size of flash chip, in bytes

`esp_err_t spi_flash_erase_sector(size_t sector)`

Erase the Flash sector.

Return `esp_err_t`

Parameters

- `sector`: Sector number, the count starts at sector 0, 4KB per sector.

`esp_err_t spi_flash_erase_range(size_t start_address, size_t size)`

Erase a range of flash sectors.

Return `esp_err_t`

Parameters

- `start_address`: Address where erase operation has to start. Must be 4kB-aligned
- `size`: Size of erased range, in bytes. Must be divisible by 4kB.

`esp_err_t spi_flash_write(size_t dest_addr, const void *src, size_t size)`

Write data to Flash.

Note If source address is in DROM, this function will return `ESP_ERR_INVALID_ARG`.

Return `esp_err_t`

Parameters

- `dest_addr`: destination address in Flash. Must be a multiple of 4 bytes.
- `src`: pointer to the source buffer.
- `size`: length of data, in bytes. Must be a multiple of 4 bytes.

`esp_err_t spi_flash_write_encrypted(size_t dest_addr, const void *src, size_t size)`

Write data encrypted to Flash.

Note Flash encryption must be enabled for this function to work.

Note Flash encryption must be enabled when calling this function. If flash encryption is disabled, the function returns `ESP_ERR_INVALID_STATE`. Use `esp_flash_encryption_enabled()` function to determine if flash encryption is enabled.

Note Both `dest_addr` and `size` must be multiples of 16 bytes. For absolute best performance, both `dest_addr` and `size` arguments should be multiples of 32 bytes.

Return esp_err_t

Parameters

- dest_addr: destination address in Flash. Must be a multiple of 16 bytes.
- src: pointer to the source buffer.
- size: length of data, in bytes. Must be a multiple of 16 bytes.

esp_err_t **spi_flash_read**(size_t src_addr, void *dest, size_t size)

Read data from Flash.

Return esp_err_t

Parameters

- src_addr: source address of the data in Flash.
- dest: pointer to the destination buffer
- size: length of data

esp_err_t **spi_flash_read_encrypted**(size_t src, void *dest, size_t size)

Read data from Encrypted Flash.

If flash encryption is enabled, this function will transparently decrypt data as it is read. If flash encryption is not enabled, this function behaves the same as spi_flash_read().

See 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 64k MMU pages and configures them to map request region of flash memory into data address space or into instruction address space. It may reuse MMU pages which already provide required mapping. As with any allocator, there is possibility of fragmentation of address space if mmap/munmap are heavily used. To troubleshoot issues with page allocation, use spi_flash_mmap_dump function.

Return ESP_OK on success, ESP_ERR_NO_MEM if pages can not be allocated

Parameters

- src_addr: Physical address in flash where requested region starts. This address *must* be aligned to 64kB boundary (SPI_FLASH_MMU_PAGE_SIZE).
- size: Size of region which has to be mapped. This size will be rounded up to a 64k boundary.
- memory: Memory space where the region should be mapped
- out_ptr: Output, pointer to the mapped memory region
- out_handle: Output, handle which should be used for spi_flash_munmap call

```
esp_err_t spi_flash_mmap_pages (int *pages, size_t pagecount, spi_flash mmap memory_t memory,
                                const void **out_ptr, spi_flash mmap handle_t *out_handle)
```

Map sequences of pages of flash memory into data or instruction address space.

This function allocates sufficient number of 64k MMU pages and configures them to map the indicated pages of flash memory contiguously into data address space or into instruction address space. In this respect, it works in a similar way as `spi_flash_mmap` but it allows mapping a (maybe non-contiguous) set of pages into a contiguous region of memory.

Return `ESP_OK` on success, `ESP_ERR_NO_MEM` if pages can not be allocated

Parameters

- `pages`: An array of numbers indicating the 64K pages in flash to be mapped contiguously into memory. These indicate the indexes of the 64K pages, not the byte-size addresses as used in other functions.
- `pagecount`: Size of the pages array
- `memory`: Memory space where the region should be mapped
- `out_ptr`: Output, pointer to the mapped memory region
- `out_handle`: Output, handle which should be used for `spi_flash_munmap` call

```
void spi_flash_munmap (spi_flash mmap handle_t handle)
```

Release region previously obtained using `spi_flash_mmap`.

Note Calling this function will not necessarily unmap memory region. Region will only be unmapped when there are no other handles which reference this region. In case of partially overlapping regions it is possible that memory will be unmapped partially.

Parameters

- `handle`: Handle obtained from `spi_flash_mmap`

```
void spi_flash_mmap_dump ()
```

Display information about mapped regions.

This function lists handles obtained using `spi_flash_mmap`, along with range of pages allocated to each handle. It also lists all non-zero entries of MMU table and corresponding reference counts.

```
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 been assigned via `spi_flash_mmap()`, any address in flash map space can be looked up.

Return

- `SPI_FLASH_CACHE2PHYS_FAIL` If cache address is outside flash cache region, or the address is not mapped.
- Otherwise, returns physical offset in flash

Parameters

- `cached`: Pointer to flashed cached memory.

```
const void *spi_flash_phys2cache (size_t phys_offs, spi_flash mmap memory_t memory)
```

Given a physical offset in flash, return the address where it is mapped in the memory space.

Physical address does not have to have been assigned via `spi_flash_mmap()`, any address in flash can be looked up.

Note Only the first matching cache address is returned. If MMU flash cache table is configured so multiple entries point to the same physical address, there may be more than one cache address corresponding to that physical address. It is also possible for a single physical address to be mapped to both the IROM and DROM regions.

Note This function doesn't impose any alignment constraints, but if memory argument is `SPI_FLASH_MMAP_INST` and `phys_offs` is not 4-byte aligned, then reading from the returned pointer will result in a crash.

Return

- `NULL` if the physical address is invalid or not mapped to flash cache of the specified memory type.
- Cached memory address (in IROM or DROM space) corresponding to `phys_offs`.

Parameters

- `phys_offs`: Physical offset in flash memory to look up.
- `memory`: Memory type to look up a flash cache address mapping for (IROM or DROM)

`bool spi_flash_cache_enabled()`

Check at runtime if flash cache is enabled on both CPUs.

Return true if both CPUs have flash cache enabled, false otherwise.

`void spi_flash_guard_set (const spi_flash_guard_funcs_t *funcs)`

Sets guard functions to access flash.

Note Pointed structure and corresponding guard functions should not reside in flash. For example structure can be placed in DRAM and functions in IRAM sections.

Parameters

- `funcs`: pointer to structure holding flash access guard functions.

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 disable 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. 2) Functions which synchronize access to internal data used by flash API. These 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. 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 func

spi_flash_guard_end_func_t end
    critical section end func

spi_flash_op_lock_func_t op_lock
    flash access API lock func

spi_flash_op_unlock_func_t op_unlock
    flash access API unlock func
```

Macros

```
ESP_ERR_FLASH_BASE
ESP_ERR_FLASH_OP_FAIL
ESP_ERR_FLASH_OP_TIMEOUT
SPI_FLASH_SEC_SIZE
    SPI Flash sector size
SPI_FLASH_MMU_PAGE_SIZE
    Flash cache MMU mapping page size
SPI_FLASH_CACHE2PHYS_FAIL
```

Type Definitions

```
typedef uint32_t spi_flash_mmap_handle_t
    Opaque handle for memory region obtained from spi_flash_mmap.

typedef void (*spi_flash_guard_start_func_t)(void)
    SPI flash critical section enter function.

typedef void (*spi_flash_guard_end_func_t)(void)
    SPI flash critical section exit function.

typedef void (*spi_flash_op_lock_func_t)(void)
    SPI flash operation lock function.

typedef void (*spi_flash_op_unlock_func_t)(void)
    SPI flash operation unlock function.
```

Enumerations

`enum spi_flash_mmap_memory_t`

Enumeration which specifies memory space requested in an mmap call.

Values:

`SPI_FLASH_MMAP_DATA`

map to data memory (Vaddr0), allows byte-aligned access, 4 MB total

`SPI_FLASH_MMAP_INST`

map to instruction memory (Vaddr1-3), allows only 4-byte-aligned access, 11 MB total

API Reference - Partition Table

Header File

- `spi_flash/include/esp_partition.h`

Functions

`esp_partition_iterator_t esp_partition_find(esp_partition_type_t type, esp_partition_subtype_t subtype, const char *label)`

Find partition based on one or more parameters.

Return iterator which can be used to enumerate all the partitions found, or NULL if no partitions were found.

Iterator obtained through this function has to be released using `esp_partition_iterator_release` when not used any more.

Parameters

- `type`: Partition type, one of `esp_partition_type_t` values
- `subtype`: Partition subtype, one of `esp_partition_subtype_t` values. To find all partitions of given type, use `ESP_PARTITION_SUBTYPE_ANY`.
- `label`: (optional) Partition label. Set this value if looking for partition with a specific name. Pass NULL otherwise.

`const esp_partition_t *esp_partition_find_first(esp_partition_type_t type, esp_partition_subtype_t subtype, const char *label)`

Find first partition based on one or more parameters.

Return pointer to `esp_partition_t` structure, or NULL if no partition is found. This pointer is valid for the lifetime of the application.

Parameters

- `type`: Partition type, one of `esp_partition_type_t` values
- `subtype`: Partition subtype, one of `esp_partition_subtype_t` values. To find all partitions of given type, use `ESP_PARTITION_SUBTYPE_ANY`.
- `label`: (optional) Partition label. Set this value if looking for partition with a specific name. Pass NULL otherwise.

const esp_partition_t *esp_partition_get (esp_partition_iterator_t iterator)

Get *esp_partition_t* structure for given partition.

Return pointer to *esp_partition_t* structure. This pointer is valid for the lifetime of the application.

Parameters

- *iterator*: Iterator obtained using *esp_partition_find*. Must be non-NULL.

esp_partition_iterator_t esp_partition_next (esp_partition_iterator_t iterator)

Move partition iterator to the next partition found.

Any copies of the iterator will be invalid after this call.

Return NULL if no partition was found, valid *esp_partition_iterator_t* otherwise.

Parameters

- *iterator*: Iterator obtained using *esp_partition_find*. Must be non-NULL.

void esp_partition_iterator_release (esp_partition_iterator_t iterator)

Release partition iterator.

Parameters

- *iterator*: Iterator obtained using *esp_partition_find*. Must be non-NULL.

const esp_partition_t *esp_partition_verify (const esp_partition_t *partition)

Verify partition data.

Given a pointer to partition data, verify this partition exists in the partition table (all fields match.)

This function is also useful to take partition data which may be in a RAM buffer and convert it to a pointer to the permanent partition data stored in flash.

Pointers returned from this function can be compared directly to the address of any pointer returned from *esp_partition_get()*, as a test for equality.

Return

- If partition not found, returns NULL.
- If found, returns a pointer to the *esp_partition_t* structure in flash. This pointer is always valid for the lifetime of the application.

Parameters

- *partition*: Pointer to partition data to verify. Must be non-NULL. All fields of this structure must match the partition table entry in flash for this function to return a successful match.

esp_err_t esp_partition_read (const esp_partition_t *partition, size_t src_offset, void *dst, size_t size)

Read data from the partition.

Return ESP_OK, if data was read successfully; ESP_ERR_INVALID_ARG, if src_offset exceeds partition size; ESP_ERR_INVALID_SIZE, if read would go out of bounds of the partition; or one of error codes from lower-level flash driver.

Parameters

- *partition*: Pointer to partition structure obtained using *esp_partition_find_first* or *esp_partition_get*. Must be non-NULL.

- `dst`: Pointer to the buffer where data should be stored. Pointer must be non-NUL 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-NUL.
- `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-NUL 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-NUL.
- `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_NV = 0x02
NV partition.

ESP_PARTITION_SUBTYPE_DATA_COREDUMP = 0x03
COREDUMP partition.

ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD = 0x80
ESPHTTPD partition.

ESP_PARTITION_SUBTYPE_DATA_FAT = 0x81
FAT partition.

ESP_PARTITION_SUBTYPE_DATA_SPIFFS = 0x82
SPIFFS partition.

ESP_PARTITION_SUBTYPE_ANY = 0xff
Used to search for partitions with any subtype.

API Reference - Flash Encrypt

Header File

- bootloader_support/include/esp_flash_encrypt.h

Functions

static bool esp_flash_encryption_enabled (void)

Is flash encryption currently enabled in hardware?

Flash encryption is enabled if the FLASH_CRYPT_CNT efuse has an odd number of bits set.

Return true if flash encryption is enabled.

esp_err_t esp_flash_encrypt_check_and_update (void)

esp_err_t esp_flash_encrypt_region (uint32_t src_addr, size_t data_length)

Encrypt-in-place a block of flash sectors.

Return ESP_OK if all operations succeeded, ESP_ERR_FLASH_OP_FAIL if SPI flash fails, ESP_ERR_FLASH_OP_TIMEOUT if flash times out.

Parameters

- **src_addr:** Source offset in flash. Should be multiple of 4096 bytes.
- **data_length:** Length of data to encrypt in bytes. Will be rounded up to next multiple of 4096 bytes.

2.6.2 Non-volatile storage library

Introduction

Non-volatile storage (NVS) library is designed to store key-value pairs in flash. This sections introduces some concepts used by NVS.

Underlying storage

Currently NVS uses a portion of main flash memory through `spi_flash_{read|write|erase}` APIs. The library uses the all the partitions with `data` type and `nvs` subtype. The application can choose to use the partition with label `nvs` through `nvs_open` API or any of the other partition by specifying its name through `nvs_open_from_part` API.

Future versions of this library may add other storage backends to keep data in another flash chip (SPI or I2C), RTC, FRAM, etc.

Note: if an NVS partition is truncated (for example, when the partition table layout is changed), its contents should be erased. ESP-IDF build system provides a make `erase_flash` target to erase all contents of the flash chip.

Note: NVS works best for storing many small values, rather than a few large values of type ‘string’ and ‘blob’. If storing large blobs or strings is required, consider using the facilities provided by the FAT filesystem on top of the wear levelling library.

Keys and values

NVS operates on key-value pairs. Keys are ASCII strings, maximum key length is currently 15 characters. Values can have one of the following types:

- integer types: `uint8_t`, `int8_t`, `uint16_t`, `int16_t`, `uint32_t`, `int32_t`, `uint64_t`, `int64_t`
- zero-terminated string
- variable length binary data (blob)

Note: String and blob values are currently limited to 1984 bytes. For strings, this includes the null terminator.

Additional types, such as `float` and `double` may be added later.

Keys are required to be unique. Writing a value for a key which already exists behaves as follows:

- if the new value is of the same type as old one, value is updated
- if the new value has different data type, an error is returned

Data type check is also performed when reading a value. An error is returned if data type of read operation doesn't match the data type of the value.

Namespaces

To mitigate potential conflicts in key names between different components, NVS assigns each key-value pair to one of namespaces. Namespace names follow the same rules as key names, i.e. 15 character maximum length. Namespace name is specified in the `nvs_open` or `nvs_open_from_part` call. This call returns an opaque handle, which is used in subsequent calls to `nvs_read_*`, `nvs_write_*`, and `nvs_commit` functions. This way, handle is associated with a namespace, and key names will not collide with same names in other namespaces. Please note that the namespaces with same name in different NVS partitions are considered as separate namespaces.

Security, tampering, and robustness

NVS library doesn't implement tamper prevention measures. It is possible for anyone with physical access to the flash chip to alter, erase, or add key-value pairs.

NVS is compatible with the ESP32 flash encryption system, and it can store key-value pairs in an encrypted form. Some metadata, like page state and write/erase flags of individual entries can not be encrypted as they are represented as bits of flash memory for efficient access and manipulation. Flash encryption can prevent some forms of modification:

- replacing keys or values with arbitrary data
- changing data types of values

The following forms of modification are still possible when flash encryption is used:

- erasing a page completely, removing all key-value pairs which were stored in that page
- corrupting data in a page, which will cause the page to be erased automatically when such condition is detected
- rolling back the contents of flash memory to an earlier snapshot
- merging two snapshots of flash memory, rolling back some key-value pairs to an earlier state (although this is possible to mitigate with the current design — TODO)

The library does try to recover from conditions when flash memory is in an inconsistent state. In particular, one should be able to power off the device at any point and time and then power it back on. This should not result in loss of data, except 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.

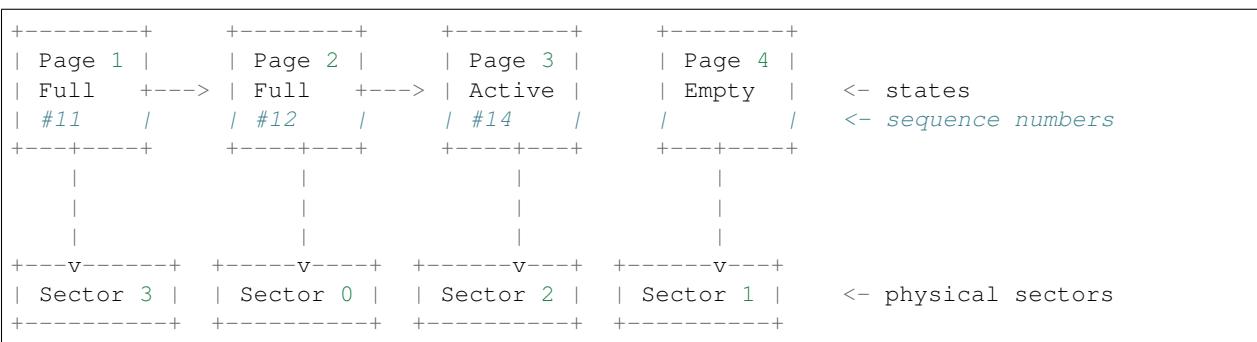
Active Flash storage is initialized, page header has been written to flash, page has a valid sequence number. Page has some empty entries and data can be written there. At most one page can be in this state at any given moment.

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.



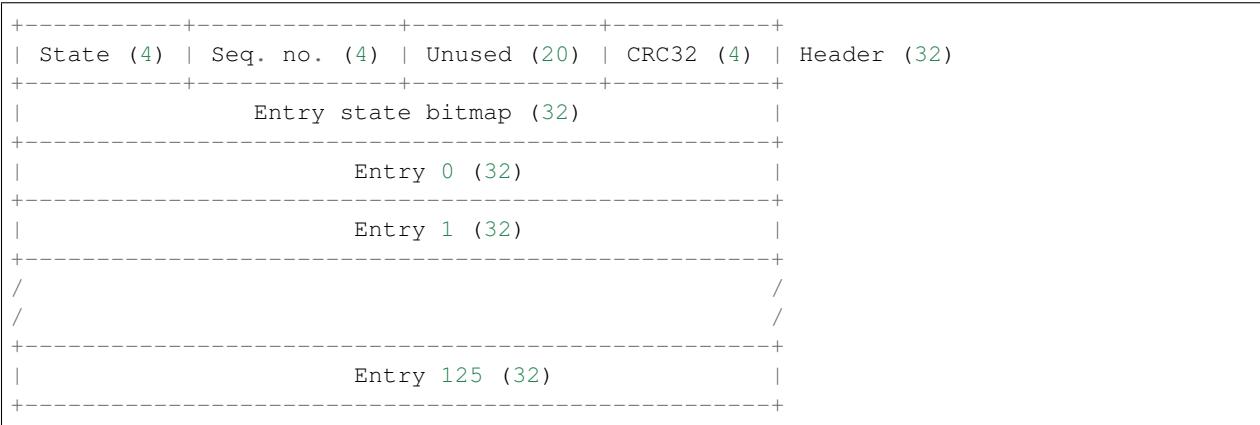
Structure of a page

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.



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 is not necessary to erase the page to change page state, unless that is a change to *erased* state.

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. Future versions of the library may store format version there.

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.

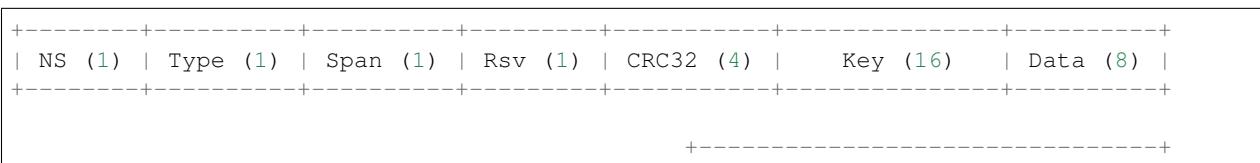
Empty (2'b11) Nothing is written into the specific entry yet. It is in an uninitialized state (all bytes 0xff).

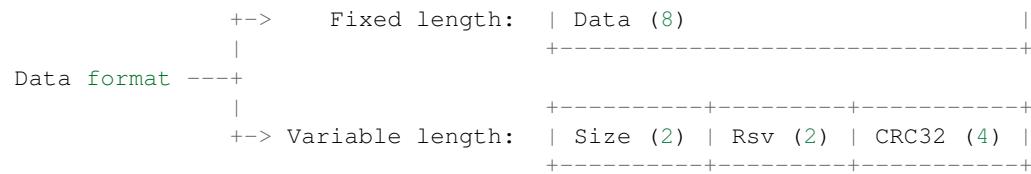
Written (2'b10) A key-value pair (or part of key-value pair which spans multiple entries) has been written into the entry.

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.





Individual fields in entry structure have the following meanings:

NS Namespace index for this entry. See section on namespaces implementation for explanation of this value.

Type One byte indicating data type of value. See `ItemType` enumeration in `nvs_types.h` for possible values.

Span Number of entries used by this key-value pair. For integer types, this is equal to 1. For strings and blobs this depends on value length.

Rsv Unused field, should be `0xff`.

CRC32 Checksum calculated over all the bytes in this entry, except for the CRC32 field itself.

Key Zero-terminated ASCII string containing key name. Maximum string length is 15 bytes, excluding zero terminator.

Data For integer types, this field contains the value itself. If the value itself is shorter than 8 bytes it is padded to the right, with unused bytes filled with `0xff`. For string and blob values, these 8 bytes hold additional data about the value, described next:

Size (Only for strings and blobs.) Size, in bytes, of actual data. For strings, this includes zero terminator.

CRC32 (Only for strings and blobs.) Checksum calculated over all bytes of data.

Variable length values (strings and blobs) are written into subsequent entries, 32 bytes per entry. *Span* field of the first entry indicates how many entries are used.

Namespaces

As mentioned above, each key-value pair belongs to one of the namespaces. Namespaces identifiers (strings) are stored as keys of key-value pairs in namespace with index 0. Values corresponding to these keys are indexes of these namespaces.

NS=0 Type=uint8_t Key="wifi" Value=1	Entry describing namespace "wifi"
+-----+	
NS=1 Type=uint32_t Key="channel" Value=6	Key "channel" in namespace "wifi"
+-----+	
NS=0 Type=uint8_t Key="pwm" Value=2	Entry describing namespace "pwm"
+-----+	
NS=2 Type=uint16_t Key="channel" Value=20	Key "channel" in namespace "pwm"
+-----+	

Item hash list

To reduce the number of reads performed from flash memory, each member of Page class maintains a list of pairs: (item index; item hash). This list makes searches much quicker. Instead of iterating over all entries, reading them from flash one at a time, `Page::findItem` first performs search for item hash in the hash list. This gives the item index within the page, if such an item exists. Due to a hash collision it is possible that a different item will be found. This is handled by falling back to iteration over items in flash.

Each node in hash list contains a 24-bit hash and 8-bit item index. Hash is calculated based on item namespace and key name. CRC32 is used for calculation, result is truncated to 24 bits. To reduce overhead of storing 32-bit entries in a linked list, list is implemented as a doubly-linked list of arrays. Each array holds 29 entries, for the total size of 128 bytes, together with linked list pointers and 32-bit count field. Minimal amount of extra RAM usage per page is therefore 128 bytes, maximum is 640 bytes.

Application Example

Two examples are provided in `storage` directory of ESP-IDF examples:

`storage/nvs_rw_value`

Demonstrates how to read and write a single integer value using NVS.

The value holds the number of ESP32 module restarts. Since it is written to NVS, the value is preserved between restarts.

Example also shows how to check if read / write operation was successful, or certain value is not initialized in NVS. Diagnostic is provided in plain text to help track program flow and capture any issues on the way.

`storage/nvs_rw_blob`

Demonstrates how to read and write a single integer value and a blob (binary large object) using NVS to preserve them between ESP32 module restarts.

- value - tracks number of ESP32 module soft and hard restarts.
- blob - contains a table with module run times. The table is read from NVS to dynamically allocated RAM. New run time is added to the table on each manually triggered soft restart and written back to NVS. Triggering is done by pulling down GPIO0.

Example also shows how to implement diagnostics if read / write operation was successful.

API Reference

Header File

- `nvs_flash/include/nvs_flash.h`

Functions

`esp_err_t nvs_flash_init(void)`

Initialize the default NVS partition.

This API initialises the default NVS partition. The default NVS partition is the one that is labelled “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_name)`
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_name`: Name (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_erase(void)`
Erase the default NVS partition.

This function erases all contents of the default NVS partition (one with label “nvs”)

Return

- ESP_OK on success
- ESP_ERR_NOT_FOUND if there is no NVS partition labeled “nvs” in the partition table

`esp_err_t nvs_flash_erase_partition(const char *part_name)`
Erase specified NVS partition.

This function erases all contents of specified NVS partition

Return

- ESP_OK on success
- ESP_ERR_NOT_FOUND if there is no NVS partition with the specified name in the partition table

Parameters

- `part_name`: Name (label) of the partition to be erased

Header File

- `nvs_flash/include/nvs.h`

Functions

`esp_err_t nvs_set_i8(nvs_handle handle, const char *key, int8_t value)`
set value for given key

This family of functions set value for the key, given its name. Note that actual storage will not be updated until `nvs_commit` function is called.

Return

- ESP_OK if value was set successfully

- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_READ_ONLY if storage handle was opened as read only
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is not enough space in the underlying storage to save the value
- ESP_ERR_NVS_REMOVE_FAILED if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.
- ESP_ERR_NVS_VALUE_TOO_LONG if the string value is too long

Parameters

- handle: Handle obtained from nvs_open function. Handles that were opened read only cannot be used.
- key: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- value: The value to set. For strings, the maximum length (including null character) is 1984 bytes.

```
esp_err_t nvs_set_u8 (nvs_handle handle, const char *key, uint8_t value)
esp_err_t nvs_set_i16 (nvs_handle handle, const char *key, int16_t value)
esp_err_t nvs_set_u16 (nvs_handle handle, const char *key, uint16_t value)
esp_err_t nvs_set_i32 (nvs_handle handle, const char *key, int32_t value)
esp_err_t nvs_set_u32 (nvs_handle handle, const char *key, uint32_t value)
esp_err_t nvs_set_i64 (nvs_handle handle, const char *key, int64_t value)
esp_err_t nvs_set_u64 (nvs_handle handle, const char *key, uint64_t value)
esp_err_t nvs_set_str (nvs_handle handle, const char *key, const char *value)
esp_err_t nvs_get_i8 (nvs_handle handle, const char *key, int8_t *out_value)
    get value for given key
```

These functions retrieve value for the key, given its name. If key does not exist, or the requested variable type doesn't match the type which was used when setting a value, an error is returned.

In case of any error, out_value is not modified.

All functions expect out_value to be a pointer to an already allocated variable of the given type.

```
// Example of using nvs_get_i32:
int32_t max_buffer_size = 4096; // default value
esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_buffer_size);
assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
// if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
// have its default value.
```

Return

- ESP_OK if the value was retrieved successfully
- ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL

- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_INVALID_LENGTH if length is not sufficient to store data

Parameters

- handle: Handle obtained from nvs_open function.
- key: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- out_value: Pointer to the output value. May be NULL for nvs_get_str and nvs_get_blob, in this case required length will be returned in length argument.

```
esp_err_t nvs_get_u8 (nvs_handle handle, const char *key, uint8_t *out_value)
esp_err_t nvs_get_i16 (nvs_handle handle, const char *key, int16_t *out_value)
esp_err_t nvs_get_u16 (nvs_handle handle, const char *key, uint16_t *out_value)
esp_err_t nvs_get_i32 (nvs_handle handle, const char *key, int32_t *out_value)
esp_err_t nvs_get_u32 (nvs_handle handle, const char *key, uint32_t *out_value)
esp_err_t nvs_get_i64 (nvs_handle handle, const char *key, int64_t *out_value)
esp_err_t nvs_get_u64 (nvs_handle handle, const char *key, uint64_t *out_value)
esp_err_t nvs_get_str (nvs_handle handle, const char *key, char *out_value, size_t *length)
    get value for given key
```

These functions retrieve value for the key, given its name. If key does not exist, or the requested variable type doesn't match the type which was used when setting a value, an error is returned.

In case of any error, out_value is not modified.

All functions expect out_value to be a pointer to an already allocated variable of the given type.

nvs_get_str and nvs_get_blob functions support WinAPI-style length queries. To get the size necessary to store the value, call nvs_get_str or nvs_get_blob with zero out_value and non-zero pointer to length. Variable pointed to by length argument will be set to the required length. For nvs_get_str, this length includes the zero terminator. When calling nvs_get_str and nvs_get_blob with non-zero out_value, length has to be non-zero and has to point to the length available in out_value. It is suggested that nvs_get/set_str is used for zero-terminated C strings, and nvs_get/set_blob used for arbitrary data structures.

```
// Example (without error checking) of using nvs_get_str to get a string into a dynamic array:
size_t required_size;
nvs_get_str(my_handle, "server_name", NULL, &required_size);
char* server_name = malloc(required_size);
nvs_get_str(my_handle, "server_name", server_name, &required_size);

// Example (without error checking) of using nvs_get_blob to get a binary data into a static array:
uint8_t mac_addr[6];
size_t size = sizeof(mac_addr);
nvs_get_blob(my_handle, "dst_mac_addr", mac_addr, &size);
```

Return

- ESP_OK if the value was retrieved successfully
- ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist

- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_INVALID_LENGTH if length is not sufficient to store data

Parameters

- `handle`: Handle obtained from `nvs_open` function.
- `key`: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- `out_value`: Pointer to the output value. May be NULL for `nvs_get_str` and `nvs_get_blob`, in this case required length will be returned in `length` argument.
- `length`: A non-zero pointer to the variable holding the length of `out_value`. In case `out_value` is 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 if namespace doesn't exist yet and mode is NVS_READONLY
- ESP_ERR_NVS_INVALID_NAME if namespace name doesn't satisfy constraints
- other error codes from the underlying storage driver

Parameters

- `name`: Namespace name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- `open_mode`: NVS_READWRITE or NVS_READONLY. If NVS_READONLY, will open a handle for reading only. All write requests will be rejected for this handle.
- `out_handle`: If successful (return code is zero), handle will be returned in this argument.

`esp_err_t nvs_open_from_partition(const char *part_name, const char *name, nvs_open_mode open_mode, nvs_handle *out_handle)`

Open non-volatile storage with a given namespace from specified partition.

The behaviour is same as `nvs_open()` API. However this API can operate on a specified NVS partition instead of default NVS partition. Note that the specified partition must be registered with NVS using `nvs_flash_init_partition()` API.

Return

- ESP_OK if storage handle was opened successfully

- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized
- ESP_ERR_NVS_PART_NOT_FOUND if the partition with specified name is not found
- ESP_ERR_NVS_NOT_FOUND id namespace doesn't exist yet and mode is NVS_READONLY
- ESP_ERR_NVS_INVALID_NAME if namespace name doesn't satisfy constraints
- other error codes from the underlying storage driver

Parameters

- part_name: Label (name) of the partition of interest for object read/write/erase
- name: Namespace name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- open_mode: NVS_READWRITE or NVS_READONLY. If NVS_READONLY, will open a handle for reading only. All write requests will be rejected for this handle.
- out_handle: If successful (return code is zero), handle will be returned in this argument.

`esp_err_t nvs_set_blob (nvs_handle handle, const char *key, const void *value, size_t length)`
set variable length binary value for given key

This family of functions set value for the key, given its name. Note that actual storage will not be updated until nvs_commit function is called.

Return

- ESP_OK if value was set successfully
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_READ_ONLY if storage handle was opened as read only
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is not enough space in the underlying storage to save the value
- ESP_ERR_NVS_REMOVE_FAILED if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.
- ESP_ERR_NVS_VALUE_TOO_LONG if the value is too long

Parameters

- handle: Handle obtained from nvs_open function. Handles that were opened read only cannot be used.
- key: Key name. Maximal length is 15 characters. Shouldn't be empty.
- value: The value to set.
- length: length of binary value to set, in bytes; Maximum length is 1984 bytes.

`esp_err_t nvs_erase_key (nvs_handle handle, const char *key)`
Erase key-value pair with given key name.

Note that actual storage may not be updated until nvs_commit function is called.

Return

- ESP_OK if erase operation was successful

- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_READ_ONLY if handle was opened as read only
- ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist
- other error codes from the underlying storage driver

Parameters

- `handle`: Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.
- `key`: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.

`esp_err_t nvs_erase_all (nvs_handle handle)`

Erase all key-value pairs in a namespace.

Note that actual storage may not be updated until `nvs_commit` function is called.

Return

- `ESP_OK` if erase operation was successful
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is NULL
- `ESP_ERR_NVS_READ_ONLY` if handle was opened as read only
- other error codes from the underlying storage driver

Parameters

- `handle`: Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.

`esp_err_t nvs_commit (nvs_handle handle)`

Write any pending changes to non-volatile storage.

After setting any values, `nvs_commit()` must be called to ensure changes are written to non-volatile storage. Individual implementations may write to storage at other times, but this is not guaranteed.

Return

- `ESP_OK` if the changes have been written successfully
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is NULL
- other error codes from the underlying storage driver

Parameters

- `handle`: Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.

`void nvs_close (nvs_handle handle)`

Close the storage handle and free any allocated resources.

This function should be called for each handle opened with `nvs_open` once the handle is not in use any more. Closing the handle may not automatically write the changes to nonvolatile storage. This has to be done explicitly using `nvs_commit` function. Once this function is called on a handle, the handle should no longer be used.

Parameters

- handle: Storage handle to close

Macros

ESP_ERR_NVS_BASE

Starting number of error codes

ESP_ERR_NVS_NOT_INITIALIZED

The storage driver is not initialized

ESP_ERR_NVS_NOT_FOUND

Id namespace doesn't exist yet and mode is NVS_READONLY

ESP_ERR_NVS_TYPE_MISMATCH

The type of set or get operation doesn't match the type of value stored in NVS

ESP_ERR_NVS_READ_ONLY

Storage handle was opened as read only

ESP_ERR_NVS_NOT_ENOUGH_SPACE

There is not enough space in the underlying storage to save the value

ESP_ERR_NVS_INVALID_NAME

Namespace name doesn't satisfy constraints

ESP_ERR_NVS_INVALID_HANDLE

Handle has been closed or is NULL

ESP_ERR_NVS_REMOVE_FAILED

The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

ESP_ERR_NVS_KEY_TOO_LONG

Key name is too long

ESP_ERR_NVS_PAGE_FULL

Internal error; never returned by nvs_ API functions

ESP_ERR_NVS_INVALID_STATE

NVS is in an inconsistent state due to a previous error. Call nvs_flash_init and nvs_open again, then retry.

ESP_ERR_NVS_INVALID_LENGTH

String or blob length is not sufficient to store data

ESP_ERR_NVS_NO_FREE_PAGES

NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call nvs_flash_init again.

ESP_ERR_NVS_VALUE_TOO_LONG

String or blob length is longer than supported by the implementation

ESP_ERR_NVS_PART_NOT_FOUND

Partition with specified name is not found in the partition table

NVS_DEFAULT_PART_NAME

Default partition name of the NVS partition in the partition table

Type Definitions

typedef uint32_t **nvs_handle**
 Opaque pointer type representing non-volatile storage handle

Enumerations

enum nvs_open_mode
 Mode of opening the non-volatile storage.

Values:

NVS_READONLY

Read only

NVS_READWRITE

Read and write

2.6.3 Virtual filesystem component

Overview

Virtual filesystem (VFS) component provides a unified interface for drivers which can perform operations on file-like objects. This can be a real filesystems (FAT, SPIFFS, etc.), or device drivers which exposes file-like interface.

This component allows C library functions, such as fopen and fprintf, to work with FS drivers. At high level, each FS driver is associated with some path prefix. When one of C library functions needs to open a file, VFS component searches for the FS driver associated with the file's path, and forwards the call to that driver. VFS also forwards read, write, and other calls for the given file to the same FS driver.

For example, one can register a FAT filesystem driver with /fat prefix, and call fopen("/fat/file.txt", "w"). VFS component will then call open function of FAT driver and pass /file.txt argument to it (and appropriate mode flags). All subsequent calls to C library functions for the returned FILE* stream will also be forwarded to the FAT driver.

FS registration

To register an FS driver, application needs to define an instance of esp_vfs_t structure and populate it with function pointers to FS APIs:

```
esp_vfs_t myfs = {
    .fd_offset = 0,
    .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

// When registering FS, context pointer (third argument) is NULL:
ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

Case 2: API functions are declared with an extra context pointer (FS driver supports multiple instances):

```
ssize_t myfs_write(myfs_t* fs, int fd, const void * data, size_t size);

// In definition of esp_vfs_t:
.flags = ESP_VFS_FLAG_CONTEXT_PTR,
.write_p = &myfs_write,
// ... other members initialized

// When registering FS, pass the FS context pointer into the third argument
// (hypothetical myfs_mount function is used for illustrative purposes)
myfs_t* myfs_inst1 = myfs_mount(partition1->offset, partition1->size);
ESP_ERROR_CHECK(esp_vfs_register("/data1", &myfs, myfs_inst1));

// Can register another instance:
myfs_t* myfs_inst2 = myfs_mount(partition2->offset, partition2->size);
ESP_ERROR_CHECK(esp_vfs_register("/data2", &myfs, myfs_inst2));
```

Paths

Each registered FS has a path prefix associated with it. This prefix may be considered a “mount point” of this partition. In case when mount points are nested, the mount point with the longest matching path prefix is used when opening the file. For instance, suppose that the following filesystems are registered in VFS:

- FS 1 on /data
- FS 2 on /data/static

Then:

- FS 1 will be used when opening a file called /data/log.txt
- FS 2 will be used when opening a file called /data/static/index.html
- Even if /index.html" doesn't exist in FS 2, FS 1 will *not* be searched for /static/index.html.

As a general rule, mount point names must start with the path separator (/) and must contain at least one character after path separator. However an empty mount point name is also supported, and may be used in cases when application needs to provide “fallback” filesystem, or override VFS functionality altogether. Such filesystem will be used if no prefix matches the path given.

VFS does not handle dots (.) in path names in any special way. VFS does not treat .. as a reference to the parent directory. I.e. in the above example, using a path /data/static/..../log.txt will not result in a call to FS 1 to open /log.txt. Specific FS drivers (such as FATFS) may handle dots in file names differently.

When opening files, FS driver will only be given relative path to files. For example:

- myfs driver is registered with /data as path prefix

- and application calls `fopen("/data/config.json", ...)`
- then VFS component will call `myfs_open("/config.json", ...)`.
- `myfs` driver will open `/config.json` file

VFS doesn't impose a limit on total file path length, but it does limit FS path prefix to `ESP_VFS_PATH_MAX` characters. Individual FS drivers may have their own filename length limitations.

File descriptors

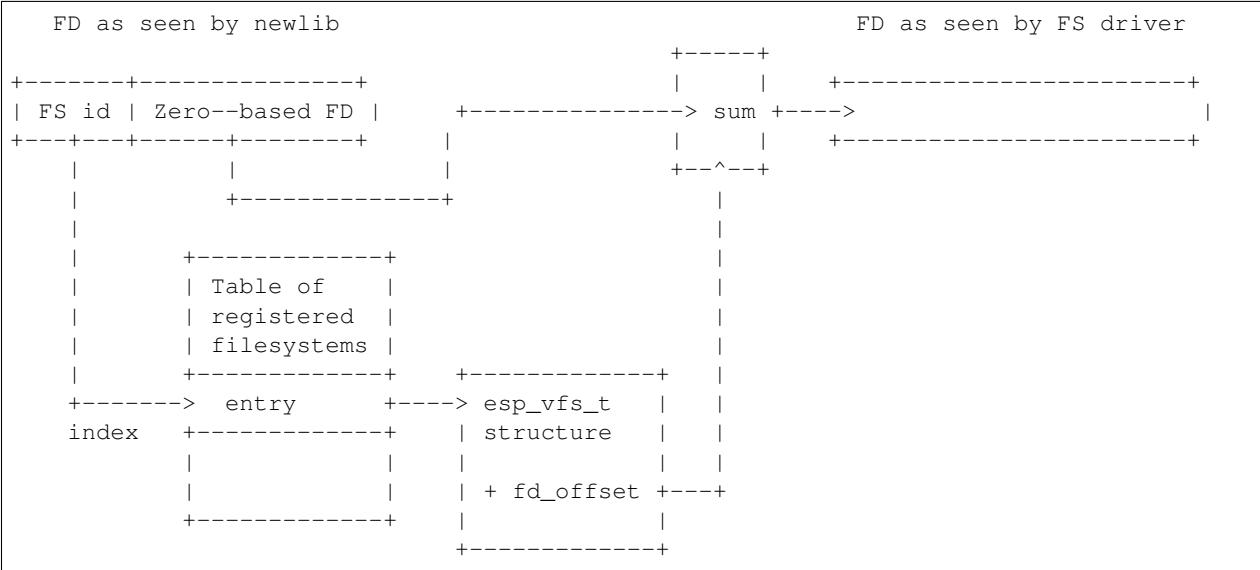
It is suggested that filesystem drivers should use small positive integers as file descriptors. VFS component assumes that `CONFIG_MAX_FD_BITS` bits (12 by default) are sufficient to represent a file descriptor.

If filesystem is configured with an option to offset all file descriptors by a constant value, such value should be passed to `fd_offset` field of `esp_vfs_t` structure. VFS component will then remove this offset when working with FDs of that specific FS, bringing them into the range of small positive integers.

While file descriptors returned by VFS component to newlib library are rarely seen by the application, the following details may be useful for debugging purposes. File descriptors returned by VFS component are composed of two parts: FS driver ID, and the actual file descriptor. Because newlib stores file descriptors as 16-bit integers, VFS component is also limited by 16 bits to store both parts.

Lower `CONFIG_MAX_FD_BITS` bits are used to store zero-based file descriptor. If FS driver has a non-zero `fd_offset` field, this `fd_offset` is subtracted FDs obtained from the FS open call, and the result is stored in the lower bits of the FD. Higher bits are used to save the index of FS in the internal table of registered filesystems.

When VFS component receives a call from newlib which has a file descriptor, this file descriptor is translated back to the FS-specific file descriptor. First, higher bits of FD are used to identify the FS. Then `fd_offset` field of the FS is added to the lower `CONFIG_MAX_FD_BITS` bits of the fd, and resulting FD is passed to the FS driver.



Standard IO streams (stdin, stdout, stderr)

If “UART for console output” menuconfig option is not set to “None”, then `stdin`, `stdout`, and `stderr` are configured to read from, and write to, a UART. It is possible to use UART0 or UART1 for standard IO. By default, UART0 is used, with 115200 baud rate, TX pin is GPIO1 and RX pin is GPIO3. These parameters can be changed in menuconfig.

Writing to `stdout` or `stderr` will send characters to the UART transmit FIFO. Reading from `stdin` will retrieve characters from the UART receive FIFO.

By default, VFS uses simple functions for reading from and writing to UART. Writes busy-wait until all data is put into UART FIFO, and reads are non-blocking, returning only the data present in the FIFO. Because of this non-blocking read behavior, higher level C library calls, such as `fscanf("%d\n", &var);` may not have desired results.

Applications which use UART driver may instruct VFS to use the driver's interrupt driven, blocking read and write functions instead. This can be done using a call to `esp_vfs_dev_uart_use_driver` function. It is also possible to revert to the basic non-blocking functions using a call to `esp_vfs_dev_uart_use_nonblocking`.

VFS also provides optional newline conversion feature for input and output. Internally, most applications send and receive lines terminated by LF ('n') character. Different terminal programs may require different line termination, such as CR or CRLF. Applications can configure this separately for input and output either via menuconfig, or by calls to `esp_vfs_dev_uart_set_rx_line_endings` and `esp_vfs_dev_uart_set_tx_line_endings` functions.

Standard streams and FreeRTOS tasks

`FILE` objects for `stdin`, `stdout`, and `stderr` are shared between all FreeRTOS tasks, but the pointers to these objects are stored in per-task `struct _reent`. The following code:

```
fprintf(stderr, "42\n");
```

actually is translated to 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_unregister(const char *base_path)`

Unregister a virtual filesystem for given path prefix

Return ESP_OK if successful, `ESP_ERR_INVALID_STATE` if VFS for given prefix hasn’t been registered

Parameters

- `base_path`: file prefix previously used in `esp_vfs_register` call

Structures

`struct esp_vfs_t`

VFS definition structure.

This structure should be filled with pointers to corresponding FS driver functions.

If the FS implementation has an option to use certain offset for all file descriptors, this value should be passed into `fd_offset` field. Otherwise VFS component will translate all FDs to start at zero offset.

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 fd_offset

file descriptor offset, determined by the FS driver

int flags

ESP_VFS_FLAG_CONTEXT_PTR or ESP_VFS_FLAG_DEFAULT

Macros

ESP_VFS_PATH_MAX

Maximum length of path prefix (not including zero terminator)

ESP_VFS_FLAG_DEFAULT

Default value of flags member in *esp_vfs_t* structure.

ESP_VFS_FLAG_CONTEXT_PTR

Flag which indicates that FS needs extra context pointer in syscalls.

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 ('\n', LF) passed into stdin:

- ESP_LINE_ENDINGS_CRLF: convert CRLF to LF
- ESP_LINE_ENDINGS_CR: convert CR to LF
- ESP_LINE_ENDINGS_LF: no modification

Note this function is not thread safe w.r.t. reading from UART

Parameters

- mode: line endings expected on UART

```
void esp_vfs_dev_uart_set_tx_line_endings (esp_line_endings_t mode)
```

Set the line endings to sent to UART.

This specifies the conversion between newlines (' ', LF) on stdout and line endings sent over UART:

- ESP_LINE_ENDINGS_CRLF: convert LF to CRLF
- ESP_LINE_ENDINGS_CR: convert LF to CR
- ESP_LINE_ENDINGS_LF: no modification

Note this function is not thread safe w.r.t. writing to UART

Parameters

- mode: line endings to send to UART

```
void esp_vfs_dev_uart_use_nonblocking (int uart_num)
```

set VFS to use simple functions for reading and writing UART Read is non-blocking, write is busy waiting until TX FIFO has enough space. These functions are used by default.

Parameters

- uart_num: UART peripheral number

```
void esp_vfs_dev_uart_use_driver (int uart_num)
```

set VFS to use UART driver for reading and writing

Note application must configure UART driver before calling these functions With these functions, read and write are blocking and interrupt-driven.

Parameters

- uart_num: UART peripheral number

Enumerations

```
enum esp_line_endings_t
```

Line ending settings.

Values:

```
ESP_LINE_ENDINGS_CRLF
```

CR + LF.

```
ESP_LINE_ENDINGS_CR
```

CR.

```
ESP_LINE_ENDINGS_LF
```

LF.

2.6.4 FAT Filesystem Support

ESP-IDF uses [FatFs](#) library to work with FAT filesystems. FatFs library resides in `fatfs` component. Although it can be used directly, many of its features can be accessed via VFS using C standard library and POSIX APIs.

Additionally, FatFs has been modified to support run-time pluggable disk IO layer. This allows mapping of FatFs drives to physical disks at run-time.

Using FatFs with VFS

`esp_vfs_fat.h` header file defines functions to connect FatFs with VFS. `esp_vfs_fat_register` function allocates a FATFS structure, and registers a given path prefix in VFS. Subsequent operations on files starting with this prefix are forwarded to FatFs APIs. `esp_vfs_fat_unregister_path` function deletes the registration with VFS, and frees the FATFS structure.

Most applications will use the following flow when working with `esp_vfs_fat_` functions:

1. Call `esp_vfs_fat_register`, specifying path prefix where the filesystem has to be mounted (e.g. `"/sdcard"`, `"/spiflash"`), FatFs drive number, and a variable which will receive a pointer to FATFS structure.
2. Call `ff_diskio_register` function to register disk IO driver for the drive number used in step 1.
3. Call `f_mount` function (and optionally `f_fdisk`, `f_mkfs`) to mount the filesystem using the same drive number which was passed to `esp_vfs_fat_register`. See FatFs documentation for more details.
4. Call POSIX and C standard library functions to open, read, write, erase, copy files, etc. Use paths starting with the prefix passed to `esp_vfs_register` (such as `"/sdcard/hello.txt"`).
5. Optionally, call FatFs library functions directly. Use paths without a VFS prefix in this case (`"/hello.txt"`).
6. Close all open files.
7. Call `f_mount` function for the same drive number, with NULL FATFS* argument, to unmount the filesystem.
8. Call `ff_diskio_register` with NULL `ff_diskio_impl_t*` argument and the same drive number.
9. Call `esp_vfs_fat_unregister_path` with the path where the file system is mounted to remove FatFs from VFS, and free the FATFS structure allocated on step 1.

Convenience functions, `esp_vfs_fat_sdmmc_mount` and `esp_vfs_fat_sdmmc_unmount`, which wrap these steps and also handle SD card initialization, are described in the next section.

```
esp_err_t esp_vfs_fat_register(const char *base_path, const char *fat_drive, size_t max_files,  
                                FATFS **out_fs)
```

Register FATFS with VFS component.

This function registers given FAT drive in VFS, at the specified base path. If only one drive is used, `fat_drive` argument can be an empty string. Refer to FATFS library documentation on how to specify FAT drive. This function also allocates FATFS structure which should be used for `f_mount` call.

Note This function doesn't mount the drive into FATFS, it just connects POSIX and C standard library IO function with FATFS. You need to mount desired drive into FATFS separately.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `esp_vfs_fat_register` was already called
- `ESP_ERR_NO_MEM` if not enough memory or too many VFSes already registered

Parameters

- `base_path`: path prefix where FATFS should be registered
- `fat_drive`: FATFS drive specification; if only one drive is used, can be an empty string
- `max_files`: maximum number of files which can be open at the same time
- `out_fs`: pointer to FATFS structure which can be used for FATFS `f_mount` call is returned via this argument.

`esp_err_t esp_vfs_fat_unregister_path (const char *base_path)`
Un-register FATFS from VFS.

Note FATFS structure returned by `esp_vfs_fat_register` is destroyed after this call. Make sure to call `f_mount` function to unmount it before calling `esp_vfs_fat_unregister_ctx`. Difference between this function and the one above is that this one will release the correct drive, while the one above will release the last registered one

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if FATFS is not registered in VFS

Parameters

- `base_path`: path prefix where FATFS is registered. This is the same used when `esp_vfs_fat_register` was called

Using FatFs with VFS and SD cards

`esp_vfs_fat.h` header file also provides a convenience function to perform steps 1–3 and 7–9, and also handle SD card initialization: `esp_vfs_fat_sdmmc_mount`. This function does only limited error handling. Developers are encouraged to look at its source code and incorporate more advanced versions into production applications. `esp_vfs_fat_sdmmc_unmount` function unmounts the filesystem and releases resources acquired by `esp_vfs_fat_sdmmc_mount`.

`esp_err_t esp_vfs_fat_sdmmc_mount (const char *base_path, const sdmmc_host_t *host_config,
const void *slot_config, const esp_vfs_fat_mount_config_t
*mount_config, sdmmc_card_t **out_card)`

Convenience function to get FAT filesystem on SD card registered in VFS.

This is an all-in-one function which does the following:

- initializes SDMMC driver or SPI driver with configuration in `host_config`
- initializes SD card with configuration in `slot_config`
- mounts FAT partition on SD card using FATFS library, with configuration in `mount_config`
- registers FATFS library with VFS, with prefix given by `base_prefix` variable

This function is intended to make example code more compact. For real world applications, developers should implement the logic of probing SD card, locating and mounting partition, and registering FATFS in VFS, with proper error checking and handling of exceptional conditions.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `esp_vfs_fat_sdmmc_mount` was already called
- `ESP_ERR_NO_MEM` if memory can not be allocated
- `ESP_FAIL` if partition can not be mounted
- other error codes from SDMMC or SPI drivers, SDMMC protocol, or FATFS drivers

Parameters

- `base_path`: path where partition should be registered (e.g. “/sdcard”)

- `host_config`: Pointer to structure describing SDMMC host. When using SDMMC peripheral, this structure can be initialized using `SDMMC_HOST_DEFAULT()` macro. When using SPI peripheral, this structure can be initialized using `SDSPI_HOST_DEFAULT()` macro.
- `slot_config`: Pointer to structure with slot configuration. For SDMMC peripheral, pass a pointer to `sdmmc_slot_config_t` structure initialized using `SDMMC_SLOT_CONFIG_DEFAULT`. For SPI peripheral, pass a pointer to `sdspi_slot_config_t` structure initialized using `SDSPI_SLOT_CONFIG_DEFAULT`.
- `mount_config`: pointer to structure with extra parameters for mounting FATFS
- `out_card`: if not NULL, pointer to the card information structure will be returned via this argument

struct esp_vfs_fat_mount_config_t

Configuration arguments for `esp_vfs_fat_sdmmc_mount` and `esp_vfs_fat_spiflash_mount` functions.

Public Members**bool format_if_mount_failed**

If FAT partition can not be mounted, and this parameter is true, create partition table and format the filesystem.

int max_files

Max number of open files.

esp_err_t esp_vfs_fat_sdmmc_unmount ()

Unmount FAT filesystem and release resources acquired using `esp_vfs_fat_sdmmc_mount`.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `esp_vfs_fat_sdmmc_mount` hasn't been called

FatFS disk IO layer

FatFs has been extended with an API to register disk IO driver at runtime.

Implementation of disk IO functions for SD/MMC cards is provided. It can be registered for the given FatFs drive number using `ff_diskio_register_sdmmc` function.

void ff_diskio_register (BYTE pdrv, const ff_diskio_impl_t *discio_impl)

Register or unregister diskio driver for given drive number.

When FATFS library calls one of `disk_xxx` functions for driver number `pdrv`, corresponding function in `discio_impl` for given `pdrv` will be called.

Parameters

- `pdrv`: drive number
- `discio_impl`: pointer to `ff_diskio_impl_t` structure with diskio functions or NULL to unregister and free previously registered drive

struct ff_diskio_impl_t

Structure of pointers to disk IO driver functions.

See FatFs documentation for details about these functions

Public Members

```
DSTATUS (*init) (BYTE pdrv)
    disk initialization function

DSTATUS (*status) (BYTE pdrv)
    disk status check function

DRESULT (*read) (BYTE pdrv, BYTE *buff, DWORD sector, UINT count)
    sector read function

DRESULT (*write) (BYTE pdrv, const BYTE *buff, DWORD sector, UINT count)
    sector write function

DRESULT (*ioctl) (BYTE pdrv, BYTE cmd, void *buff)
    function to get info about disk and do some misc operations

void ff_diskio_register_sdmmc (BYTE pdrv, sdmmc_card_t *card)
    Register SD/MMC diskio driver
```

Parameters

- pdrv: drive number
- card: pointer to *sdmmc_card_t* structure describing a card; card should be initialized before calling `f_mount`.

2.6.5 Wear Levelling APIs

Overview

Most of the flash devices and specially SPI flash devices that are used in ESP32 have sector based organization and have limited amount of erase/modification cycles per memory sector. To avoid situation when one sector reach the limit of erases when other sectors was used not often, we have made a component that avoid this situation. The wear levelling component share the amount of erases between all sectors in the memory without user interaction. The wear levelling component contains APIs related to reading, writing, erasing, memory mapping data in the external SPI flash through the partition component. It also has higher-level APIs which work with FAT filesystem defined in the *FAT filesystem*.

The wear levelling component, together with FAT FS component, works with FAT FS sector size 4096 bytes which is standard size of the flash devices. In this mode the component has best performance, but needs additional memory in the RAM. To save internal memory the component has two additional modes to work with sector size 512 bytes: Performance and Safety modes. In Performance mode by erase sector operation data will be stored to the RAM, sector will be erased and then data will be stored back to the flash. If by this operation power off situation will occur, the complete 4096 bytes will be lost. To prevent this the Safety mode was implemented. In safety mode the data will be first stored to the flash and after sector will be erased, will be stored back. If power off situation will occur, after power on, the data will be recovered. By default defined the sector size 512 bytes and Performance mode. To change these values please use the configuration menu.

The wear levelling component does not cache data in RAM. Write and erase functions modify flash directly, and flash contents is consistent when the function returns.

Wear Levelling access APIs

This is the set of APIs for working with data in flash:

- `wl_mount` mount wear levelling module for defined partition

- `wl_unmount` used to unmount levelling module
- `wl_erase_range` used to erase range of addresses in flash
- `wl_write` used to write data to the partition
- `wl_read` used to read data from the partition
- `wl_size` return size of available memory in bytes
- `wl_sector_size` returns size of one sector

Generally, try to avoid using the raw wear levelling functions in favor of filesystem-specific functions.

Memory Size

The memory size calculated in the wear Levelling module based on parameters of partition. The module use few sectors of flash for internal data.

See also

- [FAT Filesystem](#)
- [Partition Table documentation](#)

Application Example

An example which combines wear levelling driver with FATFS library is provided in `examples/storage/wear_levelling` directory. This example initializes the wear levelling driver, mounts FATFS partition, and writes and reads data from it using POSIX and C library APIs. See `README.md` file in the example directory for more information.

High level API Reference

Header Files

- `fatfs/src/esp_vfs_fat.h`

Functions

```
esp_err_t esp_vfs_fat_spiflash_mount(const char *base_path, const char *partition_label,  
                                     const esp_vfs_fat_mount_config_t *mount_config,  
                                     wl_handle_t *wl_handle)
```

Convenience function to initialize FAT filesystem in SPI flash and register it in VFS.

This is an all-in-one function which does the following:

- finds the partition with defined `partition_label`. Partition label should be configured in the partition table.
- initializes flash wear levelling library on top of the given partition
- mounts FAT partition using FATFS library on top of flash wear levelling library
- registers FATFS library with VFS, with prefix given by `base_prefix` variable

This function is intended to make example code more compact.

Return

- ESP_OK on success
- ESP_ERR_NOT_FOUND if the partition table does not contain FATFS partition with given label
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount was already called
- ESP_ERR_NO_MEM if memory can not be allocated
- ESP_FAIL if partition can not be mounted
- other error codes from wear levelling library, SPI flash driver, or FATFS drivers

Parameters

- base_path: path where FATFS partition should be mounted (e.g. “/spiflash”)
- partition_label: label of the partition which should be used
- mount_config: pointer to structure with extra parameters for mounting FATFS
- wl_handle: wear levelling driver handle

struct esp_vfs_fat_mount_config_t

Configuration arguments for esp_vfs_fat_sdmmc_mount and esp_vfs_fat_spiflash_mount functions.

Public Members

bool **format_if_mount_failed**

If FAT partition can not be mounted, and this parameter is true, create partition table and format the filesystem.

int **max_files**

Max number of open files.

esp_err_t **esp_vfs_fat_spiflash_unmount** (**const** char *base_path, *wl_handle_t* wl_handle)

Unmount FAT filesystem and release resources acquired using esp_vfs_fat_spiflash_mount.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount hasn't been called

Parameters

- base_path: path where partition should be registered (e.g. “/spiflash”)
- wl_handle: wear levelling driver handle returned by esp_vfs_fat_spiflash_mount

Mid level API Reference**Header File**

- wear_levelling/include/wear_levelling.h

Functions

`esp_err_t wl_mount (const esp_partition_t *partition, wl_handle_t *out_handle)`

Mount WL for defined partition.

Return

- ESP_OK, if the allocation was successfully;
- ESP_ERR_INVALID_ARG, if WL allocation was unsuccessful;
- ESP_ERR_NO_MEM, if there was no memory to allocate WL components;

Parameters

- `partition`: that will be used for access
- `out_handle`: handle of the WL instance

`esp_err_t wl_unmount (wl_handle_t handle)`

Unmount WL for defined partition.

Return

- ESP_OK, if the operation completed successfully;
- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL partition handle

`esp_err_t wl_erase_range (wl_handle_t handle, size_t start_addr, size_t size)`

Erase part of the WL storage.

Return

- ESP_OK, if the range was erased successfully;
- ESP_ERR_INVALID_ARG, if iterator or dst are NULL;
- ESP_ERR_INVALID_SIZE, if erase would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL handle that are related to the partition
- `start_addr`: Address where erase operation should start. Must be aligned to the result of function `wl_sector_size(...)`.
- `size`: Size of the range which should be erased, in bytes. Must be divisible by result of function `wl_sector_size(...)`.

`esp_err_t wl_write (wl_handle_t handle, size_t dest_addr, const void *src, size_t size)`

Write data to the WL storage.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using `wl_erase_range` function.

Note Prior to writing to WL storage, make sure it has been erased with `wl_erase_range` call.

Return

- ESP_OK, if data was written successfully;
- ESP_ERR_INVALID_ARG, if dst_offset exceeds partition size;
- ESP_ERR_INVALID_SIZE, if write would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

Parameters

- handle: WL handle that are related to the partition
- dest_addr: Address where the data should be written, relative to the beginning of the partition.
- src: Pointer to the source buffer. Pointer must be non-NUL 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-NUL and buffer must be at least ‘size’ bytes long.
- src_addr: Address of the data to be read, relative to the beginning of the partition.
- size: Size of data to be read, in bytes.

`size_t wl_size(wl_handle_t handle)`

Get size of the WL storage.

Return usable size, in bytes

Parameters

- handle: WL module handle that was initialized before

`size_t wl_sector_size(wl_handle_t handle)`

Get sector size of the WL instance.

Return sector size, in bytes

Parameters

- handle: WL module handle that was initialized before

Macros

`WL_INVALID_HANDLE`

Type Definitions

`typedef int32_t wl_handle_t`
wear levelling handle

2.6.6 SPIFFS Filesystem

Overview

SPIFFS is a file system intended for SPI NOR flash devices on embedded targets. It supports wear leveling, file system consistency checks and more.

Notes

- Presently, spiffs does not support directories. It produces a flat structure. If SPIFFS is mounted under /spiffs creating a file with path /spiffs/tmp/myfile.txt will create a file called /tmp/myfile.txt in SPIFFS, instead of myfile.txt under directory /spiffs/tmp.
- It is not a realtime stack. One write operation might last much longer than another.
- Presently, it does not detect or handle bad blocks.

Tools

Host-Side tools for creating SPIFS partition images exist and one such tool is `mkspiffs`. You can use it to create image from a given folder and then flash that image with `esptool.py`

To do that you need to obtain some parameters:

- Block Size: 4096 (standard for SPI Flash)
- Page Size: 256 (standard for SPI Flash)
- Image Size: Size of the partition in bytes (can be obtained from partition table)
- Partition Offset: Starting address of the partition (can be obtained from partition table)

To pack a folder into 1 Megabyte image:

```
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.

2.7 System API

2.7.1 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 heaps_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 heaps_caps_malloc() or heaps_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.

`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(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.

Call `heap_caps_check_integrity(MALLOC_CAP_INVALID, print_errors)` to check all regions’ heaps.

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.

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 (IDs are not currently used)

`MALLOC_CAP_PID3`

Memory must be mapped to PID3 memory space (IDs are not currently used)

`MALLOC_CAP_PID4`

Memory must be mapped to PID4 memory space (IDs are not currently used)

`MALLOC_CAP_PID5`

Memory must be mapped to PID5 memory space (IDs are not currently used)

`MALLOC_CAP_PID6`

Memory must be mapped to PID6 memory space (IDs are not currently used)

`MALLOC_CAP_PID7`

Memory must be mapped to PID7 memory space (IDs are not currently used)

`MALLOC_CAP_SPISRAM`

Memory must be in SPI SRAM.

`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_nono_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_nono_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.

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.

When the heap is first registered, the associated lock is NULL.

Parameters

- `heap`: Handle to a registered heap.
- `lock`: Optional pointer to a locking structure to associate with this heap.

```
void multi_heap_dump (multi_heap_handle_t heap)
```

Dump heap information to stdout.

For debugging purposes, this function dumps information about every block in the heap to stdout.

Parameters

- `heap`: Handle to a registered heap.

```
bool multi_heap_check (multi_heap_handle_t heap, bool print_errors)
```

Check heap integrity.

Walks the heap and checks all heap data structures are valid. If any errors are detected, an error-specific message can be optionally printed to stderr. Print behaviour can be overridden at compile time by defining MULTI_CHECK_FAIL_PRINTF in multi_heap_platform.h.

Return true if heap is valid, false otherwise.

Parameters

- `heap`: Handle to a registered heap.
- `print_errors`: If true, errors will be printed to stderr.

```
size_t multi_heap_free_size (multi_heap_handle_t heap)
```

Return free heap size.

Returns the number of bytes available in the heap.

Equivalent to the total_free_bytes member returned by multi_heap_get_heap_info().

Note that the heap may be fragmented, so the actual maximum size for a single malloc() may be lower. To know this size, see the largest_free_block member returned by multi_heap_get_heap_info().

Return Number of free bytes.

Parameters

- `heap`: Handle to a registered heap.

```
size_t multi_heap_minimum_free_size (multi_heap_handle_t heap)
```

Return the lifetime minimum free heap size.

Equivalent to the minimum_free_bytes member returned by multi_get_heap_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_get_heap_info`.

Public Members

`size_t total_free_bytes`

Total free bytes in the heap. Equivalent to `multi_free_heap_size()`.

`size_t total_allocated_bytes`

Total bytes allocated to data in the heap.

`size_t largest_free_block`

Size of largest free block in the heap. This is the largest malloc-able size.

`size_t minimum_free_bytes`

Lifetime minimum free heap size. Equivalent to `multi_minimum_free_heap_size()`.

`size_t allocated_blocks`

Number of (variable size) blocks allocated in the heap.

`size_t free_blocks`

Number of (variable size) free blocks in the heap.

`size_t total_blocks`

Total number of (variable size) blocks in the heap.

Type Definitions

`typedef struct multi_heap_info *multi_heap_handle_t`

Opaque handle to a registered heap.

2.7.2 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 `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 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.

It's also possible to manually check heap integrity by calling the `heap_caps_check_integrity()` function (see below). This function checks all of requested heap memory for integrity, and can be used even if assertions are disabled.

Configuration

In `make menuconfig`, under Component config there is a menu Heap memory debugging. The setting Heap corruption detection can be set to one of three levels:

Basic (no poisoning)

This is the default level. No special heap corruption features are enabled, but checks will fail if any of the heap's internal data structures are 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 (ie the same memory is freed twice).

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 the allocated buffer, the canary bytes will be corrupted and the integrity check will fail.

“Basic” heap corruption checks can also detect out of bounds writes, but this setting is more precise as even a single byte overrun will always be detected. With Basic heap checks, the number of overrun bytes before a failure is detected will depend on the properties of the heap.

Similar to other heap checks, these “canary bytes” are checked via assertion whenever memory is freed and can also be checked manually via `heap_caps_check_integrity()`.

This level increases memory usage, each individual allocation will use 9 to 12 additional bytes of memory (depending on alignment).

Comprehensive

This level incorporates the “light impact” detection features plus additional checks for uninitialized-access and use-after-free bugs. In this mode, all freshly allocated memory is filled with the pattern 0xCE, and all freed memory is filled with the pattern 0xFE.

If an application crashes reading/writing an address related to 0xCECECECE when this setting is enabled, 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 reading/writing an address related to 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 the IDF heap allocator fails because the pattern 0xFEFEFEFE was not found in freed memory then this indicates the app has a use-after-free bug where it is writing to memory which has already been freed.

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.)

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:

- If you can find the address (in memory) which is being corrupted, you can set a watchpoint on this address via JTAG to 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. A fatal exception will occur when the watchpoint triggers. For example `esp_set_watchpoint(0, (void *)addr, 4, ESP_WATCHPOINT_STORE)`. Note that the watchpoint is 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 allocate the memory address(es) immediately before the address which is being corrupted. There is a strong chance this is the code which overflows the buffer.

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 Enable 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 made. Is different for CPU 0 vs CPU 1.
- caller 0x... gives the call stack of the call to malloc()/free(), as a list of PC addresses. These can be decoded to source files and line numbers, as shown above.

The depth of the call stack recorded for each trace entry can be configured in `make menuconfig`, under Heap Memory Debugging -> Enable heap tracing -> Heap tracing stack depth. Up to 10 stack frames can be recorded for each allocation (the default is 2). Each additional stack frame increases the memory usage of each `heap_trace_record_t` record by eight bytes.

Finally, the total number of ‘leaked’ bytes (bytes allocated but not freed while trace was running) is printed, and the total number of allocations this represents.

A warning will be printed if the trace buffer was not large enough to hold all the allocations which happened. If you see this warning, consider either shortening the tracing period or increasing the number of records in the trace buffer.

Performance Impact

Enabling heap tracing in `menuconfig` increases the code size of your program, and has a very small negative impact on performance of heap allocation/free operations even when heap tracing is not running.

When heap tracing is running, heap allocation/free operations are substantially slower than when heap tracing is stopped. Increasing the depth of stack frames recorded for each allocation (see above) will also increase this performance impact.

False-Positive Memory Leaks

Not everything printed by `heap_trace_dump()` is necessarily a memory leak. Among things which may show up here, but are not memory leaks:

- Any memory which is allocated after `heap_trace_start()` but then freed after `heap_trace_stop()` will appear in the leak dump.
- Allocations may be made by other tasks in the system. Depending on the timing of these tasks, it’s quite possible this memory is freed after `heap_trace_stop()` is called.

- The first time a task uses stdio - for example, when it calls printf() - a lock (RTOS mutex semaphore) is allocated by the libc. This allocation lasts until the task is deleted.
- The Bluetooth, WiFi, and TCP/IP libraries will allocate heap memory buffers to handle incoming or outgoing data. These memory buffers are usually short lived, but some may be shown in the heap leak trace if the data was received/transmitted by the lower levels of the network while the leak trace was running.
- TCP connections will continue to use some memory after they are closed, because of the TIME_WAIT state. After the TIME_WAIT period has completed, this memory will be freed.

One way to differentiate between “real” and “false positive” memory leaks is to call the suspect code multiple times while tracing is running, and look for patterns (multiple matching allocations) in the heap trace output.

API Reference - Heap Tracing

Header File

- [heap/include/esp_heap_trace.h](#)

Functions

`esp_err_t heap_trace_init_standalone(heap_trace_record_t *record_buffer, size_t num_records)`

Initialise heap tracing in standalone mode.

This function must be called before any other heap tracing functions.

Note Standalone mode is the only mode currently supported.

To disable heap tracing and allow the buffer to be freed, stop tracing and then call `heap_trace_init_standalone(NULL, 0);`

Return

- `ESP_ERR_NOT_SUPPORTED` Project was compiled without heap tracing enabled in menuconfig.
- `ESP_ERR_INVALID_STATE` Heap tracing is currently in progress.
- `ESP_OK` Heap tracing initialised successfully.

Parameters

- `record_buffer`: Provide a buffer to use for heap trace data. Must remain valid any time heap tracing is enabled, meaning it must be allocated from internal memory not in PSRAM.
- `num_records`: Size of the heap trace buffer, as number of record structures.

`esp_err_t heap_trace_start(heap_trace_mode_t mode)`

Start heap tracing. All heap allocations & frees will be traced, until `heap_trace_stop()` is called.

Note `heap_trace_init_standalone()` must be called to provide a valid buffer, before this function is called.

Note Calling this function while heap tracing is running will reset the heap trace state and continue tracing.

Return

- `ESP_ERR_NOT_SUPPORTED` Project was compiled without heap tracing enabled in menuconfig.
- `ESP_ERR_INVALID_STATE` A non-zero-length buffer has not been set via `heap_trace_init_standalone()`.

- ESP_OK Tracing is started.

Parameters

- mode: Mode for tracing.
 - HEAP_TRACE_ALL means all heap allocations and frees are traced.
 - HEAP_TRACE_LEAKS means only suspected memory leaks are traced. (When memory is freed, the record is removed from the trace buffer.)

`esp_err_t heap_trace_stop(void)`

Stop heap tracing.

Return

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was not in progress.
- ESP_OK Heap tracing stopped..

`esp_err_t heap_trace_resume(void)`

Resume heap tracing which was previously stopped.

Unlike `heap_trace_start()`, this function does not clear the buffer of any pre-existing trace records.

The heap trace mode is the same as when `heap_trace_start()` was last called (or HEAP_TRACE_ALL if `heap_trace_start()` was never called).

Return

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was already started.
- ESP_OK Heap tracing resumed.

`size_t heap_trace_get_count(void)`

Return number of records in the heap trace buffer.

It is safe to call this function while heap tracing is running.

`esp_err_t heap_trace_get(size_t index, heap_trace_record_t *record)`

Return a raw record from the heap trace buffer.

Note It is safe to call this function while heap tracing is running, however in HEAP_TRACE_LEAK mode record indexing may skip entries unless heap tracing is stopped first.

Return

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was not initialised.
- ESP_ERR_INVALID_ARG Index is out of bounds for current heap trace record count.
- ESP_OK Record returned successfully.

Parameters

- index: Index (zero-based) of the record to return.
- record: Record where the heap trace record will be copied.

```
void heap_trace_dump(void)  
    Dump heap trace record data to stdout.
```

Note It is safe to call this function while heap tracing is running, however in HEAP_TRACE_LEAK mode the dump may skip entries unless heap tracing is stopped first.

Structures

```
struct heap_trace_record_t  
    Trace record data type. Stores information about an allocated region of memory.
```

Public Members

```
uint32_t ccount  
    CCOUNT of the CPU when the allocation was made. LSB (bit value 1) is the CPU number (0 or 1). */.
```

```
void *address  
    Address which was allocated.
```

```
size_t size  
    Size of the allocation.
```

```
void *allocated_by[CONFIG_HEAP_TRACING_STACK_DEPTH]  
    Call stack of the caller which allocated the memory.
```

```
void *freed_by[CONFIG_HEAP_TRACING_STACK_DEPTH]  
    Call stack of the caller which freed the memory (all zero if not freed.)
```

Macros

```
CONFIG_HEAP_TRACING_STACK_DEPTH
```

Enumerations

```
enum heap_trace_mode_t
```

Values:

```
HEAP_TRACE_ALL
```

```
HEAP_TRACE_LEAKS
```

2.7.3 Interrupt allocation

Overview

The ESP32 has two cores, with 32 interrupts each. Each interrupt has a certain priority level, most (but not all) interrupts are connected to the interrupt mux. Because there are more interrupt sources than interrupts, sometimes it makes sense to share an interrupt in multiple drivers. The esp_intr_alloc abstraction exists to hide all these implementation details.

A driver can allocate an interrupt for a certain peripheral by calling esp_intr_alloc (or esp_intr_alloc_sintrstatus). It can use the flags passed to this function to set the type of interrupt allocated, specifying a specific level or trigger

method. The interrupt allocation code will then find an applicable interrupt, use the interrupt mux to hook it up to the peripheral, and install the given interrupt handler and ISR to it.

This code has two different types of interrupts it handles differently: Shared interrupts and non-shared interrupts. The simplest of the two are non-shared interrupts: a separate interrupt is allocated per `esp_intr_alloc` call and this interrupt is solely used for the peripheral attached to it, with only one ISR that will get called. Shared interrupts can have multiple peripherals triggering it, with multiple ISRs being called when one of the peripherals attached signals an interrupt. Thus, ISRs that are intended for shared interrupts should check the interrupt status of the peripheral they service in order to see if any action is required.

Non-shared interrupts can be either level- or edge-triggered. Shared interrupts can only be level interrupts (because of the chance of missed interrupts when edge interrupts are used.) (The logic behind this: DevA and DevB share an int. DevB signals an int. Int line goes high. ISR handler calls code for DevA -> does nothing. ISR handler calls code for DevB, but while doing that, DevA signals an int. ISR DevB is done, clears int for DevB, exits interrupt code. Now an interrupt for DevA is still pending, but because the int line never went low (DevA kept it high even when the int for DevB was cleared) the interrupt is never serviced.)

Multicore issues

Peripherals that can generate interrupts can be divided in two types:

- External peripherals, within the ESP32 but outside the Xtensa cores themselves. Most ESP32 peripherals are of this type.
- Internal peripherals, part of the Xtensa CPU cores themselves.

Interrupt handling differs slightly between these two types of peripherals.

Internal peripheral interrupts

Each Xtensa CPU core has its own set of six internal peripherals:

- Three timer comparators
- A performance monitor
- Two software interrupts.

Internal interrupt sources are defined in `esp_intr_alloc.h` as `ETS_INTERNAL_*_INTR_SOURCE`.

These peripherals can only be configured from the core they are associated with. When generating an interrupt, the interrupt they generate is hard-wired to their associated core; it's not possible to have e.g. an internal timer comparator of one core generate an interrupt on another core. That is why these sources can only be managed using a task running on that specific core. Internal interrupt sources are still allocatable using `esp_intr_alloc` as normal, but they cannot be shared and will always have a fixed interrupt level (namely, the one associated in hardware with the peripheral).

External Peripheral Interrupts

The remaining interrupt sources are from external peripherals. These are defined in `soc/soc.h` as `ETS_*_INTR_SOURCE`.

Non-internal interrupt slots in both CPU cores are wired to an interrupt multiplexer, which can be used to route any external interrupt source to any of these interrupt slots.

- Allocating an external interrupt will always allocate it on the core that does the allocation.
- Freeing an external interrupt must always happen on the same core it was allocated on.

- Disabling and enabling external interrupts from another core is allowed.
- Multiple external interrupt sources can share an interrupt slot by passing `ESP_INTR_FLAG_SHARED` as a flag to `esp_intr_alloc()`.

Care should be taken when calling `esp_intr_alloc()` from a task which is not pinned to a core. During task switching, these tasks can migrate between cores. Therefore it is impossible to tell which CPU the interrupt is allocated on, which makes it difficult to free the interrupt handle and may also cause debugging difficulties. It is advised to use `xTaskCreatePinnedToCore()` with a specific CoreID argument to create tasks that will allocate interrupts. In the case of internal interrupt sources, this is required.

IRAM-Safe Interrupt Handlers

The `ESP_INTR_FLAG_IRAM` flag registers an interrupt handler that always runs from IRAM (and reads all its data from DRAM), and therefore does not need to be disabled during flash erase and write operations.

This is useful for interrupts which need a guaranteed minimum execution latency, as flash write and erase operations can be slow (erases can take tens or hundreds of milliseconds to complete).

It can also be useful to keep an interrupt handler in IRAM if it is called very frequently, to avoid flash cache misses.

Refer to the [SPI flash API documentation](#) for more details.

Multiple Handlers Sharing A Source

Several handlers can be assigned to a same source, given that all handlers are allocated using the `ESP_INTR_FLAG_SHARED` flag. They'll be all allocated to the interrupt, which the source is attached to, and called sequentially when the source is active. The handlers can be disabled and freed individually. The source is attached to the interrupt (enabled), if one or more handlers are enabled, otherwise detached. A handler will never be called when disabled, while **its source may still be triggered** if any one of its handler enabled.

Sources attached to non-shared interrupt do not support this feature.

Though the framework support this feature, you have to use it *very carefully*. There usually exist 2 ways to stop a interrupt from being triggered: *disable the source or mask peripheral interrupt status*. IDF only handles the enabling and disabling of the source itself, leaving status and mask bits to be handled by users. **Status bits should always be masked before the handler responsible for it is disabled, or the status should be handled in other enabled interrupt properly**. You may leave some status bits unhandled if you just disable one of all the handlers without mask the status bits, which causes the interrupt being triggered infinitely, and finally a system crash.

API Reference

Header File

- `esp32/include/esp_intr_alloc.h`

Functions

`esp_err_t esp_intr_mark_shared(int intno, int cpu, bool is_in_iram)`

Mark an interrupt as a shared interrupt.

This will mark a certain interrupt on the specified CPU as an interrupt that can be used to hook shared interrupt handlers to.

Return `ESP_ERR_INVALID_ARG` if `cpu` or `intno` is invalid `ESP_OK` otherwise

Parameters

- `intno`: The number of the interrupt (0-31)
- `cpu`: CPU on which the interrupt should be marked as shared (0 or 1)
- `is_in_iram`: Shared interrupt is for handlers that reside in IRAM and the int can be left enabled while the flash cache is disabled.

`esp_err_t esp_intr_reserve(int intno, int cpu)`

Reserve an interrupt to be used outside of this framework.

This will mark a certain interrupt on the specified CPU as reserved, not to be allocated for any reason.

Return `ESP_ERR_INVALID_ARG` if `cpu` or `intno` is invalid `ESP_OK` otherwise

Parameters

- `intno`: The number of the interrupt (0-31)
- `cpu`: CPU on which the interrupt should be marked as shared (0 or 1)

`esp_err_t esp_intr_alloc(int source, int flags, intr_handler_t handler, void *arg, intr_handle_t *ret_handle)`

Allocate an interrupt with the given parameters.

This finds an interrupt that matches the restrictions as given in the `flags` parameter, maps the given interrupt source to it and hooks up the given interrupt handler (with optional argument) as well. If needed, it can return a handle for the interrupt as well.

The interrupt will always be allocated on the core that runs this function.

If `ESP_INTR_FLAG_IRAM` flag is used, and handler address is not in IRAM or `RTC_FAST_MEM`, then `ESP_ERR_INVALID_ARG` is returned.

Return `ESP_ERR_INVALID_ARG` if the combination of arguments is invalid. `ESP_ERR_NOT_FOUND` No free interrupt found with the specified flags `ESP_OK` otherwise

Parameters

- `source`: The interrupt source. One of the `ETS_*_INTR_SOURCE` interrupt mux sources, as defined in `soc/soc.h`, or one of the internal `ETS_INTERNAL_*_INTR_SOURCE` sources as defined in this header.
- `flags`: An ORred mask of the `ESP_INTR_FLAG_*` defines. These restrict the choice of interrupts that this routine can choose from. If this value is 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is `ESP_INTR_FLAG_SHARED`, it will allocate a shared interrupt of level 1. Setting `ESP_INTR_FLAG_INTRDISABLED` will return from this function with the interrupt disabled.
- `handler`: The interrupt handler. Must be `NULL` when an interrupt of level >3 is requested, because these types of interrupts aren't C-callable.
- `arg`: Optional argument for passed to the interrupt handler
- `ret_handle`: Pointer to an `intr_handle_t` to store a handle that can later be used to request details or free the interrupt. Can be `NULL` if no handle is required.

`esp_err_t esp_intr_alloc_intrstatus(int source, int flags, uint32_t intrstatusreg, uint32_t intrstatusmask, intr_handler_t handler, void *arg, intr_handle_t *ret_handle)`

Allocate an interrupt with the given parameters.

This essentially does the same as esp_intr_alloc, but allows specifying a register and mask combo. For shared interrupts, the handler is only called if a read from the specified register, ANDed with the mask, returns non-zero. By passing an interrupt status register address and a fitting mask, this can be used to accelerate interrupt handling in the case a shared interrupt is triggered; by checking the interrupt statuses first, the code can decide which ISRs can be skipped

Return ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_ERR_NOT_FOUND No free interrupt found with the specified flags ESP_OK otherwise

Parameters

- **source:** The interrupt source. One of the ETS_*_INTR_SOURCE interrupt mux sources, as defined in soc/soc.h, or one of the internal ETS_INTERNAL_*_INTR_SOURCE sources as defined in this header.
- **flags:** An ORred mask of the ESP_INTR_FLAG_* defines. These restrict the choice of interrupts that this routine can choose from. If this value is 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is ESP_INTR_FLAG_SHARED, it will allocate a shared interrupt of level 1. Setting ESP_INTR_FLAG_INTRDISABLED will return from this function with the interrupt disabled.
- **intrstatusreg:** The address of an interrupt status register
- **intrstatusmask:** A mask. If a read of address intrstatusreg has any of the bits that are 1 in the mask set, the ISR will be called. If not, it will be skipped.
- **handler:** The interrupt handler. Must be NULL when an interrupt of level >3 is requested, because these types of interrupts aren't C-callable.
- **arg:** Optional argument for passed to the interrupt handler
- **ret_handle:** Pointer to an intr_handle_t to store a handle that can later be used to request details or free the interrupt. Can be NULL if no handle is required.

esp_err_t **esp_intr_free** (*intr_handle_t handle*)

Disable and free an interrupt.

Use an interrupt handle to disable the interrupt and release the resources associated with it.

Note When the handler shares its source with other handlers, the interrupt status bits it's responsible for should be managed properly before freeing it. see esp_intr_disable for more details.

Return ESP_ERR_INVALID_ARG if handle is invalid, or esp_intr_free runs on another core than where the interrupt is allocated on. ESP_OK otherwise

Parameters

- **handle:** The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

int **esp_intr_get_cpu** (*intr_handle_t handle*)

Get CPU number an interrupt is tied to.

Return The core number where the interrupt is allocated

Parameters

- **handle:** The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

int **esp_intr_get_intno** (*intr_handle_t handle*)

Get the allocated interrupt for a certain handle.

Return The interrupt number

Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

`esp_err_t esp_intr_disable (intr_handle_t handle)`

Disable the interrupt associated with the handle.

Note

1. For local interrupts (ESP_INTERNAL_* sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.
2. When several handlers sharing a same interrupt source, interrupt status bits, which are handled in the handler to be disabled, should be masked before the disabling, or handled in other enabled interrupts properly. Miss of interrupt status handling will cause infinite interrupt calls and finally system crash.

Return `ESP_ERR_INVALID_ARG` if the combination of arguments is invalid. `ESP_OK` otherwise

Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

`esp_err_t esp_intr_enable (intr_handle_t handle)`

Ensable the interrupt associated with the handle.

Note For local interrupts (ESP_INTERNAL_* sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.

Return `ESP_ERR_INVALID_ARG` if the combination of arguments is invalid. `ESP_OK` otherwise

Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

`void esp_intr_noniram_disable ()`

Disable interrupts that aren't specifically marked as running from IRAM.

`void esp_intr_noniram_enable ()`

Re-enable interrupts disabled by `esp_intr_noniram_disable`.

Macros

ESP_INTR_FLAG_LEVEL1

Interrupt allocation flags.

These flags can be used to specify which interrupt qualities the code calling `esp_intr_alloc*` needs. Accept a Level 1 interrupt vector

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.

ESP_INTR_FLAG_SHARED

Interrupt can be shared between ISRs.

ESP_INTR_FLAG_EDGE

Edge-triggered interrupt.

ESP_INTR_FLAG_IRAM

ISR can be called if cache is disabled.

ESP_INTR_FLAG_INTRDISABLED

Return with this interrupt disabled.

ESP_INTR_FLAG_LOWMED

Low and medium prio interrupts. These can be handled in C.

ESP_INTR_FLAG_HIGH

High level interrupts. Need to be handled in assembly.

ESP_INTR_FLAG_LEVELMASK

Mask for all level flags.

ETS_INTERNAL_TIMER0_INTR_SOURCE

Xtensa timer 0 interrupt source.

The esp_intr_alloc* functions can allocate an int for all ETS_*_INTR_SOURCE interrupt sources that are routed through the interrupt mux. Apart from these sources, each core also has some internal sources that do not pass through the interrupt mux. To allocate an interrupt for these sources, pass these pseudo-sources to the functions.

ETS_INTERNAL_TIMER1_INTR_SOURCE

Xtensa timer 1 interrupt source.

ETS_INTERNAL_TIMER2_INTR_SOURCE

Xtensa timer 2 interrupt source.

ETS_INTERNAL_SW0_INTR_SOURCE

Software int source 1.

ETS_INTERNAL_SW1_INTR_SOURCE

Software int source 2.

ETS_INTERNAL_PROFILING_INTR_SOURCE

Int source for profiling.

ETS_INTERNAL_INTR_SOURCE_OFF

Type Definitions

```
typedef void (*intr_handler_t)(void *arg)
typedef struct intr_handle_data_t intr_handle_data_t
typedef intr_handle_data_t *intr_handle_t
```

2.7.4 Watchdogs

Overview

Esp-idf has support for two types of watchdogs: a task watchdog as well as an interrupt watchdog. Both can be enabled using `make menuconfig` and selecting the appropriate options.

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

Any tasks can elect to be watched by the task watchdog. If such a task does not feed the watchdog within the time specified by the task watchdog timeout (which is configurable using `make menuconfig`), the watchdog will print out a warning with information about which processes are running on the ESP32 CPUs and which processes failed to feed the watchdog.

By default, the task watchdog watches the idle tasks. The usual cause of idle tasks not feeding the watchdog is a higher-priority process looping without yielding to the lower-priority processes, and can be an indicator of badly-written code that spinloops on a peripheral or a task that is stuck in an infinite loop.

Other task can elect to be watched by the task watchdog by calling `esp_task_wdt_feed()`. Calling this routine for the first time will register the task to the task watchdog; calling it subsequent times will feed the watchdog. If a task does not want to be watched anymore (e.g. because it is finished and will call `vTaskDelete()` on itself), it needs to call `esp_task_wdt_delete()`.

The task watchdog is built around the hardware watchdog in timer group 0. If this watchdog for some reason cannot execute the interrupt handler that prints the task data (e.g. because IRAM is overwritten by garbage or interrupts are disabled entirely) it will hard-reset the SOC.

JTAG and watchdogs

While debugging using OpenOCD, if the CPUs are halted the watchdogs will keep running, eventually resetting the CPU. This makes it very hard to debug code; that is why the OpenOCD config will disable both watchdogs on startup. This does mean that you will not get any warnings or panics from either the task or interrupt watchdog when the ESP32 is connected to OpenOCD via JTAG.

API Reference

Header Files

- `esp32/include/esp_int_wdt.h`
- `esp32/include/esp_task_wdt.h`

Functions

`void esp_int_wdt_init()`

Initialize the interrupt watchdog. This is called in the init code if the interrupt watchdog is enabled in menuconfig.

`void esp_task_wdt_init()`

Initialize the task watchdog. This is called in the init code, if the task watchdog is enabled in menuconfig.

`void esp_task_wdt_feed()`

Feed the watchdog. After the first feeding session, the watchdog will expect the calling task to keep feeding the watchdog until `task_wdt_delete()` is called.

`void esp_task_wdt_delete()`

Delete the watchdog for the current task.

2.7.5 Over The Air Updates (OTA)

OTA Process Overview

The OTA update mechanism allows a device to update itself based on data received while the normal firmware is running (for example, over WiFi or Bluetooth.)

OTA requires configuring the *Partition Table* of the device with at least two “OTA app slot” partitions (ie `ota_0` and `ota_1`) and an “OTA Data Partition”.

The OTA operation functions write a new app firmware image to whichever OTA app slot is not currently being used for booting. Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot.

OTA Data Partition

An OTA data partition (type `data`, subtype `ota`) must be included in the *Partition Table* of any project which uses the OTA functions.

For factory boot settings, the OTA data partition should contain no data (all bytes erased to 0xFF). In this case the esp-idf software bootloader will boot the factory app if it is present in the the partition table. If no factory app is included in the partition table, the first available OTA slot (usually `ota_0`) is booted.

After the first OTA update, the OTA data partition is updated to specify which OTA app slot partition should be booted next.

The OTA data partition is two flash sectors (0x2000 bytes) in size, to prevent problems if there is a power failure while it is being written. Sectors are independently erased and written with matching data, and if they disagree a counter field is used to determine which sector was written more recently.

See also

- [Partition Table documentation](#)
- [Lower-Level SPI Flash/Partition API](#)

Application Example

End-to-end example of OTA firmware update workflow: `system/ota`.

API Reference

Header File

- `app_update/include/esp_ota_ops.h`

Functions

```
esp_err_t esp_ota_begin(const esp_partition_t *partition, size_t image_size, esp_ota_handle_t
                       *out_handle)
```

Commence an OTA update writing to the specified partition.

The specified partition is erased to the specified image size.

If image size is not yet known, pass `OTA_SIZE_UNKNOWN` which will cause the entire partition to be erased.

On success, this function allocates memory that remains in use until `esp_ota_end()` is called with the returned handle.

Return

- `ESP_OK`: OTA operation commenced successfully.
- `ESP_ERR_INVALID_ARG`: partition or out_handle arguments were NULL, or partition doesn't point to an OTA app partition.
- `ESP_ERR_NO_MEM`: Cannot allocate memory for OTA operation.
- `ESP_ERR_OTA_PARTITION_CONFLICT`: Partition holds the currently running firmware, cannot update in place.
- `ESP_ERR_NOT_FOUND`: Partition argument not found in partition table.
- `ESP_ERR_OTA_SELECT_INFO_INVALID`: The OTA data partition contains invalid data.
- `ESP_ERR_INVALID_SIZE`: Partition doesn't fit in configured flash size.
- `ESP_ERR_FLASH_OP_TIMEOUT` or `ESP_ERR_FLASH_OP_FAIL`: Flash write failed.

Parameters

- `partition`: Pointer to info for partition which will receive the OTA update. Required.
- `image_size`: Size of new OTA app image. Partition will be erased in order to receive this size of image. If 0 or `OTA_SIZE_UNKNOWN`, the entire partition is erased.
- `out_handle`: On success, returns a handle which should be used for subsequent `esp_ota_write()` and `esp_ota_end()` calls.

`esp_err_t esp_ota_write(esp_ota_handle_t handle, const void *data, size_t size)`

Write OTA update data to partition.

This function can be called multiple times as data is received during the OTA operation. Data is written sequentially to the partition.

Return

- `ESP_OK`: Data was written to flash successfully.
- `ESP_ERR_INVALID_ARG`: handle is invalid.
- `ESP_ERR_OTA_VALIDATE_FAILED`: First byte of image contains invalid app image magic byte.
- `ESP_ERR_FLASH_OP_TIMEOUT` or `ESP_ERR_FLASH_OP_FAIL`: Flash write failed.
- `ESP_ERR_OTA_SELECT_INFO_INVALID`: OTA data partition has invalid contents

Parameters

- `handle`: Handle obtained from `esp_ota_begin`
- `data`: Data buffer to write
- `size`: Size of data buffer in bytes.

`esp_err_t esp_ota_end(esp_ota_handle_t handle)`

Finish OTA update and validate newly written app image.

Note After calling `esp_ota_end()`, the handle is no longer valid and any memory associated with it is freed (regardless of result).

Return

- `ESP_OK`: Newly written OTA app image is valid.
- `ESP_ERR_NOT_FOUND`: OTA handle was not found.
- `ESP_ERR_INVALID_ARG`: Handle was never written to.
- `ESP_ERR_OTA_VALIDATE_FAILED`: OTA image is invalid (either not a valid app image, or - if secure boot is enabled - signature failed to verify.)
- `ESP_ERR_INVALID_STATE`: If flash encryption is enabled, this result indicates an internal error writing the final encrypted bytes to flash.

Parameters

- `handle`: Handle obtained from `esp_ota_begin()`.

`esp_err_t esp_ota_set_boot_partition(const esp_partition_t *partition)`

Configure OTA data for a new boot partition.

Note If this function returns `ESP_OK`, calling `esp_restart()` will boot the newly configured app partition.

Return

- `ESP_OK`: OTA data updated, next reboot will use specified partition.
- `ESP_ERR_INVALID_ARG`: partition argument was NULL or didn't point to a valid OTA partition of type "app".
- `ESP_ERR_OTA_VALIDATE_FAILED`: Partition contained invalid app image. Also returned if secure boot is enabled and signature validation failed.

- ESP_ERR_NOT_FOUND: OTA data partition not found.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash erase or write failed.

Parameters

- `partition`: Pointer to info for partition containing app image to boot.

`const esp_partition_t *esp_ota_get_boot_partition(void)`

Get partition info of currently configured boot app.

If `esp_ota_set_boot_partition()` has been called, the partition which was set by that function will be returned.

If `esp_ota_set_boot_partition()` has not been called, the result is usually the same as `esp_ota_get_running_partition()`. The two results are not equal if the configured boot partition does not contain a valid app (meaning that the running partition will be an app that the bootloader chose via fallback).

If the OTA data partition is not present or not valid then the result is the first app partition found in the partition table. In priority order, this means: the factory app, the first OTA app slot, or the test app partition.

Note that there is no guarantee the returned partition is a valid app. Use `esp_image_load(ESP_IMAGE_VERIFY, ...)` to verify if the returned partition contains a bootable image.

Return Pointer to info for partition structure, or NULL if partition table is invalid or a flash read operation failed. Any returned pointer is valid for the lifetime of the application.

`const esp_partition_t *esp_ota_get_running_partition(void)`

Get partition info of currently running app.

This function is different to `esp_ota_get_boot_partition()` in that it ignores any change of selected boot partition caused by `esp_ota_set_boot_partition()`. Only the app whose code is currently running will have its partition information returned.

The partition returned by this function may also differ from `esp_ota_get_boot_partition()` if the configured boot partition is somehow invalid, and the bootloader fell back to a different app partition at boot.

Return Pointer to info for partition structure, or NULL if no partition is found or flash read operation failed. Returned pointer is valid for the lifetime of the application.

`const esp_partition_t *esp_ota_get_next_update_partition(const esp_partition_t *start_from)`

Return the next OTA app partition which should be written with a new firmware.

Call this function to find an OTA app partition which can be passed to `esp_ota_begin()`.

Finds next partition round-robin, starting from the current running partition.

Return Pointer to info for partition which should be updated next. NULL result indicates invalid OTA data partition, or that no eligible OTA app slot partition was found.

Parameters

- `start_from`: If set, treat this partition info as describing the current running partition. Can be NULL, in which case `esp_ota_get_running_partition()` is used to find the currently running partition. The result of this function is never the same as this argument.

Macros

`OTA_SIZE_UNKNOWN`

Used for esp_ota_begin() if new image size is unknown

`ESP_ERR_OTA_BASE`

Base error code for ota_ops api

`ESP_ERR_OTA_PARTITION_CONFLICT`

Error if request was to write or erase the current running partition

`ESP_ERR_OTA_SELECT_INFO_INVALID`

Error if OTA data partition contains invalid content

`ESP_ERR_OTA_VALIDATE_FAILED`

Error if OTA app image is invalid

Type Definitions

`typedef uint32_t esp_ota_handle_t`

Opaque handle for an application OTA update.

esp_ota_begin() returns a handle which is then used for subsequent calls to esp_ota_write() and esp_ota_end().

2.7.6 Sleep Modes

Overview

ESP32 is capable of light sleep and deep sleep power saving modes.

In light sleep mode, digital peripherals, most of the RAM, and CPUs are clock-gated, and supply voltage is reduced. Upon exit from light sleep, peripherals and CPUs resume operation, their internal state is preserved.

In deep sleep mode, CPUs, most of the RAM, and all the digital peripherals which are clocked from APB_CLK are powered off. The only parts of the chip which can still be powered on are: RTC controller, RTC peripherals (including ULP coprocessor), and RTC memories (slow and fast).

Wakeup from deep and light sleep modes can be done using several sources. These sources can be combined, in this case the chip will wake up when any one of the sources is triggered. Wakeup sources can be enabled using esp_sleep_enable_X_wakeup APIs. Next section describes these APIs in detail. Wakeup sources can be configured at any moment before entering light or deep sleep mode.

Additionally, the application can force specific powerdown modes for the RTC peripherals and RTC memories using esp_sleep_pd_config API.

Once wakeup sources are configured, application can enter sleep mode using esp_light_sleep_start or esp_deep_sleep_start APIs. At this point the hardware will be configured according to the requested wakeup sources, and RTC controller will either power down or power off the CPUs and digital peripherals.

WiFi/BT and sleep modes

In deep sleep mode, wireless peripherals are powered down. Before entering sleep mode, applications must disable WiFi and BT using appropriate calls (esp_bluedroid_disable, esp_bt_controller_disable, esp_wifi_stop).

WiFi can coexist with light sleep mode, allowing the chip to go into light sleep mode when there is no network activity, and waking up the chip from light sleep mode when required. However **APIs described in this section can not be used for that purpose.** `esp_light_sleep_start` forces the chip to enter light sleep mode, regardless of whether WiFi is active or not. Automatic entry into light sleep mode, coordinated with WiFi driver, will be supported using a separate set of APIs.

Wakeup sources

Timer

RTC controller has a built in timer which can be used to wake up the chip after a predefined amount of time. Time is specified at microsecond precision, but the actual resolution depends on the clock source selected for RTC SLOW_CLK. See chapter “Reset and Clock” of the ESP32 Technical Reference Manual for details about RTC clock options.

This wakeup mode doesn’t require RTC peripherals or RTC memories to be powered on during sleep.

The following function can be used to enable deep sleep wakeup using a timer.

`esp_err_t esp_sleep_enable_timer_wakeup(uint64_t time_in_us)`

Enable wakeup by timer.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if value is out of range (TBD)

Parameters

- `time_in_us`: time before wakeup, in microseconds

Touch pad

RTC IO module contains logic to trigger wakeup when a touch sensor interrupt occurs. You need to configure the touch pad interrupt before the chip starts deep sleep.

Revisions 0 and 1 of the ESP32 only support this wakeup mode when RTC peripherals are not forced to be powered on (i.e. `ESP_PD_DOMAIN_RTC_PERIPH` should be set to `ESP_PD_OPTION_AUTO`).

`esp_err_t esp_sleep_enable_touchpad_wakeup()`

Enable wakeup by touch sensor.

Note In revisions 0 and 1 of the ESP32, touch wakeup source can not be used when `RTC_PERIPH` power domain is forced to be powered on (`ESP_PD_OPTION_ON`) or when ext0 wakeup source is used.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if wakeup triggers conflict

External wakeup (ext0)

RTC IO module contains logic to trigger wakeup when one of RTC GPIOs is set to a predefined logic level. RTC IO is part of RTC peripherals power domain, so RTC peripherals will be kept powered on during deep sleep if this wakeup source is requested.

Because RTC IO module is enabled in this mode, internal pullup or pulldown resistors can also be used. They need to be configured by the application using `rtc_gpio_pullup_en` and `rtc_gpio_pulldown_en` functions, before calling `esp_sleep_start`.

In revisions 0 and 1 of the ESP32, this wakeup source is incompatible with ULP and touch wakeup sources.

Warning: After wake up from sleep, IO pad used for wakeup will be configured as RTC IO. Before using this pad as digital GPIO, reconfigure it using `rtc_gpio_deinit(gpio_num)` function.

`esp_err_t esp_sleep_enable_ext0_wakeup(gpio_num_t gpio_num, int level)`

Enable wakeup using a pin.

This function uses external wakeup feature of RTC_IO peripheral. It will work only if RTC peripherals are kept on during sleep.

This feature can monitor any pin which is an RTC IO. Once the pin transitions into the state given by level argument, the chip will be woken up.

Note This function does not modify pin configuration. The pin is configured in `esp_sleep_start`, immediately before entering sleep mode.

Note In revisions 0 and 1 of the ESP32, ext0 wakeup source can not be used together with touch or ULP wakeup sources.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if the selected GPIO is not an RTC GPIO, or the mode is invalid
- `ESP_ERR_INVALID_STATE` if wakeup triggers conflict

Parameters

- `gpio_num`: GPIO number used as wakeup source. Only GPIOs which 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 memory. RTC slow memory will be powered on during sleep if this wakeup mode is requested. RTC peripherals will be automatically powered on before ULP coprocessor starts running the program; once the program stops running, RTC peripherals are automatically powered down again.

Revisions 0 and 1 of the ESP32 only support this wakeup mode when RTC peripherals are not forced to be powered on (i.e. `ESP_PD_DOMAIN_RTC_PERIPH` should be set to `ESP_PD_OPTION_AUTO`).

The following function can be used to enable this wakeup mode:

```
esp_err_t esp_sleep_enable_ulp_wakeup()  
    Enable wakeup by ULP coprocessor.
```

Note In revisions 0 and 1 of the ESP32, ULP wakeup source can not be used when RTC_PERIPH power domain is forced to be powered on (ESP_PD_OPTION_ON) or when ext0 wakeup source is used.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if ULP co-processor is not enabled or if wakeup triggers conflict

Power-down of RTC peripherals and memories

By default, `esp_deep_sleep_start` and `esp_light_sleep_start` functions will power down all RTC power domains which are not needed by the enabled wakeup sources. To override this behaviour, `esp_sleep_pd_config` function is provided.

Note: in revision 0 of the ESP32, RTC fast memory will always be kept enabled in deep sleep, so that the deep sleep stub can run after reset. This can be overridden, if the application doesn't need clean reset behaviour after deep sleep.

If some variables in the program are placed into RTC slow memory (for example, using `RTC_DATA_ATTR` attribute), RTC slow memory will be kept powered on by default. This can be overridden using `esp_sleep_pd_config` function, if desired.

```
esp_err_t esp_sleep_pd_config(esp_sleep_pd_domain_t domain, esp_sleep_pd_option_t option)
```

Set power down mode for an RTC power domain in sleep mode.

If not set using this API, all power domains default to `ESP_PD_OPTION_AUTO`.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if either of the arguments is out of range

Parameters

- `domain`: power domain to configure
- `option`: power down option (`ESP_PD_OPTION_OFF`, `ESP_PD_OPTION_ON`, or `ESP_PD_OPTION_AUTO`)

enum esp_sleep_pd_domain_t

Power domains which can be powered down in sleep mode.

Values:

`ESP_PD_DOMAIN_RTC_PERIPH`

RTC IO, sensors and ULP co-processor.

`ESP_PD_DOMAIN_RTC_SLOW_MEM`

RTC slow memory.

`ESP_PD_DOMAIN_RTC_FAST_MEM`

RTC fast memory.

`ESP_PD_DOMAIN_MAX`

Number of domains.

enum esp_sleep_pd_option_t

Power down options.

Values:

ESP_PD_OPTION_OFF

Power down the power domain in sleep mode.

ESP_PD_OPTION_ON

Keep power domain enabled during sleep mode.

ESP_PD_OPTION_AUTO

Keep power domain enabled in sleep mode, if it is needed by one of the wakeup options. Otherwise power it down.

Entering light sleep

The following function can be used to enter light sleep once wakeup sources are configured. It is also possible to go into light sleep with no wakeup sources configured, in this case the chip will be in light sleep mode indefinitely, until external reset is applied.

esp_err_t esp_light_sleep_start()

Enter light sleep with the configured wakeup options.

Return

- ESP_OK on success (returned after wakeup)
- ESP_ERR_INVALID_STATE if WiFi or BT is not stopped

Entering deep sleep

The following function can be used to enter deep sleep once wakeup sources are configured. It is also possible to go into deep sleep with no wakeup sources configured, in this case the chip will be in deep sleep mode indefinitely, until external reset is applied.

void esp_deep_sleep_start()

Enter deep sleep with the configured wakeup options.

This function does not return.

Checking sleep wakeup cause

The following function can be used to check which wakeup source has triggered wakeup from sleep mode. For touch pad and ext1 wakeup sources, it is possible to identify pin or touch pad which has caused wakeup.

esp_sleep_wakeup_cause_t esp_sleep_get_wakeup_cause()

Get the source which caused wakeup from sleep.

Return wakeup cause, or ESP_DEEP_SLEEP_WAKEUP_UNDEFINED if reset happened for reason other than deep sleep wakeup

enum esp_sleep_wakeup_cause_t

Sleep wakeup cause.

Values:

ESP_SLEEP_WAKEUP_UNDEFINED

ESP_SLEEP_WAKEUP_EXT0

In case of deep sleep, reset was not caused by exit from deep sleep.

ESP_SLEEP_WAKEUP_EXT1

Wakeup caused by external signal using RTC_IO.

ESP_SLEEP_WAKEUP_TIMER

Wakeup caused by external signal using RTC_CNTL.

ESP_SLEEP_WAKEUP_TOUCHPAD

Wakeup caused by timer.

ESP_SLEEP_WAKEUP_ULP

Wakeup caused by touchpad.

`touch_pad_t esp_sleep_get_touchpad_wakeup_status()`

Get the touch pad which caused wakeup.

If wakeup was caused by another source, this function will return TOUCH_PAD_MAX;

Return touch pad which caused wakeup

`uint64_t esp_sleep_get_ext1_wakeup_status()`

Get the bit mask of GPIOs which caused wakeup (ext1)

If wakeup was caused by another source, this function will return 0.

Return bit mask, if GPIOOn caused wakeup, BIT(n) will be set

Application Example

Implementation of basic functionality of deep sleep is shown in [protocols/sntp](#) example, where ESP module is periodically waken up to retrieve time from NTP server.

More extensive example in [system/deep_sleep](#) illustrates usage of various deep sleep wakeup triggers and ULP coprocessor programming.

2.7.7 Logging library

Overview

Log library has two ways of managing log verbosity: compile time, set via menuconfig; and runtime, using `esp_log_level_set` function.

At compile time, filtering is done using `CONFIG_LOG_DEFAULT_LEVEL` macro, set via menuconfig. All logging statements for levels higher than `CONFIG_LOG_DEFAULT_LEVEL` will be removed by the preprocessor.

At run time, all logs below `CONFIG_LOG_DEFAULT_LEVEL` are enabled by default. `esp_log_level_set` function may be used to set logging level per module. Modules are identified by their tags, which are human-readable ASCII zero-terminated strings.

How to use this library

In each C file which uses logging functionality, define TAG variable like this:

```
static const char* TAG = "MyModule";
```

then use one of logging macros to produce output, e.g:

```
ESP_LOGW(TAG, "Baud rate error %.1f%. Requested: %d baud, actual: %d baud", error *  
→100, baud_req, baud_real);
```

Several macros are available for different verbosity levels:

- `ESP_LOGE` - error
- `ESP_LOGW` - warning
- `ESP_LOGI` - info
- `ESP_LOGD` - debug
- `ESP_LOGV` - verbose

Additionally there is an `_EARLY_` variant for each of these macros (e.g. `ESP_EARLY_LOGE`). These variants can run in startup code, before heap allocator and syscalls have been initialized. When compiling bootloader, normal `ESP_LOGx` macros fall back to the same implementation as `ESP_EARLY_LOGx` macros. So the only place where `ESP_EARLY_LOGx` have to be used explicitly is the early startup code, such as heap allocator initialization code.

(Note that such distinction would not have been necessary if we would have an `ets_vprintf` function in the ROM. Then it would be possible to switch implementation from `_EARLY_` version to normal version on the fly. Unfortunately, `ets_vprintf` in ROM has been inlined by the compiler into `ets_printf`, so it is not accessible outside.)

To override default verbosity level at file or component scope, define `LOG_LOCAL_LEVEL` macro. At file scope, define it before including `esp_log.h`, e.g.:

```
#define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE  
#include "esp_log.h"
```

At component scope, define it in component makefile:

```
CFLAGS += -D LOG_LOCAL_LEVEL=ESP_LOG_DEBUG
```

To configure logging output per module at runtime, add calls to `esp_log_level_set` function:

```
esp_log_level_set("*", ESP_LOG_ERROR);           // set all components to ERROR level  
esp_log_level_set("wifi", ESP_LOG_WARN);         // enable WARN logs from WiFi stack  
esp_log_level_set("dhcpc", ESP_LOG_INFO);          // enable INFO logs from DHCP client
```

Logging to Host via JTAG

By default logging library uses `vprintf`-like function to write formatted output to dedicated UART. With calling a simple API, all log output may be routed to JTAG instead, and make the logging several times faster. For details please refer to section [Logging to Host](#).

Application Example

Log library is commonly used by most of esp-idf components and examples. For demonstration of log functionality check `examples` folder of `espressif/esp-idf` repository, that among others, contains the following examples:

- `system/ota`
- `storage/sd_card`
- `protocols/https_request`

API Reference

Header File

- log/include/esp_log.h

Functions

void **esp_log_level_set** (**const** char *tag, *esp_log_level_t* level)

Set log level for given tag.

If logging for given component has already been enabled, changes previous setting.

Parameters

- tag: Tag of the log entries to enable. Must be a non-NUL zero terminated string. Value “*” resets log level for all tags to the given value.
- level: Selects log level to enable. Only logs at this and lower levels will be shown.

void **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.

Parameters

- func: 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 starts 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.

void **esp_log_buffer_hex** (**const** char *tag, **const** void *buffer, uint16_t buff_len)

Log a buffer of hex bytes at Info level.

Parameters

- tag: description tag
- buffer: Pointer to the buffer array
- buff_len: length of buffer in bytes

void **esp_log_buffer_char** (**const** char *tag, **const** void *buffer, uint16_t buff_len)
Log a buffer of characters at Info level. Buffer should contain only printable characters.

Parameters

- tag: description tag
- buffer: Pointer to the buffer array
- buff_len: length of buffer in bytes

Macros

LOG_COLOR_E

LOG_COLOR_W

LOG_COLOR_I

LOG_COLOR_D

LOG_COLOR_V

LOG_RESET_COLOR

LOG_FORMAT (letter, format)

LOG_LOCAL_LEVEL

ESP_EARLY_LOGE (tag, format, ...)

macro to output logs in startup code, before heap allocator and syscalls have been initialized. log at ESP_LOG_ERROR level.

See printf,ESP_LOGE

ESP_EARLY_LOGW (tag, format, ...)

macro to output logs in startup code at ESP_LOG_WARN level.

See ESP_EARLY_LOGE,ESP_LOGE,printf

ESP_EARLY_LOGI (tag, format, ...)

macro to output logs in startup code at ESP_LOG_INFO level.

See ESP_EARLY_LOGE,ESP_LOGE,printf

ESP_EARLY_LOGD (tag, format, ...)

macro to output logs in startup code at ESP_LOG_DEBUG level.

See ESP_EARLY_LOGE,ESP_LOGE,printf

ESP_EARLY_LOGV (tag, format, ...)

macro to output logs in startup code at ESP_LOG_VERBOSE level.

See `ESP_EARLY_LOGE`, `ESP_LOGE`, `printf`

`ESP_LOGE` (tag, format, ...)
`ESP_LOGW` (tag, format, ...)
`ESP_LOGI` (tag, format, ...)
`ESP_LOGD` (tag, format, ...)
`ESP_LOGV` (tag, format, ...)

Type Definitions

`typedef int (*vprintf_like_t)(const char *, va_list)`

Enumerations

`enum esp_log_level_t`

Log level.

Values:

`ESP_LOG_NONE`

No log output

`ESP_LOG_ERROR`

Critical errors, software module can not recover on its own

`ESP_LOG_WARN`

Error conditions from which recovery measures have been taken

`ESP_LOG_INFO`

Information messages which describe normal flow of events

`ESP_LOG_DEBUG`

Extra information which is not necessary for normal use (values, pointers, sizes, etc).

`ESP_LOG_VERBOSE`

Bigger chunks of debugging information, or frequent messages which can potentially flood the output.

2.7.8 Base MAC address

Overview

Several MAC addresses (universally administered by IEEE) are uniquely assigned to the networking interfaces (WiFi/BT/Ethernet). The final octet of each universally administered MAC address increases by one. Only the first one which is called base MAC address of them is stored in EFUSE or external storage, the others are generated from it. Here, ‘generate’ means adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address.

If the universally administered MAC addresses are not enough for all of the networking interfaces. Local administered MAC addresses which are derived from universally administered MAC addresses are assigned to the rest 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	18	[191:184]	1: useful. 0: useless
reserve	112	[183:72]	reserved
mac address	64	[71:8]	base MAC address
mac crc	8	[7:0]	crc of base MAC address

If using base MAC address stored in external storage, firstly get the base MAC address stored in external storage, then call API `esp_base_mac_addr_set()` to set the base MAC address for system to generate the MAC addresses used by the networking interfaces(WiFi/BT/Ethernet).

All of the steps must be done before initializing the networking interfaces(WiFi/BT/Ethernet). It is recommended to do it in `app_main()` which can be referenced in [system/base_mac_address](#).

Number of universally administered MAC address

If the number of universal MAC addresses is two, only two interfaces (WiFi station and Bluetooth) receive a universally administered MAC address. These are generated sequentially by adding 0 and 1 (respectively) to the base MAC address. The remaining two interfaces (WiFi softap and Ethernet) receive local MAC addresses. These are derived from the universal WiFi station and Bluetooth MAC addresses, respectively.

If the number of universal MAC addresses is four, all four interfaces (WiFi station, WiFi softap, Bluetooth and Ethernet) receive a universally administered MAC address. These are generated sequentially by adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address.

When using the default (Espressif-assigned) base MAC address, either setting can be used. When using a custom universal MAC address range, the correct setting will depend on the allocation of MAC addresses in this range (either 2 or 4 per device.)

API Reference

Header Files

- [esp32/include/esp_system.h](#)

Functions

`esp_err_t esp_base_mac_addr_set(uint8_t *mac)`

Set base MAC address with the MAC address which is stored in BLK3 of EFUSE or external storage e.g. flash and EEPROM.

Base MAC address is used to generate the MAC addresses used by the networking interfaces. If using base MAC address stored in BLK3 of EFUSE or external storage, call this API to set base MAC address with the MAC address which is stored in BLK3 of EFUSE or external storage before initializing WiFi/BT/Ethernet.

Return ESP_OK on success

Parameters

- `mac`: base MAC address, length: 6 bytes.

`esp_err_t esp_efuse_mac_get_custom(uint8_t *mac)`

Return base MAC address which was previously written to BLK3 of EFUSE.

Base MAC address is used to generate the MAC addresses used by the networking interfaces. This API returns the custom base MAC address which was previously written to BLK3 of EFUSE. Writing this EFUSE allows setting of a different (non-Espressif) base MAC address. It is also possible to store a custom base MAC address elsewhere, see `esp_base_mac_addr_set()` for details.

Return ESP_OK on success
ESP_ERR_INVALID_VERSION An invalid MAC version field was read from BLK3 of EFUSE
ESP_ERR_INVALID_CRC An invalid MAC CRC was read from BLK3 of EFUSE

Parameters

- `mac`: base MAC address, length: 6 bytes.

2.7.9 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. System behaviour analysis.
2. Lightweight logging to the host.
3. Collecting application specific data.

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-NUL on success, otherwise NULL.

Parameters

- `dest`: Indicates HW interface to send data.
- `size`: Size of data to write to trace buffer.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

`esp_err_t esp_apptrace_buffer_put(esp_apptrace_dest_t dest, uint8_t *ptr, uint32_t tmo)`

Indicates that the data in buffer are ready to be sent off. This function is a counterpart of and must be preceded by esp_apptrace_buffer_get.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- `dest`: Indicates HW interface to send data. Should be identical to the same parameter in call to esp_apptrace_buffer_get.
- `ptr`: Address of trace buffer to release. Should be the value returned by call to esp_apptrace_buffer_get.
- `tmo`: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

`esp_err_t esp_apptrace_write(esp_apptrace_dest_t dest, const void *data, uint32_t size, uint32_t tmo)`

Writes data to trace buffer.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- `dest`: Indicates HW interface to send data.

- **data**: Address of data to write to trace buffer.
- **size**: Size of data to write to trace buffer.
- **tmo**: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

```
int esp_apptrace_vprintf_to(esp_apptrace_dest_t dest, uint32_t tmo, const char *fmt, va_list ap)
```

vprintf-like function to sent log messages to host via specified HW interface.

Return Number of bytes written.

Parameters

- **dest**: Indicates HW interface to send data.
- **tmo**: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.
- **fmt**: Address of format string.
- **ap**: List of arguments.

```
int esp_apptrace_vprintf(const char *fmt, va_list ap)
```

vprintf-like function to sent log messages to host.

Return Number of bytes written.

Parameters

- **fmt**: Address of format string.
- **ap**: List of arguments.

```
esp_err_t esp_apptrace_flush(esp_apptrace_dest_t dest, uint32_t tmo)
```

Flushes remaining data in trace buffer to host.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- **dest**: Indicates HW interface to flush data on.
- **tmo**: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

```
esp_err_t esp_apptrace_flush_nolock(esp_apptrace_dest_t dest, uint32_t min_sz, uint32_t tmo)
```

Flushes remaining data in trace buffer to host without locking internal data. This is special version of esp_apptrace_flush which should be called from panic handler.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- **dest**: Indicates HW interface to flush data on.
- **min_sz**: Threshold for flushing data. If current filling level is above this value, data will be flushed. TRAX destinations only.
- **tmo**: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

```
esp_err_t esp_apptrace_read(esp_apptrace_dest_t dest, void *data, uint32_t *size, uint32_t tmo)
```

Reads host data from trace buffer.

Return ESP_OK on success, otherwise see esp_err_t

Parameters

- dest: Indicates HW interface to read the data on.
- data: Address of buffer to put data from trace buffer.
- size: Pointer to store size of read data. Before call to this function pointed memory must hold requested size of data
- tmo: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

`uint8_t *esp_apptrace_down_buffer_get (esp_apptrace_dest_t dest, uint32_t *size, uint32_t tmo)`

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-NUL on success, otherwise NULL.

Parameters

- dest: Indicates HW interface to receive data.
- size: Address to store size of available data in down buffer. Must be initialized with requested value.
- tmo: Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

`esp_err_t esp_apptrace_down_buffer_put (esp_apptrace_dest_t dest, uint8_t *ptr, uint32_t tmo)`

Indicates that the data in down buffer are processed. This function is a counterpart of and must be preceded by esp_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 indefinitely.

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.

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

2.8 Configuration Options

2.8.1 Introduction

ESP-IDF uses [Kconfig](#) system to provide a compile-time configuration mechanism. Kconfig is based around options of several types: integer, string, boolean. Kconfig files specify dependencies between options, default values of the options, the way the options are grouped together, etc.

Applications developers can use `make menuconfig` build target to edit components' configuration. This configuration is saved inside `sdkconfig` file in the project root directory. Based on `sdkconfig`, application build targets will generate `sdkconfig.h` file in the build directory, and will make `sdkconfig` options available to component makefiles.

2.8.2 Using `sdkconfig.defaults`

When updating ESP-IDF version, it is not uncommon to find that new Kconfig options are introduced. When this happens, application build targets will offer an interactive prompt to select values for the new options. New values are then written into `sdkconfig` file. To suppress interactive prompts, applications can either define `BATCH_BUILD` environment variable, which will cause all prompts to be suppressed. This is the same effect as that of `V` or `VERBOSE` variables. Alternatively, `defconfig` build target can be used to update configuration for all new variables to the default values.

In some cases, such as when `sdkconfig` file is under revision control, the fact that `sdkconfig` file gets changed by the build system may be inconvenient. The build system offers a way to avoid this, in the form of `sdkconfig.defaults` file. This file is never touched by the build system, and must be created manually. It can contain all the options which matter for the given application. The format is the same as that of the `sdkconfig` file. Once `sdkconfig.defaults` is created, `sdkconfig` can be deleted and added to the ignore list of the revision control system (e.g. `.gitignore` file for git). Project build targets will automatically create `sdkconfig` file, populated with the settings from `sdkconfig.defaults` file, and the rest of the settings will be set to their default values. Note that when `make defconfig` is used, settings in `sdkconfig` will be overridden by the ones in `sdkconfig.defaults`. For more information, see [Custom `sdkconfig defaults`](#).

2.8.3 Configuration Options Reference

Subsequent sections contain the list of available ESP-IDF options, automatically generated from Kconfig files. Note that depending on the options selected, some options listed here may not be visible by default in the interface of `menuconfig`.

By convention, all option names are upper case with underscores. When Kconfig generates `sdkconfig` and `sdkconfig.h` files, option names are prefixed with `CONFIG_`. So if an option `ENABLE_FOO` is defined in a Kconfig file and selected in `menuconfig`, then `sdkconfig` and `sdkconfig.h` files will have `CONFIG_ENABLE_FOO` defined. In this reference, option names are also prefixed with `CONFIG_`, same as in the source code.

LWIP

`L2_TO_L3_COPY`

Enable copy between Layer2 and Layer3 packets

Found in: Component config > LWIP

If this feature is enabled, all traffic from layer2(WIFI Driver) will be copied to a new buffer before sending it to layer3(LWIP stack), freeing the layer2 buffer. Please be notified that the total layer2 receiving buffer

is fixed and ESP32 currently supports 25 layer2 receiving buffer, when layer2 buffer runs out of memory, then the incoming packets will be dropped in hardware. The layer3 buffer is allocated from the heap, so the total layer3 receiving buffer depends on the available heap size, when heap runs out of memory, no copy will be sent to layer3 and packet will be dropped in layer2. Please make sure you fully understand the impact of this feature before enabling it.

LWIP_MAX_SOCKETS

Max number of open sockets

Found in: Component config > LWIP

Sockets take up a certain amount of memory, and allowing fewer sockets to be open at the same time conserves memory. Specify the maximum amount of sockets here. The valid value is from 1 to 16.

LWIP_THREAD_LOCAL_STORAGE_INDEX

Index for thread-local-storage pointer for lwip

Found in: Component config > LWIP

Specify the thread-local-storage-pointer index for lwip use.

LWIP_SO_REUSE

Enable SO_REUSEADDR option

Found in: Component config > LWIP

Enabling this option allows binding to a port which remains in TIME_WAIT.

LWIP_SO_RCVBUF

Enable SO_RCVBUF option

Found in: Component config > LWIP

Enabling this option allows checking for available data on a netconn.

LWIP_DHCP_MAX_NTP_SERVERS

Maximum number of NTP servers

Found in: Component config > LWIP

Set maximum number of NTP servers used by LwIP SNTP module. First argument of sntp_setserver/sntp_setservername functions is limited to this value.

LWIP_IP_FRAG

Enable fragment outgoing IP packets

Found in: Component config > LWIP

Enabling this option allows fragmenting outgoing IP packets if their size exceeds MTU.

LWIP_IP_REASSEMBLY

Enable reassembly incoming fragmented IP packets

Found in: Component config > LWIP

Enabling this option allows reassembling incoming fragmented IP packets.

TCP

TCP_MAXRTX

Maximum number of retransmissions of data segments

Found in: Component config > LWIP > TCP

Set maximum number of retransmissions of data segments.

TCP_SYNMAXRTX

Maximum number of retransmissions of SYN segments

Found in: Component config > LWIP > TCP

Set maximum number of retransmissions of SYN segments.

TCP_MSS

Maximum Segment Size (MSS)

Found in: Component config > LWIP > TCP

Set maximum segment size for TCP transmission.

Can be set lower to save RAM, the default value 1436 will give best throughput.

TCP_MSL

Maximum segment lifetime (MSL)

Found in: Component config > LWIP > TCP

Set maximum segment lifetime in milliseconds.

TCP_SND_BUF_DEFAULT

Default send buffer size

Found in: Component config > LWIP > TCP

Set default send buffer size for new TCP sockets.

Per-socket send buffer size can be changed at runtime with `lwip_setsockopt(s, TCP_SNDBUF, ...)`.

This value must be at least 2x the MSS size, and the default is 4x the default MSS size.

Setting a smaller default SNDBUF size can save some RAM, but will decrease performance.

TCP_WND_DEFAULT

Default receive window size

Found in: Component config > LWIP > TCP

Set default TCP receive window size for new TCP sockets.

Per-socket receive window size can be changed at runtime with `lwip_setsockopt(s, TCP_WINDOW, ...)`.

Setting a smaller default receive window size can save some RAM, but will significantly decrease performance.

TCP_RECVMBX_SIZE

Default TCP receive mail box size

Found in: Component config > LWIP > TCP

Set TCP receive mail box size. Generally bigger value means higher throughput but more memory. The recommended value is: `TCP_WND_DEFAULT/TCP_MSS + 2`, e.g. if `TCP_WND_DEFAULT=14360, TCP_MSS=1436`, then the recommended receive mail box size is $(14360/1436 + 2) = 12$.

TCP receive mail box is a per socket mail box, when the application receives packets from TCP socket, LWIP core firstly posts the packets to TCP receive mail box and the application then fetches the packets from mail box. It means LWIP can cache maximum `TCP_RECVMBX_SIZE` packets for each TCP socket, so the maximum possible cached TCP packets for all TCP sockets is `TCP_RECVMBX_SIZE` multiples the maximum TCP socket number. In other words, the bigger `TCP_RECVMBX_SIZE` means more memory. On the other hand, if the receive mail box is too small, the mail box may be full. If the mail box is full, the LWIP drops the packets. So generally we need to make sure the TCP receive mail box is big enough to avoid packet drop between LWIP core and application.

TCP_QUEUE_OOSEQ

Queue incoming out-of-order segments

Found in: Component config > LWIP > TCP

Queue incoming out-of-order segments for later use.

Disable this option to save some RAM during TCP sessions, at the expense of increased retransmissions if segments arrive out of order.

TCP_OVERSIZE

Pre-allocate transmit PBUF size

Found in: Component config > LWIP > TCP

Allows enabling “oversize” allocation of TCP transmission pbufs ahead of time, which can reduce the length of pbuf chains used for transmission.

This will not make a difference to sockets where Nagle’s algorithm is disabled.

Default value of MSS is fine for most applications, 25% MSS may save some RAM when only transmitting small amounts of data. Disabled will have worst performance and fragmentation characteristics, but uses least RAM overall.

Available options:

- TCP_OVERSIZE_MSS
- TCP_OVERSIZE_QUARTER_MSS
- TCP_OVERSIZE_DISABLE

UDP

UDP_RECVMBX_SIZE

Default UDP receive mail box size

Found in: Component config > LWIP > UDP

Set UDP receive mail box size. The recommended value is 6.

UDP receive mail box is a per socket mail box, when the application receives packets from UDP socket, LWIP core firstly posts the packets to UDP receive mail box and the application then fetches the packets from mail box. It means LWIP can cache maximum UDP_RECVMBX_SIZE packets for each UDP socket, so the maximum possible cached UDP packets for all UDP sockets is UDP_RECVMBX_SIZE multiples the maximum UDP socket number. In other words, the bigger UDP_RECVMBX_SIZE means more memory. On the other hand, if the receive mail box is too small, the mail box may be full. If the mail box is full, the LWIP drops the packets. So generally we need to make sure the UDP receive mail box is big enough to avoid packet drop between LWIP core and application.

LWIP_DHCP_DOES_ARP_CHECK

Enable an ARP check on the offered address

Found in: Component config > LWIP

Enabling this option allows check if the offered IP address is not already in use by another host on the network.

TCP/IP_TASK_STACK_SIZE

TCP/IP Task Stack Size

Found in: Component config > LWIP

Configure TCP/IP task stack size, used by LWIP to process multi-threaded TCP/IP operations. The default is 2560 bytes, setting this stack too small will result in stack overflow crashes.

PPP_SUPPORT

Enable PPP support (new/experimental)

Found in: Component config > LWIP

Enable PPP stack. Now only PPP over serial is possible.

PPP over serial support is experimental and unsupported.

PPP_PAP_SUPPORT

Enable PAP support

Found in: Component config > LWIP

Enable Password Authentication Protocol (PAP) support

PPP_CHAP_SUPPORT

Enable CHAP support

Found in: Component config > LWIP

Enable Challenge Handshake Authentication Protocol (CHAP) support

PPP_MSCHAP_SUPPORT

Enable MSCHAP support

Found in: Component config > LWIP

Enable Microsoft version of the Challenge-Handshake Authentication Protocol (MSCHAP) support

PPP_MPPE_SUPPORT

Enable MPPE support

Found in: Component config > LWIP

Enable Microsoft Point-to-Point Encryption (MPPE) support

PPP_DEBUG_ON

Enable PPP debug log output

Found in: Component config > LWIP

Enable PPP debug log output

ICMP

LWIP_MULTICAST_PING

Respond to multicast pings

Found in: Component config > LWIP > ICMP

LWIP_BROADCAST_PING

Respond to broadcast pings

Found in: Component config > LWIP > ICMP

FreeRTOS

FREERTOS_UNICORE

Run FreeRTOS only on first core

Found in: Component config > FreeRTOS

This version of FreeRTOS normally takes control of all cores of the CPU. Select this if you only want to start it on the first core. This is needed when e.g. another process needs complete control over the second core.

FREERTOS_CORETIMER

Xtensa timer to use as the FreeRTOS tick source

Found in: Component config > FreeRTOS

FreeRTOS needs a timer with an associated interrupt to use as the main tick source to increase counters, run timers and do pre-emptive multitasking with. There are multiple timers available to do this, with different interrupt priorities. Check

Available options:

- FREERTOS_CORETIMER_0
- FREERTOS_CORETIMER_1

FREERTOS_HZ

Tick rate (Hz)

Found in: Component config > FreeRTOS

Select the tick rate at which FreeRTOS does pre-emptive context switching.

FREERTOS_ASSERT_ON_UNTESTED_FUNCTION

Halt when an SMP-untested function is called

Found in: Component config > FreeRTOS

Some functions in FreeRTOS have not been thoroughly tested yet when moving to the SMP implementation of FreeRTOS. When this option is enabled, these fuctions will throw an assert().

FREERTOS_CHECK_STACKOVERFLOW

Check for stack overflow

Found in: Component config > FreeRTOS

FreeRTOS can check for stack overflows in threads and trigger an user function called vApplicationStackOverflowHook when this happens.

Available options:

- FREERTOS_CHECK_STACKOVERFLOW_NONE
- FREERTOS_CHECK_STACKOVERFLOW_PTRVAL
- FREERTOS_CHECK_STACKOVERFLOW_CANARY

FREERTOS_WATCHPOINT_END_OF_STACK

Set a debug watchpoint as a stack overflow check

Found in: Component config > FreeRTOS

FreeRTOS can check if a stack has overflowed its bounds by checking either the value of the stack pointer or by checking the integrity of canary bytes. (See FREERTOS_CHECK_STACKOVERFLOW for more information.) These checks only happen on a context switch, and the situation that caused the stack overflow may already be long gone by then. This option will use the debug memory watchpoint 1 (the second one) to allow breaking into the debugger (or panic'ing) as soon as any of the last 32 bytes on the stack of a task are overwritten. The side effect is that using gdb, you effectively only have one watchpoint; the 2nd one is overwritten as soon as a task switch happens.

This check only triggers if the stack overflow writes within 4 bytes of the end of the stack, rather than overshooting further, so it is worth combining this approach with one of the other stack overflow check methods.

When this watchpoint is hit, gdb will stop with a SIGTRAP message. When no OCD is attached, esp-idf will panic on an unhandled debug exception.

FREERTOS_INTERRUPT_BACKTRACE

Enable backtrace from interrupt to task context

Found in: Component config > FreeRTOS

If this option is enabled, interrupt stack frame will be modified to point to the code of the interrupted task as its return address. This helps the debugger (or the panic handler) show a backtrace from the interrupt to the task which was interrupted. This also works for nested interrupts: higer level interrupt stack can be traced back to the lower level interrupt. This option adds 4 instructions to the interrupt dispatching code.

FREERTOS_THREAD_LOCAL_STORAGE_POINTERS

Number of thread local storage pointers

Found in: Component config > FreeRTOS

FreeRTOS has the ability to store per-thread pointers in the task control block. This controls the number of pointers available.

Value 0 turns off this functionality.

If using the LWIP TCP/IP stack (with WiFi or Ethernet), this value must be at least 1. See the LWIP_THREAD_LOCAL_STORAGE_INDEX config item in LWIP configuration to determine which thread-local-storage pointer is reserved for LWIP.

FREERTOS_ASSERT

FreeRTOS assertions

Found in: Component config > FreeRTOS

Failed FreeRTOS configASSERT() assertions can be configured to behave in different ways.

Available options:

- FREERTOS_ASSERT_FAIL_ABORT
- FREERTOS_ASSERT_FAIL_PRINT_CONTINUE
- FREERTOS_ASSERT_DISABLE

FREERTOS_BREAK_ON_SCHEDULER_START_JTAG

Stop program on scheduler start when JTAG/OCD is detected

Found in: Component config > FreeRTOS

If JTAG/OCD is connected, stop execution when the scheduler is started and the first task is executed.

ENABLE_MEMORY_DEBUG

Enable heap memory debug

Found in: Component config > FreeRTOS

Enable this option to show malloc heap block and memory crash detect

FREERTOS_IDLE_TASK_STACKSIZE

Idle Task stack size

Found in: Component config > FreeRTOS

The idle task has its own stack, sized in bytes. The default size is enough for most uses. Size can be reduced to 768 bytes if no (or simple) FreeRTOS idle hooks are used. The stack size may need to be increased above the default if the app installs idle hooks that use a lot of stack memory.

FREERTOS_ISR_STACKSIZE

ISR stack size

Found in: Component config > FreeRTOS

The interrupt handlers have their own stack. The size of the stack can be defined here. Each processor has its own stack, so the total size occupied will be twice this.

FREERTOS_LEGACY_HOOKS

Use FreeRTOS legacy hooks

Found in: Component config > FreeRTOS

FreeRTOS offers a number of hooks/callback functions that are called when a timer tick happens, the idle thread runs etc. esp-idf replaces these by runtime registerable hooks using the esp_register_freertos_xxx_hook system, but for legacy reasons the old hooks can also still be enabled. Please enable this only if you have code that for some reason can't be migrated to the esp_register_freertos_xxx_hook system.

FREERTOS_LEGACY_IDLE_HOOK

Enable legacy idle hook

Found in: Component config > FreeRTOS

If enabled, FreeRTOS will call a function called vApplicationIdleHook when the idle thread on a CPU is running. Please make sure your code defines such a function.

FREERTOS_LEGACY_TICK_HOOK

Enable legacy tick hook

Found in: Component config > FreeRTOS

If enabled, FreeRTOS will call a function called vApplicationTickHook when a FreeRTOS tick is executed. Please make sure your code defines such a function.

FREERTOS_MAX_TASK_NAME_LEN

Maximum task name length

Found in: Component config > FreeRTOS

Changes the maximum task name length. Each task allocated will include this many bytes for a task name. Using a shorter value saves a small amount of RAM, a longer value allows more complex names.

For most uses, the default of 16 is OK.

SUPPORT_STATIC_ALLOCATION

Enable FreeRTOS static allocation API

Found in: Component config > FreeRTOS

FreeRTOS gives the application writer the ability to instead provide the memory themselves, allowing the following objects to optionally be created without any memory being allocated dynamically:

- Tasks
- Software Timers
- Queues
- Event Groups
- Binary Semaphores
- Counting Semaphores
- Recursive Semaphores
- Mutexes

Whether it is preferable to use static or dynamic memory allocation is dependent on the application, and the preference of the application writer. Both methods have pros and cons, and both methods can be used within the same RTOS application.

Creating RTOS objects using statically allocated RAM has the benefit of providing the application writer with more control: RTOS objects can be placed at specific memory locations. The maximum RAM footprint can be determined at link time, rather than run time. The application writer does not need to concern themselves with graceful handling of memory allocation failures. It allows the RTOS to be used in applications that simply don't allow any dynamic memory allocation (although FreeRTOS includes allocation schemes that can overcome most objections).

ENABLE_STATIC_TASK_CLEAN_UP_HOOK

Enable static task clean up hook

Found in: Component config > FreeRTOS

Enable this option to make FreeRTOS call the static task clean up hook when a task is deleted.

Bear in mind that if this option is enabled you will need to implement the following function:

```
void vPortCleanUpTCB ( void *pxTCB ) {  
    // place clean up code here  
}
```

TIMER_TASK_PRIORITY

FreeRTOS timer task priority

Found in: Component config > FreeRTOS

The timer service task (primarily) makes use of existing FreeRTOS features, allowing timer functionality to be added to an application with minimal impact on the size of the application's executable binary.

Use this constant to define the priority that the timer task will run at.

TIMER_TASK_STACK_DEPTH

FreeRTOS timer task stack size

Found in: Component config > FreeRTOS

The timer service task (primarily) makes use of existing FreeRTOS features, allowing timer functionality to be added to an application with minimal impact on the size of the application's executable binary.

Use this constant to define the size (in bytes) of the stack allocated for the timer task.

TIMER_QUEUE_LENGTH

FreeRTOS timer queue length

Found in: Component config > FreeRTOS

FreeRTOS provides a set of timer related API functions. Many of these functions use a standard FreeRTOS queue to send commands to the timer service task. The queue used for this purpose is called the 'timer command queue'. The 'timer command queue' is private to the FreeRTOS timer implementation, and cannot be accessed directly.

For most uses the default value of 10 is OK.

FREERTOS_DEBUG_INTERNALS

Debug FreeRTOS internals

Found in: Component config > FreeRTOS

Enable this option to show the menu with internal FreeRTOS debugging features. This option does not change any code by itself, it just shows/hides some options.

FREERTOS_PORTMUX_DEBUG

Debug portMUX portENTER_CRITICAL/portEXIT_CRITICAL

Found in: Component config > FreeRTOS

If enabled, debug information (including integrity checks) will be printed to UART for the port-specific MUX implementation.

FREERTOS_PORTMUX_DEBUG_RECURSIVE

Debug portMUX Recursion

Found in: Component config > FreeRTOS

If enabled, additional debug information will be printed for recursive portMUX usage.

Wear Levelling

WL_SECTOR_SIZE

Wear Levelling library sector size

Found in: Component config > Wear Levelling

Sector size used by wear levelling library. You can set default sector size or size that will fit to the flash device sector size.

With sector size set to 4096 bytes, wear levelling library is more efficient. However if FAT filesystem is used on top of wear levelling library, it will need more temporary storage: 4096 bytes for each mounted filesystem and 4096 bytes for each opened file.

With sector size set to 512 bytes, wear levelling library will perform more operations with flash memory, but less RAM will be used by FAT filesystem library (512 bytes for the filesystem and 512 bytes for each file opened).

Available options:

- WL_SECTOR_SIZE_512
- WL_SECTOR_SIZE_4096

WL_SECTOR_MODE

Sector store mode

Found in: Component config > Wear Levelling

Specify the mode to store data into flash:

- In Performance mode a data will be stored to the RAM and then stored back to the flash. Compared to the Safety mode, this operation is faster, but if power will be lost when erase sector operation is in progress, then the data from complete flash device sector will be lost.
- In Safety mode data from complete flash device sector will be read from flash, modified, and then stored back to flash. Compared to the Performance mode, this operation is slower, but if power is lost during erase sector operation, then the data from full flash device sector will not be lost.

Available options:

- WL_SECTOR_MODE_PERF
- WL_SECTOR_MODE_SAFE

Heap memory debugging

HEAP_CORRUPTION_DETECTION

Heap corruption detection

Found in: Component config > Heap memory debugging

Enable heap poisoning features to detect heap corruption caused by out-of-bounds access to heap memory.

See the “Heap Memory Debugging” page of the IDF documentation for a description of each level of heap corruption detection.

Available options:

- HEAP_POISONING_DISABLED
- HEAP_POISONING_LIGHT
- HEAP_POISONING_COMPREHENSIVE

HEAP_TRACING

Enable heap tracing

Found in: Component config > Heap memory debugging

Enables the heap tracing API defined in esp_heap_trace.h.

This function causes a moderate increase in IRAM code 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.

HEAP_TRACING_STACK_DEPTH

Heap tracing stack depth

Found in: Component config > Heap memory debugging

Number of stack frames to save when tracing heap operation callers.

More stack frames uses more memory in the heap trace buffer (and slows down allocation), but can provide useful information.

Partition Table

PARTITION_TABLE_TYPE

Partition Table

Found in: Partition Table

The partition table to flash to the ESP32. The partition table determines where apps, data and other resources are expected to be found.

The predefined partition table CSV descriptions can be found in the components/partition_table directory. Otherwise it's possible to create a new custom partition CSV for your application.

Available options:

- PARTITION_TABLE_SINGLE_APP
- PARTITION_TABLE_TWO_OTA
- PARTITION_TABLE_CUSTOM

PARTITION_TABLE_CUSTOM_FILENAME

Custom partition CSV file

Found in: Partition Table

Name of the custom partition CSV filename. This path is evaluated relative to the project root directory.

PARTITION_TABLE_CUSTOM_APP_BIN_OFFSET

Factory app partition offset

Found in: Partition Table

If using a custom partition table, specify the offset in the flash where ‘make flash’ should write the built app.

PARTITION_TABLE_CUSTOM_PHY_DATA_OFFSET

PHY data partition offset

Found in: Partition Table

If using a custom partition table, specify the offset in the flash where ‘make flash’ should write the initial PHY data file.

ESP32-specific

ESP32_DEFAULT_CPU_FREQ_MHZ

CPU frequency

Found in: Component config > ESP32-specific

CPU frequency to be set on application startup.

Available options:

- ESP32_DEFAULT_CPU_FREQ_80
- ESP32_DEFAULT_CPU_FREQ_160
- ESP32_DEFAULT_CPU_FREQ_240

MEMMAP_SMP

Reserve memory for two cores

Found in: Component config > ESP32-specific

The ESP32 contains two cores. If you plan to only use one, you can disable this item to save some memory. (ToDo: Make this automatically depend on unicore support)

SPIRAM_SUPPORT

Support for external, SPI-connected RAM

Found in: Component config > ESP32-specific

This enables support for an external SPI RAM chip, connected in parallel with the main SPI flash chip.

SPI RAM config

SPIRAM_BOOT_INIT

Initialize SPI RAM when booting the ESP32

Found in: Component config > ESP32-specific > SPI RAM config

If this is enabled, the SPI RAM will be enabled during initial boot. Unless you have specific requirements, you'll want to leave this enabled so memory allocated during boot-up can also be placed in SPI RAM.

SPIRAM_USE

SPI RAM access method

Found in: Component config > ESP32-specific > SPI RAM config

The SPI RAM can be accessed in multiple methods: by just having it available as an unmanaged memory region in the ESP32 memory map, by integrating it in the ESP32s heap as ‘special’ memory needing `heap_caps_malloc` to allocate, or by fully integrating it making `malloc()` also able to return SPI RAM pointers.

Available options:

- SPIRAM_USE_MEMMAP
- SPIRAM_USE_CAPS_ALLOC
- SPIRAM_USE_MALLOC

SPIRAM_TYPE

Type of SPI RAM chip in use

Found in: Component config > ESP32-specific > SPI RAM config

Available options:

- SPIRAM_TYPE_ESPPSRAM32

SPIRAM_SPEED

Set RAM clock speed

Found in: Component config > ESP32-specific > SPI RAM config

Select the speed for the SPI RAM chip. If SPI RAM is enabled, we only support three combinations of SPI speed mode we supported now:

1. Flash SPI running at 40Mhz and RAM SPI running at 40Mhz
2. Flash SPI running at 80Mhz and RAM SPI running at 40Mhz
3. Flash SPI running at 80Mhz and RAM SPI running at 80Mhz

Note: If the third mode(80Mhz+80Mhz) is enabled, the VSPI port will be occupied by the system.

Application code should never touch VSPI hardware in this case. The option to select 80MHz will only be visible if the flash SPI speed is also 80MHz. (`ESPTOOLPY_FLASHFREQ_80M` is true)

Available options:

- SPIRAM_SPEED_40M
- SPIRAM_SPEED_80M

SPIRAM_MEMTEST

Run memory test on SPI RAM initialization

Found in: Component config > ESP32-specific > SPI RAM config

Runs a rudimentary memory test on initialization. Aborts when memory test fails. Disable this for slightly faster startop.

SPIRAM_CACHE_WORKAROUND

Enable workaround for bug in SPI RAM cache for Rev1 ESP32s

Found in: Component config > ESP32-specific > SPI RAM config

Revision 1 of the ESP32 has a bug that can cause a write to PSRAM not to take place in some situations when the cache line needs to be fetched from external RAM and an interrupt occurs. This enables a fix in the compiler that makes sure the specific code that is vulnerable to this will not be emitted.

This will also not use any bits of newlib that are located in ROM, opting for a version that is compiled with the workaround and located in flash instead.

ESP32_TRAX

Use TRAX tracing feature

Found in: Component config > ESP32-specific

The ESP32 contains a feature which allows you to trace the execution path the processor has taken through the program. This is stored in a chunk of 32K (16K for single-processor) of memory that can't be used for general purposes anymore. Disable this if you do not know what this is.

ESP32_TRAX_TWOBANKS

Reserve memory for tracing both pro as well as app cpu execution

Found in: Component config > ESP32-specific

The ESP32 contains a feature which allows you to trace the execution path the processor has taken through the program. This is stored in a chunk of 32K (16K for single-processor) of memory that can't be used for general purposes anymore. Disable this if you do not know what this is.

ESP32_COREDUMP_TO_FLASH_OR_UART

Core dump destination

Found in: Component config > ESP32-specific

Select place to store core dump: flash, uart or none (to disable core dumps generation).

If core dump is configured to be stored in flash and custom partition table is used add corresponding entry to your CSV. For examples, please see predefined partition table CSV descriptions in the components/partition_table directory.

Available options:

- ESP32_ENABLE_COREDUMP_TO_FLASH
- ESP32_ENABLE_COREDUMP_TO_UART
- ESP32_ENABLE_COREDUMP_TO_NONE

ESP32_CORE_DUMP_UART_DELAY

Core dump print to UART delay

Found in: Component config > ESP32-specific

Config delay (in ms) before printing core dump to UART. Delay can be interrupted by pressing Enter key.

ESP32_CORE_DUMP_LOG_LEVEL

Core dump module logging level

Found in: Component config > ESP32-specific

Config core dump module logging level (0-5).

NUMBER_OF_UNIVERSAL_MAC_ADDRESS

Number of universally administered (by IEEE) MAC address

Found in: Component config > ESP32-specific

Configure the number of universally administered (by IEEE) MAC addresses. During initialisation, MAC addresses for each network interface are generated or derived from a single base MAC address. If the number of universal MAC addresses is four, all four interfaces (WiFi station, WiFi softap, Bluetooth and Ethernet) receive a universally administered MAC address. These are generated sequentially by adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address. If the number of universal MAC addresses is two, only two interfaces (WiFi station and Bluetooth) receive a universally administered MAC address. These are generated sequentially by adding 0 and 1 (respectively) to the base MAC address. The remaining two interfaces (WiFi softap and Ethernet) receive local MAC addresses. These are derived from the universal WiFi station and Bluetooth MAC addresses, respectively. When using the default (Espressif-assigned) base MAC address, either setting can be used. When using a custom universal MAC address range, the correct setting will depend on the allocation of MAC addresses in this range (either 2 or 4 per device.)

Available options:

- TWO_UNIVERSAL_MAC_ADDRESS
- FOUR_UNIVERSAL_MAC_ADDRESS

SYSTEM_EVENT_QUEUE_SIZE

System event queue size

Found in: Component config > ESP32-specific

Config system event queue size in different application.

SYSTEM_EVENT_TASK_STACK_SIZE

Event loop task stack size

Found in: Component config > ESP32-specific

Config system event task stack size in different application.

MAIN_TASK_STACK_SIZE

Main task stack size

Found in: Component config > ESP32-specific

Configure the “main task” stack size. This is the stack of the task which calls `app_main()`. If `app_main()` returns then this task is deleted and its stack memory is freed.

IPC_TASK_STACK_SIZE

Inter-Processor Call (IPC) task stack size

Found in: Component config > ESP32-specific

Configure the IPC tasks stack size. One IPC task runs on each core (in dual core mode), and allows for cross-core function calls.

See IPC documentation for more details.

The default stack size should be enough for most common use cases. It can be shrunk if you are sure that you do not use any custom IPC functionality.

TIMER_TASK_STACK_SIZE

High-resolution timer task stack size

Found in: Component config > ESP32-specific

Configure the stack size of `esp_timer/ets_timer` task. This task is used to dispatch callbacks of timers created using `ets_timer` and `esp_timer` APIs. If you are seeing stack overflow errors in timer task, increase this value.

Note that this is not the same as FreeRTOS timer task. To configure FreeRTOS timer task size, see “FreeRTOS timer task stack size” option in “FreeRTOS” menu.

NEWLIB_STDOUT_LINE_ENDING

Line ending for UART output

Found in: Component config > ESP32-specific

This option allows configuring the desired line endings sent to UART when a newline ('n', LF) appears on stdout. Three options are possible:

CRLF: whenever LF is encountered, prepend it with CR

LF: no modification is applied, stdout is sent as is

CR: each occurrence of LF is replaced with CR

This option doesn't affect behavior of the UART driver (drivers/uart.h).

Available options:

- NEWLIB_STDOUT_LINE_ENDING_CRLF
- NEWLIB_STDOUT_LINE_ENDING_LF
- NEWLIB_STDOUT_LINE_ENDING_CR

NEWLIB_STDIN_LINE_ENDING

Line ending for UART input

Found in: Component config > ESP32-specific

This option allows configuring which input sequence on UART produces a newline ('n', LF) on stdin. Three options are possible:

CRLF: CRLF is converted to LF

LF: no modification is applied, input is sent to stdin as is

CR: each occurrence of CR is replaced with LF

This option doesn't affect behavior of the UART driver (drivers/uart.h).

Available options:

- NEWLIB_STDIN_LINE_ENDING_CRLF
- NEWLIB_STDIN_LINE_ENDING_LF
- NEWLIB_STDIN_LINE_ENDING_CR

NEWLIB_NANO_FORMAT

Enable 'nano' formatting options for printf/scanf family

Found in: Component config > ESP32-specific

ESP32 ROM contains parts of newlib C library, including printf/scanf family of functions. These functions have been compiled with so-called "nano" formatting option. This option doesn't support 64-bit integer formats and C99 features, such as positional arguments.

For more details about "nano" formatting option, please see newlib readme file, search for '--enable-newlib-nano-formatted-io': <https://sourceware.org/newlib/README>

If this option is enabled, build system will use functions available in ROM, reducing the application binary size. Functions available in ROM run faster than functions which run from flash. Functions available in ROM can also run when flash instruction cache is disabled.

If you need 64-bit integer formatting support or C99 features, keep this option disabled.

CONSOLE_UART

UART for console output

Found in: Component config > ESP32-specific

Select whether to use UART for console output (through stdout and stderr).

- Default is to use UART0 on pins GPIO1(TX) and GPIO3(RX).
- If “Custom” is selected, UART0 or UART1 can be chosen, and any pins can be selected.
- If “None” is selected, there will be no console output on any UART, except for initial output from ROM bootloader. This output can be further suppressed by bootstrapping GPIO13 pin to low logic level.

Available options:

- CONSOLE_UART_DEFAULT
- CONSOLE_UART_CUSTOM
- CONSOLE_UART_NONE

CONSOLE_UART_NUM

UART peripheral to use for console output (0-1)

Found in: Component config > ESP32-specific

Due of a ROM bug, UART2 is not supported for console output via ets_printf.

Available options:

- CONSOLE_UART_CUSTOM_NUM_0
- CONSOLE_UART_CUSTOM_NUM_1

CONSOLE_UART_TX_GPIO

UART TX on GPIO#

Found in: Component config > ESP32-specific

CONSOLE_UART_RX_GPIO

UART RX on GPIO#

Found in: Component config > ESP32-specific

CONSOLE_UART_BAUDRATE

UART console baud rate

Found in: Component config > ESP32-specific

ULP_COPROC_ENABLED

Enable Ultra Low Power (ULP) Coprocessor

Found in: Component config > ESP32-specific

Set to ‘y’ if you plan to load a firmware for the coprocessor.

If this option is enabled, further coprocessor configuration will appear in the Components menu.

ULP_COPROC_RESERVE_MEM

RTC slow memory reserved for coprocessor

Found in: Component config > ESP32-specific

Bytes of memory to reserve for ULP coprocessor firmware & data.

Data is reserved at the beginning of RTC slow memory.

ESP32_PANIC

Panic handler behaviour

Found in: Component config > ESP32-specific

If FreeRTOS detects unexpected behaviour or an unhandled exception, the panic handler is invoked.
Configure the panic handlers action here.

Available options:

- ESP32_PANIC_PRINT_HALT
- ESP32_PANIC_PRINT_REBOOT
- ESP32_PANIC_SILENT_REBOOT
- ESP32_PANIC_GDBSTUB

ESP32_DEBUG_OCDWARE

Make exception and panic handlers JTAG/OCD aware

Found in: Component config > ESP32-specific

The FreeRTOS panic and unhandled exception handlers can detect a JTAG OCD debugger and instead of panicking, have the debugger stop on the offending instruction.

INT_WDT

Interrupt watchdog

Found in: Component config > ESP32-specific

This watchdog timer can detect if the FreeRTOS tick interrupt has not been called for a certain time, either because a task turned off interrupts and did not turn them on for a long time, or because an interrupt handler did not return. It will try to invoke the panic handler first and failing that reset the SoC.

INT_WDT_TIMEOUT_MS

Interrupt watchdog timeout (ms)

Found in: Component config > ESP32-specific

The timeout of the watchdog, in milliseconds. Make this higher than the FreeRTOS tick rate.

INT_WDT_CHECK_CPU1

Also watch CPU1 tick interrupt

Found in: Component config > ESP32-specific

Also detect if interrupts on CPU 1 are disabled for too long.

TASK_WDT

Task watchdog

Found in: Component config > ESP32-specific

This watchdog timer can be used to make sure individual tasks are still running.

TASK_WDT_PANIC

Invoke panic handler when Task Watchdog is triggered

Found in: Component config > ESP32-specific

Normally, the Task Watchdog will only print out a warning if it detects it has not been fed. If this is enabled, it will invoke the panic handler instead, which can then halt or reboot the chip.

TASK_WDT_TIMEOUT_S

Task watchdog timeout (seconds)

Found in: Component config > ESP32-specific

Timeout for the task WDT, in seconds.

TASK_WDT_CHECK_IDLE_TASK

Task watchdog watches CPU0 idle task

Found in: Component config > ESP32-specific

With this turned on, the task WDT can detect if the idle task is not called within the task watchdog timeout period. The idle task not being called usually is a symptom of another task hoarding the CPU. It is also a bad thing because FreeRTOS household tasks depend on the idle task getting some runtime every now and then. Take Care: With this disabled, this watchdog will trigger if no tasks register themselves within the timeout value.

TASK_WDT_CHECK_IDLE_TASK_CPU1

Task watchdog also watches CPU1 idle task

Found in: Component config > ESP32-specific

Also check the idle task that runs on CPU1.

BROWNOUT_DET

Hardware brownout detect & reset

Found in: Component config > ESP32-specific

The ESP32 has a built-in brownout detector which can detect if the voltage is lower than a specific value. If this happens, it will reset the chip in order to prevent unintended behaviour.

BROWNOUT_DET_LVL_SEL

Brownout voltage level

Found in: Component config > ESP32-specific

The brownout detector will reset the chip when the supply voltage is below this level.

Available options:

- BROWNOUT_DET_LVL_SEL_0
- BROWNOUT_DET_LVL_SEL_1
- BROWNOUT_DET_LVL_SEL_2
- BROWNOUT_DET_LVL_SEL_3
- BROWNOUT_DET_LVL_SEL_4
- BROWNOUT_DET_LVL_SEL_5
- BROWNOUT_DET_LVL_SEL_6
- BROWNOUT_DET_LVL_SEL_7

ESP32_TIME_SYSCALL

Timers used for gettimeofday function

Found in: Component config > ESP32-specific

This setting defines which hardware timers are used to implement ‘gettimeofday’ and ‘time’ functions in C library.

- If only FRC1 timer is used, gettimeofday will provide time at microsecond resolution. Time will not be preserved when going into deep sleep mode.
- If both FRC1 and RTC timers are used, timekeeping will continue in deep sleep. Time will be reported at 1 microsecond resolution.
- If only RTC timer is used, timekeeping will continue in deep sleep, but time will be measured at 6.(6) microsecond resolution. Also the gettimeofday function itself may take longer to run.
- If no timers are used, gettimeofday and time functions return -1 and set errno to ENOSYS.
- When RTC is used for timekeeping, two RTC_STORE registers are used to keep time in deep sleep mode.

Available options:

- `ESP32_TIME_SYSCALL_USE_RTC`
- `ESP32_TIME_SYSCALL_USE_RTC_FRC1`
- `ESP32_TIME_SYSCALL_USE_FRC1`
- `ESP32_TIME_SYSCALL_USE_NONE`

ESP32_RTC_CLOCK_SOURCE

RTC clock source

Found in: Component config > ESP32-specific

Choose which clock is used as RTC clock source.

Available options:

- `ESP32_RTC_CLOCK_SOURCE_INTERNAL_RC`
- `ESP32_RTC_CLOCK_SOURCE_EXTERNAL_CRYSTAL`

ESP32_RTC_CLK_CAL_CYCLES

Number of cycles for RTC_SLOW_CLK calibration

Found in: Component config > ESP32-specific

When the startup code initializes RTC_SLOW_CLK, it can perform calibration by comparing the RTC_SLOW_CLK frequency with main XTAL frequency. This option sets the number of RTC_SLOW_CLK cycles measured by the calibration routine. Higher numbers increase calibration precision, which may be important for applications which spend a lot of time in deep sleep. Lower numbers reduce startup time.

When this option is set to 0, clock calibration will not be performed at startup, and approximate clock frequencies will be assumed:

- 150000 Hz if internal RC oscillator is used as clock source
- 32768 Hz if the 32k crystal oscillator is used

ESP32_DEEP_SLEEP_WAKEUP_DELAY

Extra delay in deep sleep wake stub (in us)

Found in: Component config > ESP32-specific

When ESP32 exits deep sleep, the CPU and the flash chip are powered on at the same time. CPU will run deep sleep stub first, and then proceed to load code from flash. Some flash chips need sufficient time to pass between power on and first read operation. By default, without any extra delay, this time is approximately 900us, although some flash chip types need more than that.

By default extra delay is set to 2000us. When optimizing startup time for applications which require it, this value may be reduced.

If you are seeing “flash read err, 1000” message printed to the console after deep sleep reset, try increasing this value.

ESP32_XTAL_FREQ_SEL

Main XTAL frequency

Found in: Component config > ESP32-specific

ESP32 currently supports the following XTAL frequencies:

- 26 MHz
- 40 MHz

Startup code can automatically estimate XTAL frequency. This feature uses the internal 8MHz oscillator as a reference. Because the internal oscillator frequency is temperature dependent, it is not recommended to use automatic XTAL frequency detection in applications which need to work at high ambient temperatures and use high-temperature qualified chips and modules.

Available options:

- `ESP32_XTAL_FREQ_40`
- `ESP32_XTAL_FREQ_26`
- `ESP32_XTAL_FREQ_AUTO`

DISABLE_BASIC_ROM_CONSOLE

Permanently disable BASIC ROM Console

Found in: Component config > ESP32-specific

If set, the first time the app boots it will disable the BASIC ROM Console permanently (by burning an efuse).

Otherwise, the BASIC ROM Console starts on reset if no valid bootloader is read from the flash.

(Enabling secure boot also disables the BASIC ROM Console by default.)

NO_BLOBS

No Binary Blobs

Found in: Component config > ESP32-specific

If enabled, this disables the linking of binary libraries in the application build. Note that after enabling this Wi-Fi/Bluetooth will not work.

ESP_TIMER_PROFILING

Enable esp_timer profiling features

Found in: Component config > ESP32-specific

If enabled, esp_timer_dump will dump information such as number of times the timer was started, number of times the timer has triggered, and the total time it took for the callback to run. This option has some effect on timer performance and the amount of memory used for timer storage, and should only be used for debugging/testing purposes.

Wi-Fi

SW_COEXIST_ENABLE

Software controls WiFi/Bluetooth coexistence

Found in: Component config > Wi-Fi

If enabled, WiFi & Bluetooth coexistence is controlled by software rather than hardware. Recommended for heavy traffic scenarios. Both coexistence configuration options are automatically managed, no user intervention is required.

ESP32_WIFI_STATIC_RX_BUFFER_NUM

Max number of WiFi static RX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi static rx buffers. Each buffer takes approximately 1.6KB of RAM. The static rx buffers are allocated when esp_wifi_init is called, they are not freed until esp_wifi_deinit is called. WiFi hardware use these buffers to receive packets, generally larger number for higher throughput but more memory, smaller number for lower throughput but less memory.

ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM

Max number of WiFi dynamic RX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi dynamic rx buffers, 0 means no limitation for dynamic rx buffer allocation. The size of dynamic rx buffers is not fixed. For each received packet in static rx buffers, WiFi driver makes a copy to dynamic rx buffers and then deliver it to high layer stack. The dynamic rx buffer is freed when the application, such as socket, successfully received the packet. For some applications, the WiFi driver receiving speed is faster than application consuming speed, we may run out of memory if no limitation for

the dynamic rx buffer number. Generally the number of dynamic rx buffer should be no less than static rx buffer number if it is not 0.

ESP32_WIFI_TX_BUFFER

Type of WiFi TX buffers

Found in: Component config > Wi-Fi

Select type of WiFi tx buffers and show the submenu with the number of WiFi tx buffers choice. If “STATIC” is selected, WiFi tx buffers are allocated when WiFi is initialized and released when WiFi is de-initialized. If “DYNAMIC” is selected, WiFi tx buffer is allocated when tx data is delivered from LWIP to WiFi and released when tx data is sent out by WiFi. The size of each static tx buffers is fixed to about 1.6KB and the size of dynamic tx buffers is depend on the length of the data delivered from LWIP. If PSRAM is enabled, “STATIC” should be selected to guarantee enough WiFi tx buffers. If PSRAM is disabled, “DYNAMIC” should be selected to improve the utilization of RAM.

Available options:

- ESP32_WIFI_STATIC_TX_BUFFER
- ESP32_WIFI_DYNAMIC_TX_BUFFER

ESP32_WIFI_STATIC_TX_BUFFER_NUM

Max number of WiFi static TX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi static tx buffers. Each buffer takes approximately 1.6KB of RAM. The static rx buffers are allocated when esp_wifi_init is called, they are not released until esp_wifi_deinit is called. For each tx packet from high layer stack, WiFi driver make a copy of it. For some applications, especially the UDP application, the high layer deliver speed is faster than the WiFi tx speed, we may run out of static tx buffers.

ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM

Max number of WiFi dynamic TX buffers

Found in: Component config > Wi-Fi

Set the number of WiFi dynamic tx buffers, 0 means no limitation for dynamic tx buffer allocation. The size of dynamic tx buffers is not fixed. For each tx packet from high layer stack, WiFi driver make a copy of it. For some applications, especially the UDP application, the high layer deliver speed is faster than the WiFi tx speed, we may run out of memory if no limitation for the dynamic tx buffer number.

ESP32_WIFI_AMPDU_ENABLED

WiFi AMPDU

Found in: Component config > Wi-Fi

Select this option to enable AMPDU feature

ESP32_WIFI_TX_BA_WIN

WiFi AMPDU TX BA window size

Found in: Component config > Wi-Fi

Set the size of WiFi Block Ack TX window. Generally a bigger value means higher throughput but more memory. Most of time we should NOT change the default value unless special reason, e.g. test the maximum UDP TX throughput with iperf etc. For iperf test in shieldbox, the recommended value is 9~12.

ESP32_WIFI_RX_BA_WIN

WiFi AMPDU RX BA window size

Found in: Component config > Wi-Fi

Set the size of WiFi Block Ack RX window. Generally a bigger value means higher throughput 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.

ESP32_WIFI_NVS_ENABLED

WiFi NVS flash

Found in: Component config > Wi-Fi

Select this option to enable WiFi NVS flash

PHY

ESP32_PHY_CALIBRATION_AND_DATA_STORAGE

Do phy calibration and store calibration data in NVS

Found in: Component config > PHY

If this option is enabled, NVS will be initialized and calibration data will be loaded from there. PHY calibration will be skipped on deep sleep wakeup. If calibration data is not found, full calibration will be performed and stored in NVS. In all other cases, only partial calibration will be performed.

If unsure, choose ‘y’.

ESP32_PHY_INIT_DATA_IN_PARTITION

Use a partition to store PHY init data

Found in: Component config > PHY

If enabled, PHY init data will be loaded from a partition. When using a custom partition table, make sure that PHY data partition is included (type: ‘data’, subtype: ‘phy’). With default partition tables, this is done automatically. If PHY init data is stored in a partition, it has to be flashed there, otherwise runtime error will occur.

If this option is not enabled, PHY init data will be embedded into the application binary.

If unsure, choose ‘n’.

ESP32_PHY_MAX_WIFI_TX_POWER

Max WiFi TX power (dBm)

Found in: Component config > PHY

Set maximum transmit power for WiFi radio. Actual transmit power for high data rates may be lower than this setting.

Log output

LOG_DEFAULT_LEVEL

Default log verbosity

Found in: Component config > Log output

Specify how much output to see in logs by default. You can set lower verbosity level at runtime using esp_log_level_set function.

Note that this setting limits which log statements are compiled into the program. So setting this to, say, “Warning” would mean that changing log level to “Debug” at runtime will not be possible.

Available options:

- LOG_DEFAULT_LEVEL_NONE
- LOG_DEFAULT_LEVEL_ERROR
- LOG_DEFAULT_LEVEL_WARN
- LOG_DEFAULT_LEVEL_INFO
- LOG_DEFAULT_LEVEL_DEBUG
- LOG_DEFAULT_LEVEL_VERBOSE

LOG_COLORS

Use ANSI terminal colors in log output

Found in: Component config > Log output

Enable ANSI terminal color codes in bootloader output.

In order to view these, your terminal program must support ANSI color codes.

PThreads

ESP32_PTHREAD_TASK_PRIO_DEFAULT

Default task priority

Found in: Component config > PThreads

Priority used to create new tasks with default pthread parameters.

ESP32_PTHREAD_TASK_STACK_SIZE_DEFAULT

Default task stack size

Found in: Component config > PThreads

Stack size used to create new tasks with default pthread parameters.

Application Level Tracing

ESP32_APPTRACE_DESTINATION

Data Destination

Found in: Component config > Application Level Tracing

Select destination for application trace: trace memory or none (to disable).

Available options:

- ESP32_APPTRACE_DEST_TRAX
- ESP32_APPTRACE_DEST_NONE

ESP32_APPTRACE_ONPANIC_HOST_FLUSH_TMO

Timeout for flushing last trace data to host on panic

Found in: Component config > Application Level Tracing

Timeout for flushing last trace data to host in case of panic. In ms. Use -1 to disable timeout and wait forever.

ESP32_APPTRACE_POSTMORTEM_FLUSH_TRAX_THRESH

Threshold for flushing last trace data to host on panic

Found in: Component config > Application Level Tracing

Threshold for flushing last trace data to host on panic in post-mortem mode. This is minimal amount of data needed to perform flush. In bytes.

ESP32_APPTRACE_PENDING_DATA_SIZE_MAX

Size of the pending data buffer

Found in: Component config > Application Level Tracing

Size of the buffer for events in bytes. It is useful for buffering events from the time critical code (scheduler, ISRs etc). If this parameter is 0 then events will be discarded when main HW buffer is full.

FreeRTOS SystemView Tracing

SYSVIEW_ENABLE

SystemView Tracing Enable

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables support for SEGGER SystemView tracing functionality.

SYSVIEW_TS_SOURCE

ESP32 timer to use as SystemView timestamp source

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

SystemView needs one source for timestamps when tracing events from both cores. This option selects HW timer for it.

Available options:

- SYSVIEW_TS_SOURCE_TIMER_00
- SYSVIEW_TS_SOURCE_TIMER_01
- SYSVIEW_TS_SOURCE_TIMER_10
- SYSVIEW_TS_SOURCE_TIMER_11

SYSVIEW_EVT_OVERFLOW_ENABLE

Trace Buffer Overflow Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Trace Buffer Overflow” event.

SYSVIEW_EVT_ISR_ENTER_ENABLE

ISR Enter Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “ISR Enter” event.

SYSVIEW_EVT_ISR_EXIT_ENABLE

ISR Exit Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “ISR Exit” event.

SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE

ISR Exit to Scheduler Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “ISR to Scheduler” event.

SYSVIEW_EVT_TASK_START_EXEC_ENABLE

Task Start Execution Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Task Start Execution” event.

SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE

Task Stop Execution Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Task Stop Execution” event.

SYSVIEW_EVT_TASK_START_READY_ENABLE

Task Start Ready State Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Task Start Ready State” event.

SYSVIEW_EVT_TASK_STOP_READY_ENABLE

Task Stop Ready State Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Task Stop Ready State” event.

SYSVIEW_EVT_TASK_CREATE_ENABLE

Task Create Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Task Create” event.

SYSVIEW_EVT_TASK_TERMINATE_ENABLE

Task Terminate Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Task Terminate” event.

SYSVIEW_EVT_IDLE_ENABLE

System Idle Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “System Idle” event.

SYSVIEW_EVT_TIMER_ENTER_ENABLE

Timer Enter Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Timer Enter” event.

SYSVIEW_EVT_TIMER_EXIT_ENABLE

Timer Exit Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing

Enables “Timer Exit” event.

Ethernet

DMA_RX_BUF_NUM

Number of DMA RX buffers

Found in: Component config > Ethernet

Number of DMA receive buffers. Each buffer is 1600 bytes. Buffers are allocated statically. Larger number of buffers increases throughput. If enable flow ctrl, the num must be above 9 .

DMA_TX_BUF_NUM

Number of DMA RX buffers

Found in: Component config > Ethernet

Number of DMA transmit buffers. Each buffer is 1600 bytes. Buffers are allocated statically. Larger number of buffers increases throughput.

EMAC_L2_TO_L3_RX_BUF_MODE

Enable copy between Layer2 and Layer3

Found in: Component config > Ethernet

If this options is selected, a copy of each received buffer will be created when passing it from the Ethernet MAC (L2) to the IP stack (L3). Otherwise, IP stack will receive pointers to the DMA buffers used by Ethernet MAC.

When Ethernet MAC doesn't have any unused buffers left, it will drop incoming packets (flow control may help with this problem, to some extent).

The buffers for the IP stack are allocated from the heap, so the total number of receive buffers is limited by the available heap size, if this option is selected.

If unsure, choose n.

EMAC_TASK_PRIORITY

EMAC_TASK_PRIORITY

Found in: Component config > Ethernet

Ethernet MAC task priority.

Bootloader config

LOG_BOOTLOADER_LEVEL

Bootloader log verbosity

Found in: Bootloader config

Specify how much output to see in bootloader logs.

Available options:

- LOG_BOOTLOADER_LEVEL_NONE
- LOG_BOOTLOADER_LEVEL_ERROR
- LOG_BOOTLOADER_LEVEL_WARN
- LOG_BOOTLOADER_LEVEL_INFO
- LOG_BOOTLOADER_LEVEL_DEBUG
- LOG_BOOTLOADER_LEVEL_VERBOSE

BOOTLOADER_SPI_WP_PIN

SPI Flash WP Pin when customising pins via efuse (read help)

Found in: Bootloader config

This value is ignored unless flash mode is set to QIO or QOUT *and* the SPI flash pins have been overridden by setting the efuses SPI_PAD_CONFIG_xxx.

When this is the case, the Efuse config only defines 3 of the 4 Quad I/O data pins. The WP pin (aka ESP32 pin "SD_DATA_3" or SPI flash pin "IO2") is not specified in Efuse. That pin number is compiled into the bootloader instead.

The default value (GPIO 7) is correct for WP pin on ESP32-D2WD integrated flash.

Security features

SECURE_BOOT_ENABLED

Enable secure boot in bootloader (READ DOCS FIRST)

Found in: Security features

Build a bootloader which enables secure boot on first boot.

Once enabled, secure boot will not boot a modified bootloader. The bootloader will only load a partition table or boot an app if the data has a verified digital signature. There are implications for reflashing updated apps once secure boot is enabled.

When enabling secure boot, JTAG and ROM BASIC Interpreter are permanently disabled by default.

Refer to <https://esp-idf.readthedocs.io/en/latest/security/secure-boot.html> before enabling.

SECURE_BOOTLOADER_MODE

Secure bootloader mode

Found in: Security features

Available options:

- SECURE_BOOTLOADER_ONE_TIME_FLASH
- SECURE_BOOTLOADER_REFFLASHABLE

SECURE_BOOT_BUILD_SIGNED_BINARIES

Sign binaries during build

Found in: Security features

Once secure boot is enabled, bootloader will only boot if partition table and app image are signed.

If enabled, these binary files are signed as part of the build process. The file named in “Secure boot private signing key” will be used to sign the image.

If disabled, unsigned app/partition data will be built. They must be signed manually using espsecure.py (for example, on a remote signing server.)

SECURE_BOOT_SIGNING_KEY

Secure boot private signing key

Found in: Security features

Path to the key file used to sign partition tables and app images for secure boot. Once secure boot is enabled, bootloader will only boot if partition table and app image are signed.

Key file is an ECDSA private key (NIST256p curve) in PEM format.

Path is evaluated relative to the project directory.

You can generate a new signing key by running the following command: espsecure.py generate_signing_key secure_boot_signing_key.pem

See docs/security/secure-boot.rst for details.

SECURE_BOOT_VERIFICATION_KEY

Secure boot public signature verification key

Found in: Security features

Path to a public key file used to verify signed images. This key is compiled into the bootloader, and may also be used to verify signatures on OTA images after download.

Key file is in raw binary format, and can be extracted from a PEM formatted private key using the espsecure.py extract_public_key command.

Refer to <https://esp-idf.readthedocs.io/en/latest/security/secure-boot.html> before enabling.

SECURE_BOOT_INSECURE

Allow potentially insecure options

Found in: Security features

You can disable some of the default protections offered by secure boot, in order to enable testing or a custom combination of security features.

Only enable these options if you are very sure.

Refer to <https://esp-idf.readthedocs.io/en/latest/security/secure-boot.html> before enabling.

FLASH_ENCRYPTION_ENABLED

Enable flash encryption on boot (READ DOCS FIRST)

Found in: Security features

If this option is set, flash contents will be encrypted by the bootloader on first boot.

Note: After first boot, the system will be permanently encrypted. Re-flashing an encrypted system is complicated and not always possible.

Read <https://esp-idf.readthedocs.io/en/latest/security/flash-encryption.html> before enabling.

FLASH_ENCRYPTION_INSECURE

Allow potentially insecure options

Found in: Security features

You can disable some of the default protections offered by flash encryption, in order to enable testing or a custom combination of security features.

Only enable these options if you are very sure.

Refer to docs/security/secure-boot.rst and docs/security/flash-encryption.rst for details.

Potentially insecure options

SECURE_BOOT_ALLOW_ROM_BASIC

Leave ROM BASIC Interpreter available on reset

Found in: Security features > Potentially insecure options

By default, the BASIC ROM Console starts on reset if no valid bootloader is read from the flash.

When either flash encryption or secure boot are enabled, the default is to disable this BASIC fallback mode permanently via efuse.

If this option is set, this efuse is not burned and the BASIC ROM Console may remain accessible. Only set this option in testing environments.

SECURE_BOOT_ALLOW_JTAG

Allow JTAG Debugging

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable JTAG (across entire chip) on first boot when either secure boot or flash encryption is enabled.

Setting this option leaves JTAG on for debugging, which negates all protections of flash encryption and some of the protections of secure boot.

Only set this option in testing environments.

FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_ENCRYPT

Leave UART bootloader encryption enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader encryption access on first boot. If set, the UART bootloader will still be able to access hardware encryption.

It is recommended to only set this option in testing environments.

FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_DECRYPT

Leave UART bootloader decryption enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader decryption access on first boot. If set, the UART bootloader will still be able to access hardware decryption.

Only set this option in testing environments. Setting this option allows complete bypass of flash encryption.

FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_CACHE

Leave UART bootloader flash cache enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader flash cache access on first boot. If set, the UART bootloader will still be able to access the flash cache.

Only set this option in testing environments.

SECURE_BOOT_TEST_MODE

Secure boot test mode: don't permanently set any efuses

Found in: Security features > Potentially insecure options

If this option is set, all permanent secure boot changes (via Efuse) are disabled.

Log output will state changes which would be applied, but they will not be.

This option is for testing purposes only - it completely disables secure boot protection.

OpenSSL

OPENSSL_DEBUG

Enable OpenSSL debugging

Found in: Component config > OpenSSL

Enable OpenSSL debugging function.

If the option is enabled, “SSL_DEBUG” works.

OPENSSL_DEBUG_LEVEL

OpenSSL debugging level

Found in: Component config > OpenSSL

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.

OPENSSL_LOWLEVEL_DEBUG

Enable OpenSSL low-level module debugging

Found in: Component config > OpenSSL

If the option is enabled, low-level module debugging function of OpenSSL is enabled, e.g. mbedTLS internal debugging function.

OPENSSL_ASSERT

Select OpenSSL assert function

Found in: Component config > OpenSSL

OpenSSL function needs “assert” function to check if input parameters are valid.

If you want to use assert debugging function, “OPENSSL_DEBUG” should be enabled.

Available options:

- OPENSSL_ASSERT_DO_NOTHING
- OPENSSL_ASSERT_EXIT
- OPENSSL_ASSERT_DEBUG
- OPENSSL_ASSERT_DEBUG_EXIT
- OPENSSL_ASSERT_DEBUG_BLOCK

AWS_IOT_SDK

Amazon Web Services IoT Platform

Found in: Component config

Select this option to enable support for the AWS IoT platform, via the esp-idf component for the AWS IoT Device C SDK.

AWS_IOT_MQTT_HOST

AWS IoT Endpoint Hostname

Found in: Component config

Default endpoint host name to connect to AWS IoT MQTT/S gateway

This is the custom endpoint hostname and is specific to an AWS IoT account. You can find it by logging into your AWS IoT Console and clicking the Settings button. The endpoint hostname is shown under the “Custom Endpoint” heading on this page.

If you need per-device hostnames for different regions or accounts, you can override the default hostname in your app.

AWS_IOT_MQTT_PORT

AWS IoT MQTT Port

Found in: Component config

Default port number to connect to AWS IoT MQTT/S gateway

If you need per-device port numbers for different regions, you can override the default port number in your app.

AWS_IOT_MQTT_TX_BUF_LEN

MQTT TX Buffer Length

Found in: Component config

Maximum MQTT transmit buffer size. This is the maximum MQTT message length (including protocol overhead) which can be sent.

Sending longer messages will fail.

AWS_IOT_MQTT_RX_BUF_LEN

MQTT RX Buffer Length

Found in: Component config

Maximum MQTT receive buffer size. This is the maximum MQTT message length (including protocol overhead) which can be received.

Longer messages are dropped.

AWS_IOT_MQTT_NUM_SUBSCRIBE_HANDLERS

Maximum MQTT Topic Filters

Found in: Component config

Maximum number of concurrent MQTT topic filters.

AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL

Auto reconnect initial interval (ms)

Found in: Component config

Initial delay before making first reconnect attempt, if the AWS IoT connection fails. Client will perform exponential backoff, starting from this value.

AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL

Auto reconnect maximum interval (ms)

Found in: Component config

Maximum delay between reconnection attempts. If the exponentially increased delay interval reaches this value, the client will stop automatically attempting to reconnect.

BT_ENABLED

Bluetooth

Found in: Component config

Select this option to enable Bluetooth and show the submenu with Bluetooth configuration choices.

BLUEDROID_ENABLED

Bluedroid Bluetooth stack enabled

Found in: Component config

This enables the default Bluedroid Bluetooth stack

BTC_TASK_STACK_SIZE

Bluetooth event (callback to application) task stack size

Found in: Component config

This select btc task stack size

BLUEDROID_MEM_DEBUG

Bluedroid memory debug

Found in: Component config

Bluedroid memory debug

CLASSIC_BT_ENABLED

Classic Bluetooth

Found in: Component config

For now this option needs “SMP_ENABLE” to be set to yes

BT_DRAM_RELEASE

Release DRAM from Classic BT controller

Found in: Component config

This option should only be used when BLE only. Enabling this option will release about 30K DRAM from Classic BT. The released DRAM will be used as system heap memory.

GATTS_ENABLE

Include GATT server module(GATTS)

Found in: Component config

This option can be disabled when the app work only on gatt client mode

GATT_ENABLE

Include GATT client module(GATT)

Found in: Component config

This option can be close when the app work only on gatt server mode

BLE_SMP_ENABLE

Include BLE security module(SMP)

Found in: Component config

This option can be close when the app not used the ble security connect.

BT_STACK_NO_LOG

Close the bluedroid bt stack log print

Found in: Component config

This select can save the rodata code size

BT_ACL_CONNECTIONS

BT/BLE MAX ACL CONNECTIONS(1~7)

Found in: Component config

Maximum BT/BLE connection count

BTDM_CONTROLLER_RUN_APP_CPU

Run controller on APP CPU

Found in: Component config

Run controller on APP CPU.

BT_HCI_UART

HCI use UART as IO

Found in: Component config

Default HCI use VHCI, if this option choose, HCI will use UART(0/1/2) as IO. Besides, it can set uart number and uart baudrate.

BT_HCI_UART_NO

UART Number for HCI

Found in: Component config

Uart number for HCI.

BT_HCI_UART_BAUDRATE

UART Baudrate for HCI

Found in: Component config

UART Baudrate for HCI. Please use standard baudrate.

mbedTLS

MBEDTLS_SSL_MAX_CONTENT_LEN

TLS maximum message content length

Found in: Component config > mbedTLS

Maximum TLS message length (in bytes) supported by mbedTLS.

16384 is the default and this value is required to comply fully with TLS standards.

However you can set a lower value in order to save RAM. This is safe if the other end of the connection supports Maximum Fragment Length Negotiation Extension (max_fragment_length, see RFC6066) or you know for certain that it will never send a message longer than a certain number of bytes.

If the value is set too low, symptoms are a failed TLS handshake or a return value of MBEDTLS_ERR_SSL_INVALID_RECORD (-0x7200).

MBEDTLS_DEBUG

Enable mbedTLS debugging

Found in: Component config > mbedTLS

Enable mbedTLS debugging functions at compile time.

If this option is enabled, you can include “mbedtls/esp_debug.h” **and** call
mbedtls_esp_enable_debug_log() **at runtime in order to enable mbedTLS debug output via**
the ESP log mechanism.

MBEDTLS_HARDWARE_AES

Enable hardware AES acceleration

Found in: Component config > mbedTLS

Enable hardware accelerated AES encryption & decryption.

Note that if the ESP32 CPU is running at 240MHz, hardware AES does not offer any speed boost over software AES.

MBEDTLS_HARDWARE_MPI

Enable hardware MPI (bignum) acceleration

Found in: Component config > mbedTLS

Enable hardware accelerated multiple precision integer operations.

Hardware accelerated multiplication, modulo multiplication, and modular exponentiation for up to 4096 bit results.

These operations are used by RSA.

MBEDTLS_MPI_USE_INTERRUPT

Use interrupt for MPI operations

Found in: Component config > mbedTLS

Use an interrupt to coordinate MPI operations.

This allows other code to run on the CPU while an MPI operation is pending. Otherwise the CPU busily waits.

MBEDTLS_HARDWARE_SHA

Enable hardware SHA acceleration

Found in: Component config > mbedTLS

Enable hardware accelerated SHA1, SHA256, SHA384 & SHA512 in mbedTLS.

Due to a hardware limitation, hardware acceleration is only guaranteed if SHA digests are calculated one at a time. If more than one SHA digest is calculated at the same time, one will be calculated fully in hardware and the rest will be calculated (at least partially calculated) in software. This happens automatically.

SHA hardware acceleration is faster than software in some situations but slower in others. You should benchmark to find the best setting for you.

MBEDTLS_HAVE_TIME

Enable mbedtls time

Found in: Component config > mbedTLS

System has time.h and time(). The time does not need to be correct, only time differences are used,

MBEDTLS_HAVE_TIME_DATE

Enable mbedtls time data

Found in: Component config > mbedTLS

System has time.h and time(), gmtime() and the clock is correct. The time needs to be correct (not necessarily very accurate, but at least the date should be correct). This is used to verify the validity period of X.509 certificates.

It is suggested that you should get the real time by “SNTP”.

MBEDTLS_TLS_MODE

TLS Protocol Role

Found in: Component config > mbedTLS

mbedTLS can be compiled with protocol support for the TLS server, TLS client, or both server and client.

Reducing the number of TLS roles supported saves code size.

Available options:

- MBEDTLS_TLS_SERVER_AND_CLIENT
- MBEDTLS_TLS_SERVER_ONLY
- MBEDTLS_TLS_CLIENT_ONLY
- MBEDTLS_TLS_DISABLED

TLS Key Exchange Methods

MBEDTLS_PSK_MODES

Enable pre-shared-key ciphersuites

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to show configuration for different types of pre-shared-key TLS authentication methods.

Leaving this options disabled will save code size if they are not used.

MBEDTLS_KEY_EXCHANGE_PSK

Enable PSK based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support symmetric key PSK (pre-shared-key) TLS key exchange modes.

MBEDTLS_KEY_EXCHANGE_DHE_PSK

Enable DHE-PSK based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

MBEDTLS_KEY_EXCHANGE_ECDHE_PSK

Enable ECDHE-PSK based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support Elliptic-Curve-Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

MBEDTLS_KEY_EXCHANGE_RSA_PSK

Enable RSA-PSK based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support RSA PSK (pre-shared-key) TLS authentication modes.

MBEDTLS_KEY_EXCHANGE_RSA

Enable RSA-only based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-RSA-WITH-

MBEDTLS_KEY_EXCHANGE_DHE_RSA

Enable DHE-RSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-DHE-RSA-WITH-

MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE

Support Elliptic Curve based ciphersuites

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to show Elliptic Curve based ciphersuite mode options.

Disabling all Elliptic Curve ciphersuites saves code size and can give slightly faster TLS handshakes, provided the server supports RSA-only ciphersuite modes.

MBEDTLS_KEY_EXCHANGE_ECDHE_RSA

Enable ECDHE-RSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA

Enable ECDHE-ECDSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA

Enable ECDH-ECDSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

MBEDTLS_KEY_EXCHANGE_ECDH_RSA

Enable ECDH-RSA based ciphersuite modes

Found in: Component config > mbedTLS > TLS Key Exchange Methods

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

MBEDTLS_SSL_RENEGOTIATION

Support TLS renegotiation

Found in: Component config > mbedTLS

The two main uses of renegotiation are (1) refresh keys on long-lived connections and (2) client authentication after the initial handshake. If you don't need renegotiation, disabling it will save code size and reduce the possibility of abuse/vulnerability.

MBEDTLS_SSL_PROTO_SSL3

Legacy SSL 3.0 support

Found in: Component config > mbedTLS

Support the legacy SSL 3.0 protocol. Most servers will speak a newer TLS protocol these days.

MBEDTLS_SSL_PROTO_TLS1

Support TLS 1.0 protocol

Found in: Component config > mbedTLS

MBEDTLS_SSL_PROTO_TLS1_1

Support TLS 1.1 protocol

Found in: Component config > mbedTLS

MBEDTLS_SSL_PROTO_TLS1_2

Support TLS 1.2 protocol

Found in: Component config > mbedTLS

MBEDTLS_SSL_PROTO_DTLS

Support DTLS protocol (all versions)

Found in: Component config > mbedTLS

Requires TLS 1.1 to be enabled for DTLS 1.0 Requires TLS 1.2 to be enabled for DTLS 1.2

MBEDTLS_SSL_ALPN

Support ALPN (Application Layer Protocol Negotiation)

Found in: Component config > mbedTLS

Disabling this option will save some code size if it is not needed.

MBEDTLS_SSL_SESSION_TICKETS

TLS: Support RFC 5077 SSL session tickets

Found in: Component config > mbedTLS

Support RFC 5077 session tickets. See mbedTLS documentation for more details.

Disabling this option will save some code size.

Symmetric Ciphers

MBEDTLS_AES_C

AES block cipher

Found in: Component config > mbedTLS > Symmetric Ciphers

MBEDTLS_CAMELLIA_C

Camellia block cipher

Found in: Component config > mbedTLS > Symmetric Ciphers

MBEDTLS_DES_C

DES block cipher (legacy, insecure)

Found in: Component config > mbedTLS > Symmetric Ciphers

Enables the DES block cipher to support 3DES-based TLS ciphersuites.

3DES is vulnerable to the Sweet32 attack and should only be enabled if absolutely necessary.

MBEDTLS_RC4_MODE

RC4 Stream Cipher (legacy, insecure)

Found in: Component config > mbedTLS > Symmetric Ciphers

ARCFOUR (RC4) stream cipher can be disabled entirely, enabled but not added to default ciphersuites, or enabled completely.

Please consider the security implications before enabling RC4.

Available options:

- MBEDTLS_RC4_DISABLED
- MBEDTLS_RC4_ENABLED_NO_DEFAULT
- MBEDTLS_RC4_ENABLED

MBEDTLS_BLOWFISH_C

Blowfish block cipher (read help)

Found in: Component config > mbedTLS > Symmetric Ciphers

Enables the Blowfish block cipher (not used for TLS sessions.)

The Blowfish cipher is not used for mbedTLS TLS sessions but can be used for other purposes. Read up on the limitations of Blowfish (including Sweet32) before enabling.

MBEDTLS_XTEA_C

XTEA block cipher

Found in: Component config > mbedTLS > Symmetric Ciphers

Enables the XTEA block cipher.

MBEDTLS_CCM_C

CCM (Counter with CBC-MAC) block cipher modes

Found in: Component config > mbedTLS > Symmetric Ciphers

Enable Counter with CBC-MAC (CCM) modes for AES and/or Camellia ciphers.

Disabling this option saves some code size.

MBEDTLS_GCM_C

GCM (Galois/Counter) block cipher modes

Found in: Component config > mbedTLS > Symmetric Ciphers

Enable Galois/Counter Mode for AES and/or Camellia ciphers.

This option is generally faster than CCM.

MBEDTLS_RIPEMD160_C

Enable RIPEMD-160 hash algorithm

Found in: Component config > mbedTLS

Enable the RIPEMD-160 hash algorithm.

Certificates

MBEDTLS_PEM_PARSE_C

Read & Parse PEM formatted certificates

Found in: Component config > mbedTLS > Certificates

Enable decoding/parsing of PEM formatted certificates.

If your certificates are all in the simpler DER format, disabling this option will save some code size.

MBEDTLS_PEM_WRITE_C

Write PEM formatted certificates

Found in: Component config > mbedTLS > Certificates

Enable writing of PEM formatted certificates.

If writing certificate data only in DER format, disabling this option will save some code size.

MBEDTLS_X509_CRL_PARSE_C

X.509 CRL parsing

Found in: Component config > mbedTLS > Certificates

Support for parsing X.509 Certificate Revocation Lists.

MBEDTLS_X509_CSR_PARSE_C

X.509 CSR parsing

Found in: Component config > mbedTLS > Certificates

Support for parsing X.509 Certificate Signing Requests

MBEDTLS_ECP_C

Elliptic Curve Ciphers

Found in: Component config > mbedTLS

MBEDTLS_ECDH_C

Elliptic Curve Diffie-Hellman (ECDH)

Found in: Component config > mbedTLS

Enable ECDH. Needed to use ECDHE-xxx TLS ciphersuites.

MBEDTLS_ECDSA_C

Elliptic Curve DSA

Found in: Component config > mbedTLS

Enable ECDSA. Needed to use ECDSA-xxx TLS ciphersuites.

MBEDTLS_ECP_DP_SECP192R1_ENABLED

Enable SECP192R1 curve

Found in: Component config > mbedTLS

Enable support for SECP192R1 Elliptic Curve.

MBEDTLS_ECP_DP_SECP224R1_ENABLED

Enable SECP224R1 curve

Found in: Component config > mbedTLS

Enable support for SECP224R1 Elliptic Curve.

MBEDTLS_ECP_DP_SECP256R1_ENABLED

Enable SECP256R1 curve

Found in: Component config > mbedTLS

Enable support for SECP256R1 Elliptic Curve.

MBEDTLS_ECP_DP_SECP384R1_ENABLED

Enable SECP384R1 curve

Found in: Component config > mbedTLS

Enable support for SECP384R1 Elliptic Curve.

MBEDTLS_ECP_DP_SECP521R1_ENABLED

Enable SECP521R1 curve

Found in: Component config > mbedTLS

Enable support for SECP521R1 Elliptic Curve.

MBEDTLS_ECP_DP_SECP192K1_ENABLED

Enable SECP192K1 curve

Found in: Component config > mbedTLS

Enable support for SECP192K1 Elliptic Curve.

MBEDTLS_ECP_DP_SECP224K1_ENABLED

Enable SECP224K1 curve

Found in: Component config > mbedTLS

Enable support for SECP224K1 Elliptic Curve.

MBEDTLS_ECP_DP_SECP256K1_ENABLED

Enable SECP256K1 curve

Found in: Component config > mbedTLS

Enable support for SECP256K1 Elliptic Curve.

MBEDTLS_ECP_DP_BP256R1_ENABLED

Enable BP256R1 curve

Found in: Component config > mbedTLS

support for DP Elliptic Curve.

MBEDTLS_ECP_DP_BP384R1_ENABLED

Enable BP384R1 curve

Found in: Component config > mbedTLS

support for DP Elliptic Curve.

MBEDTLS_ECP_DP_BP512R1_ENABLED

Enable BP512R1 curve

Found in: Component config > mbedTLS

support for DP Elliptic Curve.

MBEDTLS_ECP_DP_CURVE25519_ENABLED

Enable CURVE25519 curve

Found in: Component config > mbedTLS

Enable support for CURVE25519 Elliptic Curve.

MBEDTLS_ECP_NIST_OPTIM

NIST ‘modulo p’ optimisations

Found in: Component config > mbedTLS

NIST ‘modulo p’ optimisations increase Elliptic Curve operation performance.

Disabling this option saves some code size.

FAT Filesystem support

FATFS_CHOOSE_CODEPAGE

OEM Code Page

Found in: Component config > FAT Filesystem support

OEM code page used for file name encodings. Required to be set to a non-ASCII value to use Long Filenames.

Available options:

- FATFS_CODEPAGE_ASCII
- FATFS_CODEPAGE_437
- FATFS_CODEPAGE_720
- FATFS_CODEPAGE_737
- FATFS_CODEPAGE_771
- FATFS_CODEPAGE_775
- FATFS_CODEPAGE_850
- FATFS_CODEPAGE_852
- FATFS_CODEPAGE_855
- FATFS_CODEPAGE_857
- FATFS_CODEPAGE_860
- FATFS_CODEPAGE_861
- FATFS_CODEPAGE_862
- FATFS_CODEPAGE_863
- FATFS_CODEPAGE_864
- FATFS_CODEPAGE_865
- FATFS_CODEPAGE_866
- FATFS_CODEPAGE_869
- FATFS_CODEPAGE_932
- FATFS_CODEPAGE_936
- FATFS_CODEPAGE_949
- FATFS_CODEPAGE_950

FATFS_LONG_Filenames

Long filename support

Found in: Component config > FAT Filesystem support

Support long filenames in FAT. Long filename data increases memory usage. FATFS can be configured to store the buffer for long filename data in stack or heap.

Available options:

- FATFS_LFN_NONE
- FATFS_LFN_HEAP
- FATFS_LFN_STACK

FATFS_MAX_LFN

Max long filename length

Found in: Component config > FAT Filesystem support

Maximum long filename length. Can be reduced to save RAM.

SPIFFS Configuration

SPIFFS_MAX_PARTITIONS

Maximum Number of Partitions

Found in: Component config > SPIFFS Configuration

Define maximum number of partitions that can be mounted.

SPIFFS Cache Configuration

SPIFFS_CACHE

Enable SPIFFS Cache

Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration

Enables/disable memory read caching of nucleus file system operations.

SPIFFS_CACHE_WR

Enable SPIFFS Write Caching

Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration

Enables memory write caching for file descriptors in hydrogen.

SPIFFS_CACHE_STATS

Enable SPIFFS Cache Statistics

Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration

Enable/disable statistics on caching. Debug/test purpose only.

SPIFFS_PAGE_CHECK

Enable SPIFFS Page Check

Found in: Component config > SPIFFS Configuration

Always check header of each accessed page to ensure consistent state. If enabled it will increase number of reads, will increase flash.

SPIFFS_GC_MAX_RUNS

Set Maximum GC Runs

Found in: Component config > SPIFFS Configuration

Define maximum number of gc runs to perform to reach desired free pages.

SPIFFS_GC_STATS

Enable SPIFFS GC Statistics

Found in: Component config > SPIFFS Configuration

Enable/disable statistics on gc. Debug/test purpose only.

SPIFFS_OBJ_NAME_LEN

Set SPIFFS Maximum Name Length

Found in: Component config > SPIFFS Configuration

Object name maximum length. Note that this length include the zero-termination character, meaning maximum string of characters can at most be SPIFFS_OBJ_NAME_LEN - 1.

SPIFFS_USE_MAGIC

Enable SPIFFS Filesystem Magic

Found in: Component config > SPIFFS Configuration

Enable this to have an identifiable spiffs filesystem. This will look for a magic in all sectors to determine if this is a valid spiffs system or not on mount point.

SPIFFS_USE_MAGIC_LENGTH

Enable SPIFFS Filesystem Length Magic

Found in: Component config > SPIFFS Configuration

If this option is enabled, the magic will also be dependent on the length of the filesystem. For example, a filesystem configured and formatted for 4 megabytes will not be accepted for mounting with a configuration defining the filesystem as 2 megabytes.

Debug Configuration

SPIFFS_DBG

Enable general SPIFFS debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print general debug messages to the console

SPIFFS_API_DBG

Enable SPIFFS API debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print API debug mesages to the console

SPIFFS_GC_DBG

Enable SPIFFS Garbage Cleaner debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print GC debug mesages to the console

SPIFFS_CACHE_DBG

Enable SPIFFS Cache debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print Cache debug mesages to the console

SPIFFS_CHECK_DBG

Enable SPIFFS Filesystem Check debug

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enabling this option will print Filesystem Check debug mesages to the console

SPIFFS_TEST_VISUALISATION

Enable SPIFFS Filesystem Visualization

Found in: Component config > SPIFFS Configuration > Debug Configuration

Enable this option to enable SPIFFS_vis function in the api.

SPI Flash driver

SPI_FLASH_ENABLE_COUNTERS

Enable operation counters

Found in: Component config > SPI Flash driver

This option enables the following APIs:

- spi_flash_reset_counters
- spi_flash_dump_counters
- spi_flash_get_counters

These APIs may be used to collect performance data for spi_flash APIs and to help understand behaviour of libraries which use SPI flash.

SPI_FLASH_ROM_DRIVER_PATCH

Enable SPI flash ROM driver patched functions

Found in: Component config > SPI Flash driver

Enable this flag to use patched versions of SPI flash ROM driver functions. This option is needed to write to flash on ESP32-D2WD, and any configuration where external SPI flash is connected to non-default pins.

Serial flasher config

ESPTOOLPY_PORT

Default serial port

Found in: Serial flasher config

The serial port that's connected to the ESP chip. This can be overridden by setting the ESPPORT environment variable.

ESPTOOLPY_BAUD

Default baud rate

Found in: Serial flasher config

Default baud rate to use while communicating with the ESP chip. Can be overridden by setting the ESPBAUD variable.

Available options:

- ESPTOOLPY_BAUD_115200B
- ESPTOOLPY_BAUD_230400B
- ESPTOOLPY_BAUD_921600B
- ESPTOOLPY_BAUD_2MB
- ESPTOOLPY_BAUD_OTHER

ESPTOOLPY_BAUD_OTHER_VAL

Other baud rate value

Found in: Serial flasher config

ESPTOOLPY_COMPRESSED

Use compressed upload

Found in: Serial flasher config

The flasher tool can send data compressed using zlib, letting the ROM on the ESP chip decompress it on the fly before flashing it. For most payloads, this should result in a speed increase.

FLASHMODE

Flash SPI mode

Found in: Serial flasher config

Mode the flash chip is flashed in, as well as the default mode for the binary to run in.

Available options:

- FLASHMODE_QIO
- FLASHMODE_QOUT
- FLASHMODE_DIO
- FLASHMODE_DOUT

ESPTOOLPY_FLASHFREQ

Flash SPI speed

Found in: Serial flasher config

The SPI flash frequency to be used.

Available options:

- ESPTOOLPY_FLASHFREQ_80M
- ESPTOOLPY_FLASHFREQ_40M
- ESPTOOLPY_FLASHFREQ_26M

- ESPTOOLPY_FLASHFREQ_20M

ESPTOOLPY_FLASHSIZE

Flash size

Found in: Serial flasher config

SPI flash size, in megabytes

Available options:

- ESPTOOLPY_FLASHSIZE_1MB
- ESPTOOLPY_FLASHSIZE_2MB
- ESPTOOLPY_FLASHSIZE_4MB
- ESPTOOLPY_FLASHSIZE_8MB
- ESPTOOLPY_FLASHSIZE_16MB

ESPTOOLPY_FLASHSIZE_DETECT

Detect flash size when flashing bootloader

Found in: Serial flasher config

If this option is set, ‘make flash’ targets will automatically detect the flash size and update the bootloader image when flashing.

ESPTOOLPY_BEFORE

Before flashing

Found in: Serial flasher config

Configure whether esptool.py should reset the ESP32 before flashing.

Automatic resetting depends on the RTS & DTR signals being wired from the serial port to the ESP32. Most USB development boards do this internally.

Available options:

- ESPTOOLPY_BEFORE_RESET
- ESPTOOLPY_BEFORE_NORESET

ESPTOOLPY_AFTER

After flashing

Found in: Serial flasher config

Configure whether esptool.py should reset the ESP32 after flashing.

Automatic resetting depends on the RTS & DTR signals being wired from the serial port to the ESP32. Most USB development boards do this internally.

Available options:

- ESPTOOLPY_AFTER_RESET
- ESPTOOLPY_AFTER_NORESET

MONITOR_BAUD

‘make monitor’ baud rate

Found in: Serial flasher config

Baud rate to use when running ‘make monitor’ to view serial output from a running chip.

Can override by setting the MONITORBAUD environment variable.

Available options:

- MONITOR_BAUD_9600B
- MONITOR_BAUD_57600B
- MONITOR_BAUD_115200B
- MONITOR_BAUD_230400B
- MONITOR_BAUD_921600B
- MONITOR_BAUD_2MB
- MONITOR_BAUD_OTHER

MONITOR_BAUD_OTHER_VAL

Custom baud rate value

Found in: Serial flasher config

tcpip adapter

IP_LOST_TIMER_INTERVAL

IP Address lost timer interval (seconds)

Found in: Component config > tcpip adapter

The value of 0 indicates the IP lost timer is disabled, otherwise the timer is enabled.

The IP address may be lost because of some reasons, e.g. when the station disconnects from soft-AP, or when DHCP IP renew fails etc. If the IP lost timer is enabled, it will be started everytime the IP is lost. Event SYSTEM_EVENT_STA_LOST_IP will be raised if the timer expires. The IP lost timer is stopped if the station get the IP again before the timer expires.

2.8.4 Customisations

Because IDF builds by default with [Warning On Undefined Variables](#), when the Kconfig tool generates Makefiles (the `auto.conf` file) its behaviour has been customised. In normal Kconfig, a variable which is set to “no” is undefined. In IDF’s version of Kconfig, this variable is defined in the Makefile but has an empty value.

(Note that `ifdef` and `ifndef` can still be used in Makefiles, because they test if a variable is defined *and has a non-empty value*.)

When generating header files for C & C++, the behaviour is not customised - so `#ifdef` can be used to test if a boolean config item is set or not.

CHAPTER 3

ESP32 Hardware Reference

3.1 ESP32 Modules and Boards

Espressif designed and manufactured several development modules and boards to help users evaluate functionality of ESP32 chip. 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.

3.1.1 ESP-WROOM-32

The smallest module intended for installation in final products. Can be also used for evaluation after adding extra components like programming interface, boot strapping resistors and break out headers.



Fig. 3.1: ESP-WROOM-32 module (front and back)

- [Schematic \(PDF\)](#)
- [Datasheet \(PDF\)](#)

- ESP32 Module Reference Design (ZIP) containing OrCAD schematic, PCB layout, gerbers and BOM

3.1.2 ESP32-WROVER

A step upgrade of ESP-WROOM-32 described above with an additional 4 MB SPI PSRAM (Pseudo static RAM). Module is provided in two versions: ‘ESP32-WROVER’ with PCB antenna (shown below) and ‘ESP32-WROVER-I’ with an IPEX antenna.

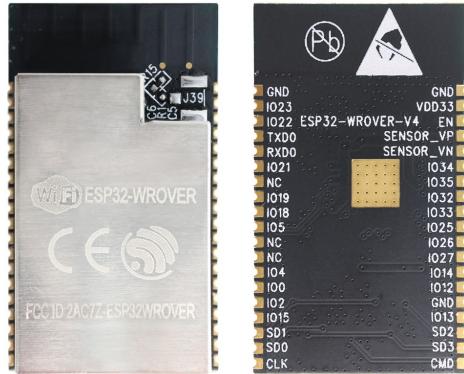


Fig. 3.2: ESP32-WROVER module (front and back)

- [Datasheet \(PDF\)](#)
- [ESP-PSRAM32 Datasheet \(PDF\)](#)

3.1.3 ESP32 Core Board V2 / ESP32 DevKitC

Small and convenient development board with ESP-WROOM-32 module installed, break out pin headers and minimum additional components. Includes USB to serial programming interface, that also provides power supply for the board. Has press buttons to reset the board and put it in upload mode.

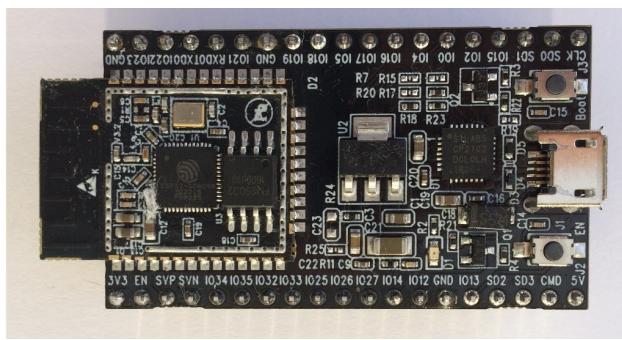


Fig. 3.3: ESP32 Core Board V2 / ESP32 DevKitC board

- [Schematic \(PDF\)](#)
- [ESP32 Development Board Reference Design \(ZIP\)](#) containing OrCAD schematic, PCB layout, gerbers and BOM
- [ESP32-DevKitC Getting Started Guide](#)
- [CP210x USB to UART Bridge VCP Drivers](#)

3.1.4 ESP32 Demo Board V2

One of first feature rich evaluation boards that contains several pin headers, dip switches, USB to serial programming interface, reset and boot mode press buttons, power switch, 10 touch pads and separate header to connect LCD screen.

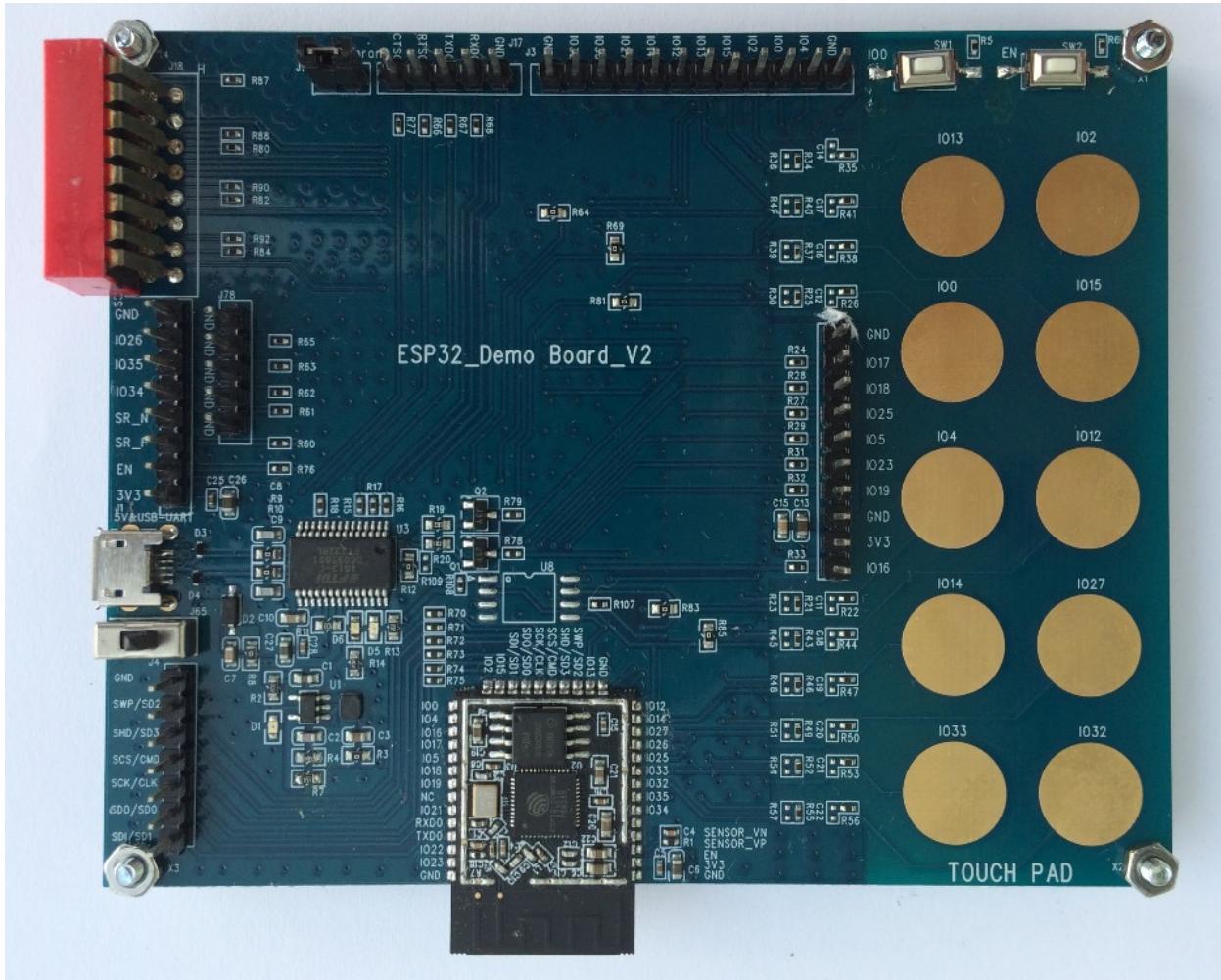


Fig. 3.4: ESP32 Demo Board V2

- Schematic (PDF)
- FTDI Virtual COM Port Drivers

3.1.5 ESP-WROVER-KIT

This section describes several revisions of ESP-WROVER-KIT development board.

All versions of ESP-WROVER-KIT are ready to accommodate an *ESP-WROOM-32* or *ESP32-WROVER* module.

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.

ESP-WROVER-KIT V1 / ESP32 DevKitJ V1

First version of ESP-WROVER-KIT. Shipped with ESP-WROOM-32 on board.

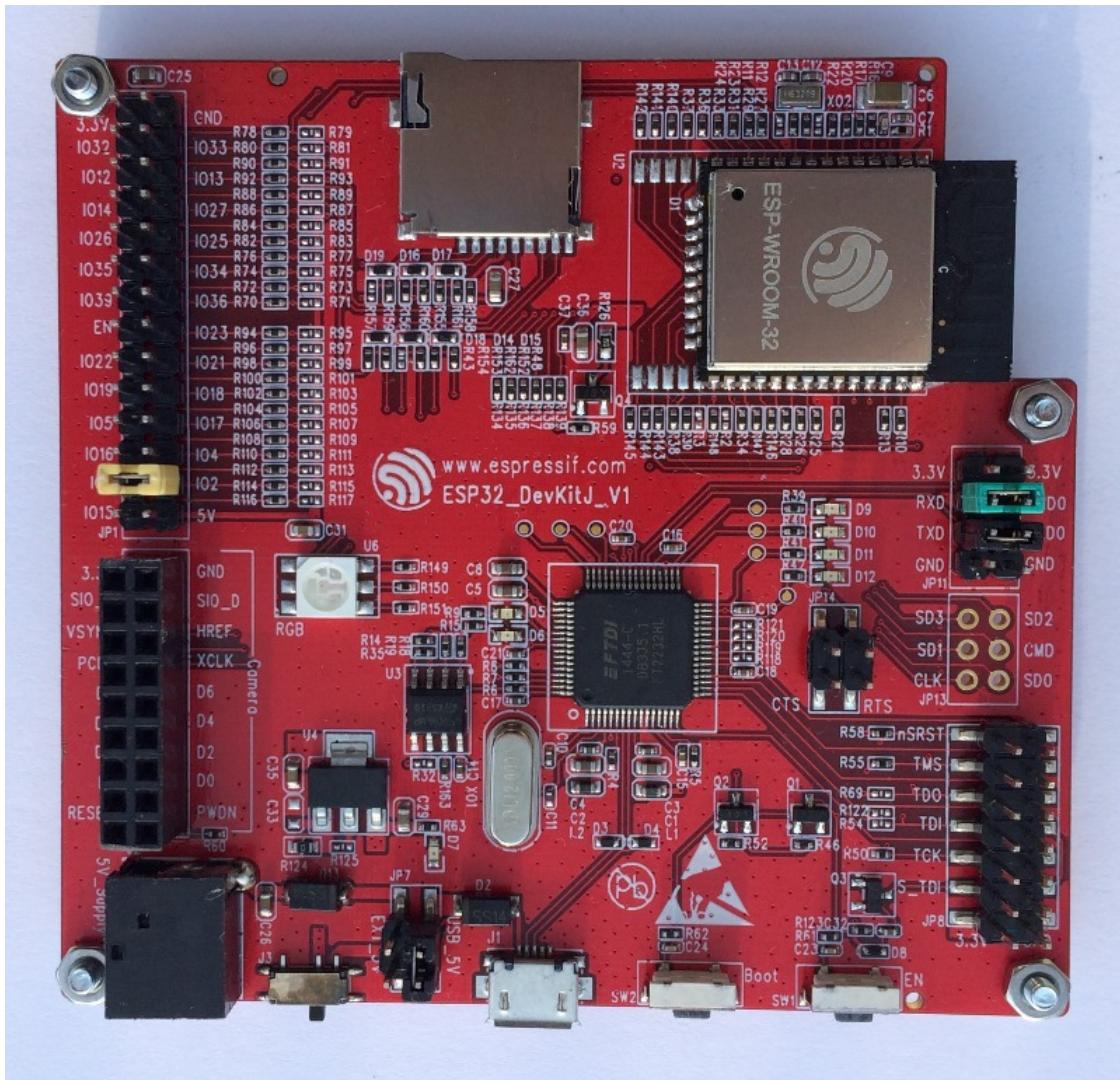


Fig. 3.5: ESP-WROVER-KIT V1 / ESP32 DevKitJ V1 board

The board has red soldermask.

- [Schematic \(PDF\)](#)
- [JTAG Debugging](#)
- [FTDI Virtual COM Port Drivers](#)

ESP-WROVER-KIT V2

This is updated version of ESP32 DevKitJ V1 described above with design improvements identified when DevKitJ was in use, e.g. improved support for SD card. By default board has ESP-WROOM-32 module installed.

Comparing to previous version, this board has a shiny black finish and a male camera header.

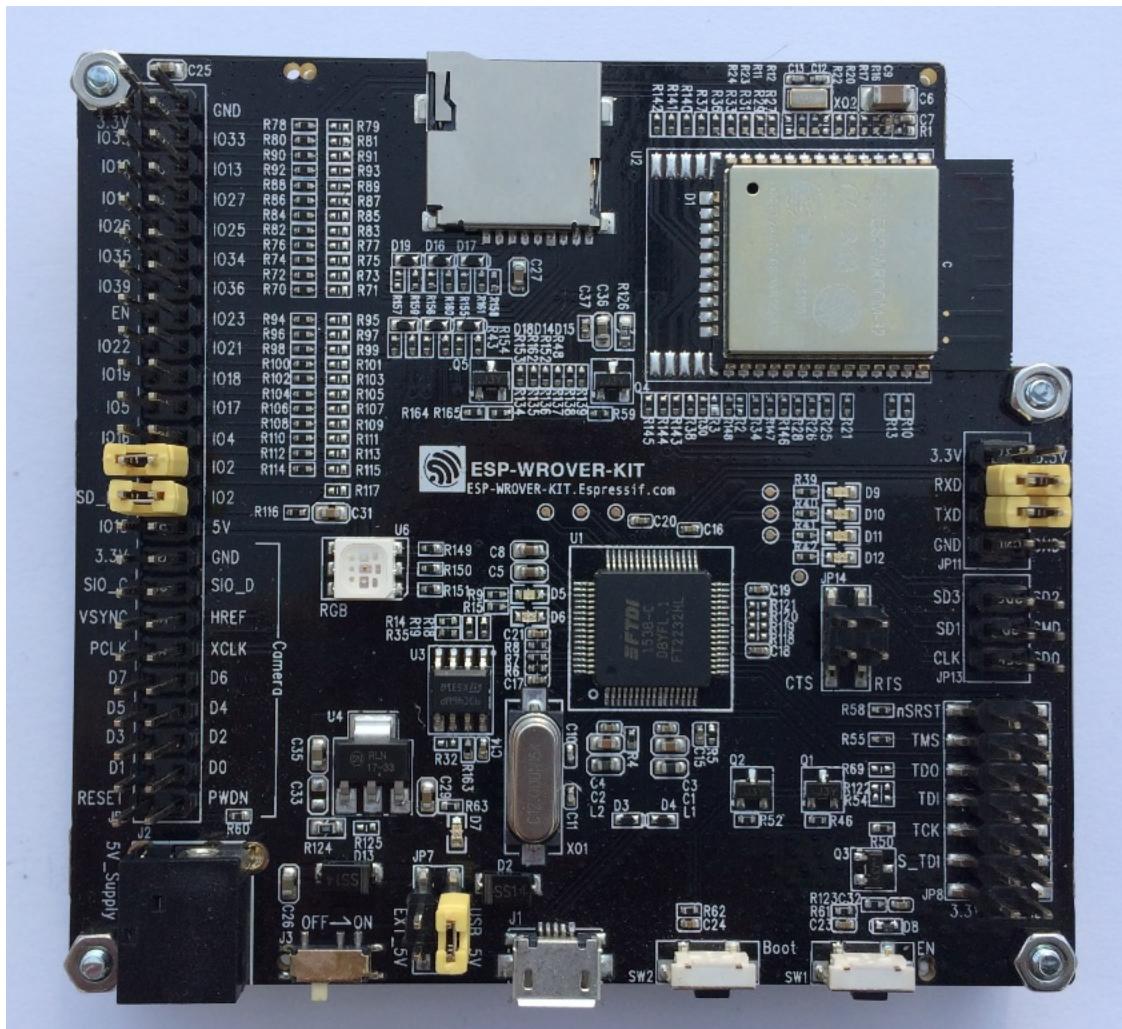


Fig. 3.6: ESP-WROVER-KIT V2 board

- Schematic V2 (PDF)
- *ESP-WROVER-KIT V2 Getting Started Guide*
- *JTAG Debugging*
- FTDI Virtual COM Port Drivers

ESP-WROVER-KIT V3

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.

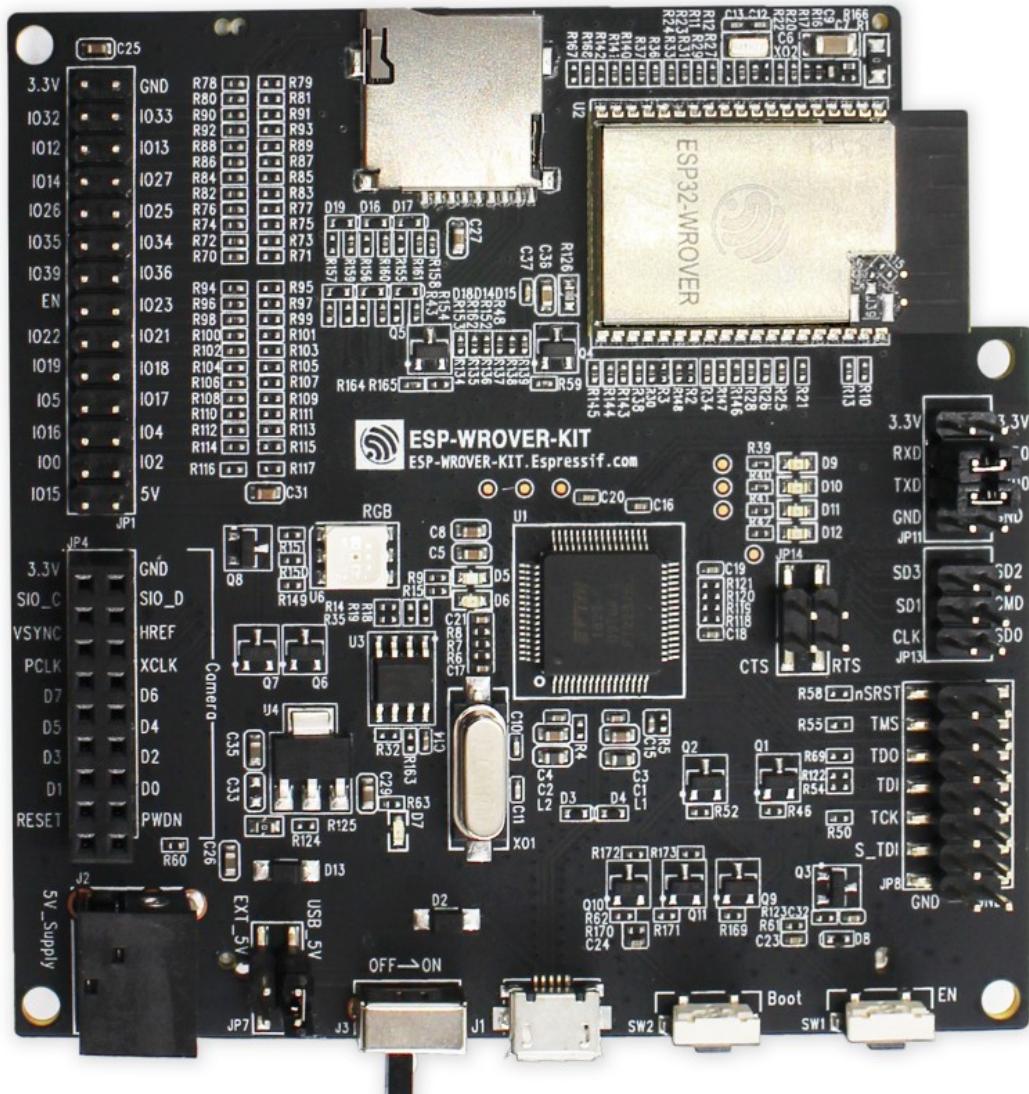


Fig. 3.7: ESP-WROVER-KIT V3 board

The camera header has been changed from male back to female. The board soldermask is matte black. The board on picture above has ESP32-WROVER is installed.

- [Schematic V3 \(PDF\)](#)
- [*ESP-WROVER-KIT V3 Getting Started Guide*](#)
- [*JTAG Debugging*](#)
- [FTDI Virtual COM Port Drivers](#)

4.1 General Notes About ESP-IDF Programming

4.1.1 Application startup flow

This note explains various steps which happen before `app_main` function of an ESP-IDF application is called.

The high level view of startup process is as follows:

1. First-stage bootloader in ROM loads second-stage bootloader image to RAM (IRAM & DRAM) from flash offset 0x1000.
2. Second-stage bootloader loads partition table and main app image from flash. Main app incorporates both RAM segments and read-only segments mapped via flash cache.
3. Main app image executes. At this point the second CPU and RTOS scheduler can be started.

This process is explained in detail in the following sections.

First stage bootloader

After SoC reset, PRO CPU will start running immediately, executing reset vector code, while APP CPU will be held in reset. During startup process, PRO CPU does all the initialization. APP CPU reset is de-asserted in the `call_start_cpu0` function of application startup code. Reset vector code is located at address 0x40000400 in the mask ROM of the ESP32 chip and can not be modified.

Startup code called from the reset vector determines the boot mode by checking `GPIO_STRAP_REG` register for bootstrap pin states. Depending on the reset reason, the following takes place:

1. Reset from deep sleep: if the value in `RTC_CNTL_STORE6_REG` is non-zero, and CRC value of RTC memory in `RTC_CNTL_STORE7_REG` is valid, use `RTC_CNTL_STORE6_REG` as an entry point address and jump immediately to it. If `RTC_CNTL_STORE6_REG` is zero, or `RTC_CNTL_STORE7_REG` contains invalid CRC, or once the code called via `RTC_CNTL_STORE6_REG` returns, proceed with boot as if it was a power-on reset.

Note: to run customized code at this point, a deep sleep stub mechanism is provided. Please see [deep sleep](#) documentation for this.

2. For power-on reset, software SOC reset, and watchdog SOC reset: check the `GPIO_STRAP_REG` register if UART or SDIO download mode is requested. If this is the case, configure UART or SDIO, and wait for code to be downloaded. Otherwise, proceed with boot as if it was due to software CPU reset.
3. For software CPU reset and watchdog CPU reset: configure SPI flash based on EFUSE values, and attempt to load the code from flash. This step is described in more detail in the next paragraphs. If loading code from flash fails, unpack BASIC interpreter into the RAM and start it. Note that RTC watchdog is still enabled when this happens, so unless any input is received by the interpreter, watchdog will reset the SOC in a few hundred milliseconds, repeating the whole process. If the interpreter receives any input from the UART, it disables the watchdog.

Application binary image is loaded from flash starting at address 0x1000. First 4kB sector of flash is used to store secure boot IV and signature of the application image. Please check secure boot documentation for details about this.

Second stage bootloader

In ESP-IDF, the binary image which resides at offset 0x1000 in flash is the second stage bootloader. Second stage bootloader source code is available in components/bootloader directory of ESP-IDF. Note that this arrangement is not the only one possible with the ESP32 chip. It is possible to write a fully featured application which would work when flashed to offset 0x1000, but this is out of scope of this document. Second stage bootloader is used in ESP-IDF to add flexibility to flash layout (using partition tables), and allow for various flows associated with flash encryption, secure boot, and over-the-air updates (OTA) to take place.

When the first stage bootloader is finished checking and loading the second stage bootloader, it jumps to the second stage bootloader entry point found in the binary image header.

Second stage bootloader reads the partition table found at offset 0x8000. See [partition tables](#) documentation for more information. The bootloader finds factory and OTA partitions, and decides which one to boot based on data found in *OTA info* partition.

For the selected partition, second stage bootloader copies data and code sections which are mapped into IRAM and DRAM to their load addresses. For sections which have load addresses in DROM and IROM regions, flash MMU is configured to provide the correct mapping. Note that the second stage bootloader configures flash MMU for both PRO and APP CPUs, but it only enables flash MMU for PRO CPU. Reason for this is that second stage bootloader code is loaded into the memory region used by APP CPU cache. The duty of enabling cache for APP CPU is passed on to the application. Once code is loaded and flash MMU is set up, second stage bootloader jumps to the application entry point found in the binary image header.

Currently it is not possible to add application-defined hooks to the bootloader to customize application partition selection logic. This may be required to load different application image depending on a state of a GPIO, for example. Such customization features will be added to ESP-IDF in the future. For now, bootloader can be customized by copying bootloader component into application directory and making necessary changes there. ESP-IDF build system will compile the component in application directory instead of ESP-IDF components directory in this case.

Application startup

ESP-IDF application entry point is `call_start_cpu0` function found in `components/esp32/cpu_start.c`. Two main things this function does are to enable heap allocator and to make APP CPU jump to its entry point, `call_start_cpu1`. The code on PRO CPU sets the entry point for APP CPU, de-asserts APP CPU reset, and waits for a global flag to be set by the code running on APP CPU, indicating that it has started. Once this is done, PRO CPU jumps to `start_cpu0` function, and APP CPU jumps to `start_cpu1` function.

Both `start_cpu0` and `start_cpu1` are weak functions, meaning that they can be overridden in the application, if some application-specific change to initialization sequence is needed. Default implementation of `start_cpu0` enables or initializes components depending on choices made in `menuconfig`. Please see source code of this function in `components/esp32/cpu_start.c` for an up to date list of steps performed. Note that any C++ global

constructors present in the application will be called at this stage. Once all essential components are initialized, *main task* is created and FreeRTOS scheduler is started.

While PRO CPU does initialization in `start_cpu0` function, APP CPU spins in `start_cpul` function, waiting for the scheduler to be started on the PRO CPU. Once the scheduler is started on the PRO CPU, code on the APP CPU starts the scheduler as well.

Main task is the task which runs `app_main` function. Main task stack size and priority can be configured in `menuconfig`. Application can use this task for initial application-specific setup, for example to launch other tasks. Application can also use main task for event loops and other general purpose activities. If `app_main` function returns, main task is deleted.

4.1.2 Application memory layout

ESP32 chip has flexible memory mapping features. This section describes how ESP-IDF uses these features by default.

Application code in ESP-IDF can be placed into one of the following memory regions.

IRAM (instruction RAM)

ESP-IDF allocates part of *Internal SRAM0* region (defined in the Technical Reference Manual) for instruction RAM. Except for the first 64 kB block which is used for PRO and APP CPU caches, the rest of this memory range (i.e. from 0x40080000 to 0x400A0000) is used to store parts of application which need to run from RAM.

A few components of ESP-IDF and parts of WiFi stack are placed into this region using the linker script.

If some application code needs to be placed into IRAM, it can be done using `IRAM_ATTR` define:

```
#include "esp_attr.h"

void IRAM_ATTR gpio_isr_handler(void* arg)
{
    // ...
}
```

Here are the cases when parts of application may or should be placed into IRAM.

- Interrupt handlers must be placed into IRAM if `ESP_INTR_FLAG_IRAM` is used when registering the interrupt handler. In this case, ISR may only call functions placed into IRAM or functions present in ROM. *Note 1:* all FreeRTOS APIs are currently placed into IRAM, so are safe to call from interrupt handlers. If the ISR is placed into IRAM, all constant data used by the ISR and functions called from ISR (including, but not limited to, `const char` arrays), must be placed into DRAM using `DRAM_ATTR`.
- Some timing critical code may be placed into IRAM to reduce the penalty associated with loading the code from flash. ESP32 reads code and data from flash via a 32 kB cache. In some cases, placing a function into IRAM may reduce delays caused by a cache miss.

IROM (code executed from Flash)

If a function is not explicitly placed into IRAM or RTC memory, it is placed into flash. The mechanism by which Flash MMU is used to allow code execution from flash is described in the Technical Reference Manual. ESP-IDF places the code which should be executed from flash starting from the beginning of 0x400D0000 — 0x40400000 region. Upon startup, second stage bootloader initializes Flash MMU to map the location in flash where code is located into the beginning of this region. Access to this region is transparently cached using two 32kB blocks in 0x40070000 — 0x40080000 range.

Note that the code outside `0x40000000 -- 0x40400000` region may not be reachable with Window ABI `CALLx` instructions, so special care is required if `0x40400000 -- 0x40800000` or `0x40800000 -- 0x40C00000` regions are used by the application. ESP-IDF doesn't use these regions by default.

RTC fast memory

The code which has to run after wake-up from deep sleep mode has to be placed into RTC memory. Please check detailed description in *deep sleep* documentation.

DRAM (data RAM)

Non-constant static data and zero-initialized data is placed by the linker into the 256 kB `0x3FFB0000 -- 0x3FFF0000` region. Note that this region is reduced by 64kB (by shifting start address to `0x3FFC0000`) if Bluetooth stack is used. Length of this region is also reduced by 16 kB or 32kB if trace memory is used. All space which is left in this region after placing static data there is used for the runtime heap.

Constant data may also be placed into DRAM, for example if it is used in an ISR (see notes in IRAM section above). To do that, `DRAM_ATTR` define can be used:

```
DRAM_ATTR const char[] format_string = "%p %x";
char buffer[64];
sprintf(buffer, format_string, ptr, val);
```

Needless to say, it is not advised to use `printf` and other output functions in ISRs. For debugging purposes, use `ESP_EARLY_LOGx` macros when logging from ISRs. Make sure that both `TAG` and format string are placed into DRAM in that case.

DROM (data stored in Flash)

By default, constant data is placed by the linker into a 4 MB region (`0x3F400000 -- 0x3F800000`) which is used to access external flash memory via Flash MMU and cache. Exceptions to this are literal constants which are embedded by the compiler into application code.

RTC slow memory

Global and static variables used by code which runs from RTC memory (i.e. deep sleep stub code) must be placed into RTC slow memory. Please check detailed description in *deep sleep* documentation.

4.2 Build System

This document explains the Espressif IoT Development Framework build system and the concept of “components”

Read this document if you want to know how to organise a new ESP-IDF project.

We recommend using the `esp-idf-template` project as a starting point for your project.

4.2.1 Using the Build System

The esp-idf README file contains a description of how to use the build system to build your project.

4.2.2 Overview

An ESP-IDF project can be seen as an amalgamation of a number of components. For example, for a webserver that shows the current humidity, there could be:

- The ESP32 base libraries (libc, rom bindings etc)
- The WiFi drivers
- A TCP/IP stack
- The FreeRTOS operating system
- A webserver
- A driver for the humidity sensor
- Main code tying it all together

ESP-IDF makes these components explicit and configurable. To do that, when a project is compiled, the build environment will look up all the components in the ESP-IDF directories, the project directories and (optionally) in additional custom component directories. It then allows the user to configure the ESP-IDF project using a text-based menu system to customize each component. After the components in the project are configured, the build process will compile the project.

Concepts

- A “project” is a directory that contains all the files and configuration to build a single “app” (executable), as well as additional supporting output such as a partition table, data/filesystem partitions, and a bootloader.
- “Project configuration” is held in a single file called `sdkconfig` in the root directory of the project. This configuration file is modified via `make menuconfig` to customise the configuration of the project. A single project contains exactly one project configuration.
- An “app” is an executable which is built by `esp-idf`. A single project will usually build two apps - a “project app” (the main executable, ie your custom firmware) and a “bootloader app” (the initial bootloader program which launches the project app).
- “components” are modular pieces of standalone code which are compiled into static libraries (.a files) and linked into an app. Some are provided by `esp-idf` itself, others may be sourced from other places.

Some things are not part of the project:

- “ESP-IDF” is not part of the project. Instead it is standalone, and linked to the project via the `IDF_PATH` environment variable which holds the path of the `esp-idf` directory. This allows the IDF framework to be decoupled from your project.
- The toolchain for compilation is not part of the project. The toolchain should be installed in the system command line `PATH`, or the path to the toolchain can be set as part of the compiler prefix in the project configuration.

Example Project

An example project directory tree might look like this:

```
- myProject/
    - Makefile
    - sdkconfig
    - components/
        - component1/
            - component.mk
            - Kconfig
            - src1.c
```

```

        - component2/
          - component.mk
          - Kconfig
          - src1.c
          - include/
            - component2.h
      - main/
        - src1.c
        - src2.c
        - component.mk

    - build/

```

This example “myProject” contains the following elements:

- A top-level project Makefile. This Makefile set the PROJECT_NAME variable and (optionally) defines other project-wide make variables. It includes the core \$(IDF_PATH)/make/project.mk makefile which implements the rest of the ESP-IDF build system.
- “sdkconfig” project configuration file. This file is created/updated when “make menuconfig” runs, and holds configuration for all of the components in the project (including esp-idf itself). The “sdkconfig” file may or may not be added to the source control system of the project.
- Optional “components” directory contains components that are part of the project. A project does not have to contain custom components of this kind, but it can be useful for structuring reusable code or including third party components that aren’t part of ESP-IDF.
- “main” directory is a special “pseudo-component” that contains source code for the project itself. “main” is a default name, the Makefile variable COMPONENT_DIRS includes this component but you can modify this variable (or set EXTRA_COMPONENT_DIRS) to look for components in other places.
- “build” directory is where build output is created. After the make process is run, this directory will contain interim object files and libraries as well as final binary output files. This directory is usually not added to source control or distributed with the project source code.

Component directories contain a component makefile - component.mk. This may contain variable definitions to control the build process of the component, and its integration into the overall project. See *Component Makefiles* for more details.

Each component may also include a Kconfig file defining the *component configuration* options that can be set via the project configuration. Some components may also include Kconfig.projbuild and Makefile.projbuild files, which are special files for *overriding parts of the project*.

Project Makefiles

Each project has a single Makefile that contains build settings for the entire project. By default, the project Makefile can be quite minimal.

Minimal Example Makefile

```

PROJECT_NAME := myProject

include $(IDF_PATH)/make/project.mk

```

Mandatory Project Variables

- PROJECT_NAME: Name of the project. Binary output files will use this name - ie myProject.bin, myProject.elf.

Optional Project Variables

These variables all have default values that can be overridden for custom behaviour. Look in `make/project.mk` for all of the implementation details.

- `PROJECT_PATH`: Top-level project directory. Defaults to the directory containing the Makefile. Many other project variables are based on this variable. The project path cannot contain spaces.
- `BUILD_DIR_BASE`: The build directory for all objects/libraries/binaries. Defaults to `$ (PROJECT_PATH) / build`.
- `COMPONENT_DIRS`: Directories to search for components. Defaults to `$(IDF_PATH)/components`, `$(PROJECT_PATH)/components`, `$(PROJECT_PATH) / main` and `EXTRA_COMPONENT_DIRS`. Override this variable if you don't want to search for components in these places.
- `EXTRA_COMPONENT_DIRS`: Optional list of additional directories to search for components.
- `COMPONENTS`: A list of component names to build into the project. Defaults to all components found in the `COMPONENT_DIRS` directories.

Any paths in these Makefile variables should be absolute paths. You can convert relative paths using `$(PROJECT_PATH) / xxx`, `$(IDF_PATH) / xxx`, or use the Make function `$(abspath xxx)`.

These variables should all be set before the line `include $(IDF_PATH) / make/project.mk` in the Makefile.

Component Makefiles

Each project contains one or more components, which can either be part of esp-idf or added from other component directories.

A component is any directory that contains a `component.mk` file.

Searching for Components

The list of directories in `COMPONENT_DIRS` is searched for the project's components. Directories in this list can either be components themselves (ie they contain a `component.mk` file), or they can be top-level directories whose subdirectories are components.

Running the `make list-components` target dumps many of these variables and can help debug the discovery of component directories.

Multiple components with the same name

When esp-idf is collecting all the components to compile, it will do this in the order specified by `COMPONENT_DIRS`; by default, this means the idf components first, the project components second and optionally the components in `EXTRA_COMPONENT_DIRS` last. If two or more of these directories contain component subdirectories with the same name, the component in the last place searched is used. This allows, for example, overriding esp-idf components with a modified version by simply copying the component from the esp-idf component directory to the project component tree and then modifying it there. If used in this way, the esp-idf directory itself can remain untouched.

Minimal Component Makefile

The minimal `component.mk` file is an empty file(!). If the file is empty, the default component behaviour is set:

- All source files in the same directory as the makefile (*.c, *.cpp, *.S) will be compiled into the component library
- A sub-directory “include” will be added to the global include search path for all other components.
- The component library will be linked into the project app.

See *example component makefiles* for more complete component makefile examples.

Note that there is a difference between an empty `component.mk` file (which invokes default component build behaviour) and no `component.mk` file (which means no default component build behaviour will occur.) It is possible for a component to have no `component.mk` file, if it only contains other files which influence the project configuration or build process.

Preset Component Variables

The following component-specific variables are available for use inside `component.mk`, but should not be modified:

- `COMPONENT_PATH`: The component directory. Evaluates to the absolute path of the directory containing `component.mk`. The component path cannot contain spaces.
- `COMPONENT_NAME`: Name of the component. Defaults to the name of the component directory.
- `COMPONENT_BUILD_DIR`: The component build directory. Evaluates to the absolute path of a directory inside `$(BUILD_DIR_BASE)` where this component’s source files are to be built. This is also the Current Working Directory any time the component is being built, so relative paths in make targets, etc. will be relative to this directory.
- `COMPONENT_LIBRARY`: Name of the static library file (relative to the component build directory) that will be built for this component. Defaults to `$(COMPONENT_NAME).a`.

The following variables are set at the project level, but exported for use in the component build:

- `PROJECT_NAME`: Name of the project, as set in project Makefile
- `PROJECT_PATH`: Absolute path of the project directory containing the project Makefile.
- `COMPONENTS`: Name of all components that are included in this build.
- `CONFIG_*`: Each value in the project configuration has a corresponding variable available in make. All names begin with `CONFIG_`.
- `CC, LD, AR, OBJCOPY`: Full paths to each tool from the gcc xtensa cross-toolchain.
- `HOSTCC, HOSTLD, HOSTAR`: Full names of each tool from the host native toolchain.
- `IDF_VER`: Git version of ESP-IDF (produced by `git describe`)

If you modify any of these variables inside `component.mk` then this will not prevent other components from building but it may make your component hard to build and/or debug.

Optional Project-Wide Component Variables

The following variables can be set inside `component.mk` to control build settings across the entire project:

- `COMPONENT_ADD_INCLUDEDIRS`: Paths, relative to the component directory, which will be added to the include search path for all components in the project. Defaults to `include` if not overridden. If an include directory is only needed to compile this specific component, add it to `COMPONENT_PRIV_INCLUDEDIRS` instead.

- **COMPONENT_ADD_LDFLAGS**: Add linker arguments to the LDFLAGS for the app executable. Defaults to `-l$ (COMPONENT_NAME)`. If adding pre-compiled libraries to this directory, add them as absolute paths - ie `$(COMPONENT_PATH)/libwhatever.a`
- **COMPONENT_DEPENDS**: Optional list of component names that should be compiled before this component. This is not necessary for link-time dependencies, because all component include directories are available at all times. It is necessary if one component generates an include file which you then want to include in another component. Most components do not need to set this variable.
- **COMPONENT_ADD_LINKER_DEPS**: Optional list of component-relative paths to files which should trigger a re-link of the ELF file if they change. Typically used for linker script files and binary libraries. Most components do not need to set this variable.

The following variable only works for components that are part of esp-idf itself:

- **COMPONENT_SUBMODULES**: Optional list of git submodule paths (relative to COMPONENT_PATH) used by the component. These will be checked (and initialised if necessary) by the build process. This variable is ignored if the component is outside the IDF_PATH directory.

Optional Component-Specific Variables

The following variables can be set inside `component.mk` to control the build of that component:

- **COMPONENT_PRIV_INCLUDEDIRS**: Directory paths, must be relative to the component directory, which will be added to the include search path for this component's source files only.
- **COMPONENT_EXTRA_INCLUDES**: Any extra include paths used when compiling the component's source files. These will be prefixed with '`-I`' and passed as-is to the compiler. Similar to the `COMPONENT_PRIV_INCLUDEDIRS` variable, except these paths are not expanded relative to the component directory.
- **COMPONENT_SRCDIRS**: Directory paths, must be relative to the component directory, which will be searched for source files (`*.cpp`, `*.c`, `*.S`). Defaults to '`.`', ie the component directory itself. Override this to specify a different list of directories which contain source files.
- **COMPONENT_OBJS**: Object files to compile. Default value is a `.o` file for each source file that is found in `COMPONENT_SRCDIRS`. Overriding this list allows you to exclude source files in `COMPONENT_SRCDIRS` that would otherwise be compiled. See *Specifying source files*
- **COMPONENT_EXTRA_CLEAN**: Paths, relative to the component build directory, of any files that are generated using custom make rules in the `component.mk` file and which need to be removed as part of `make clean`. See *Source Code Generation* for an example.
- **COMPONENT_OWNBUILDTARGET & COMPONENT_OWNCLEANTARGET**: These targets allow you to fully override the default build behaviour for the component. See *Fully Overriding The Component Makefile* for more details.
- **COMPONENT_CONFIG_ONLY**: If set, this flag indicates that the component produces no built output at all (ie `COMPONENT_LIBRARY` is not built), and most other component variables are ignored. This flag is used for IDF internal components which contain only `KConfig.projbuild` and/or `Makefile.projbuild` files to configure the project, but no source files.
- **CFLAGS**: Flags passed to the C compiler. A default set of `CFLAGS` is defined based on project settings. Component-specific additions can be made via `CFLAGS +=`. It is also possible (although not recommended) to override this variable completely for a component.
- **CPPFLAGS**: Flags passed to the C preprocessor (used for `.c`, `.cpp` and `.S` files). A default set of `CPPFLAGS` is defined based on project settings. Component-specific additions can be made via `CPPFLAGS +=`. It is also possible (although not recommended) to override this variable completely for a component.

- `CXXFLAGS`: Flags passed to the C++ compiler. A default set of `CXXFLAGS` is defined based on project settings. Component-specific additions can be made via `CXXFLAGS +=`. It is also possible (although not recommended) to override this variable completely for a component.

To apply compilation flags to a single source file, you can add a variable override as a target, ie:

```
apps/dhcpserver.o: CFLAGS += -Wno-unused-variable
```

This can be useful if there is upstream code that emits warnings.

Component Configuration

Each component can also have a `Kconfig` file, alongside `component.mk`. This contains configuration settings to add to the “make menuconfig” for this component.

These settings are found under the “Component Settings” menu when menuconfig is run.

To create a component KConfig file, it is easiest to start with one of the KConfig files distributed with esp-idf.

For an example, see *Adding conditional configuration*.

Preprocessor Definitions

ESP-IDF build systems adds the following C preprocessor definitions on the command line:

- `ESP_PLATFORM` — Can be used to detect that build happens within ESP-IDF.
- `IDF_VER` — Defined to a git version string. E.g. `v2.0` for a tagged release or `v1.0-275-g0efaa4f` for an arbitrary commit.

Build Process Internals

Top Level: Project Makefile

- “make” is always run from the project directory and the project makefile, typically named `Makefile`.
- The project makefile sets `PROJECT_NAME` and optionally customises other *optional project variables*
- The project makefile includes `$(IDF_PATH)/make/project.mk` which contains the project-level Make logic.
- `project.mk` fills in default project-level make variables and includes make variables from the project configuration. If the generated makefile containing project configuration is out of date, then it is regenerated (via targets in `project_config.mk`) and then the make process restarts from the top.
- `project.mk` builds a list of components to build, based on the default component directories or a custom list of components set in *optional project variables*.
- Each component can set some *optional project-wide component variables*. These are included via generated makefiles named `component_project_vars.mk` - there is one per component. These generated makefiles are included into `project.mk`. If any are missing or out of date, they are regenerated (via a recursive make call to the component makefile) and then the make process restarts from the top.
- `Makefile.projbuild` files from components are included into the make process, to add extra targets or configuration.
- By default, the project makefile also generates top-level build & clean targets for each component and sets up `app` and `clean` targets to invoke all of these sub-targets.

- In order to compile each component, a recursive make is performed for the component makefile.

To better understand the project make process, have a read through the `project.mk` file itself.

Second Level: Component Makefiles

- Each call to a component makefile goes via the `$ (IDF_PATH) /make/component_wrapper.mk` wrapper makefile.
- 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 by disabling `CONFIG_MAKE_WARN_UNDEFINED_VARIABLES`.

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*.

Configuration-Only Components

Some special components which contain no source files, only `Kconfig.projbuild` and `Makefile.projbuild`, may set the flag `COMPONENT_CONFIG_ONLY` in the `component.mk` file. If this flag is set, most other component variables are ignored and no build step is run for the component.

Example Component Makefiles

Because the build environment tries to set reasonable defaults that will work most of the time, `component.mk` can be very small or even empty (see *Minimal Component Makefile*). However, overriding *component variables* is usually required for some functionality.

Here are some more advanced examples of `component.mk` makefiles:

Adding source directories

By default, sub-directories are ignored. If your project has sources in sub-directories instead of in the root of the component then you can tell that to the build system by setting COMPONENT_SRCDIRS:

```
COMPONENT_SRCDIRS := src1 src2
```

This will compile all source files in the src1/ and src2/ sub-directories instead.

Specifying source files

The standard component.mk logic adds all .S and .c files in the source directories as sources to be compiled unconditionally. It is possible to circumvent that logic and hard-code the objects to be compiled by manually setting the COMPONENT_OBJS variable to the name of the objects that need to be generated:

```
COMPONENT_OBJS := file1.o file2.o thing/filea.o thing/fileb.o anotherthing/main.o
COMPONENT_SRCDIRS := . thing anotherthing
```

Note that COMPONENT_SRCDIRS must be set as well.

Adding conditional configuration

The configuration system can be used to conditionally compile some files depending on the options selected in make menuconfig. For this, ESP-IDF has the compile_only_if and compile_only_if_not macros:

Kconfig:

```
config FOO_ENABLE_BAR
    bool "Enable the BAR feature."
    help
        This enables the BAR feature of the FOO component.
```

component.mk:

```
$(call compile_only_if,$(CONFIG_FOO_ENABLE_BAR),bar.o)
```

As can be seen in the example, the compile_only_if macro takes a condition and a list of object files as parameters. If the condition is true (in this case: if the BAR feature is enabled in menuconfig) the object files (in this case: bar.o) will always be compiled. The opposite goes as well: if the condition is not true, bar.o will never be compiled. compile_only_if_not does the opposite: compile if the condition is false, not compile if the condition is true.

This can also be used to select or stub out an implementation, as such:

Kconfig:

```
config ENABLE_LCD_OUTPUT
    bool "Enable LCD output."
    help
        Select this if your board has a LCD.

config ENABLE_LCD_CONSOLE
    bool "Output console text to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output debugging output to the lcd
```

```
config ENABLE_LCD_PLOT
    bool "Output temperature plots to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output temperature plots
```

component.mk:

```
# If LCD is enabled, compile interface to it, otherwise compile dummy interface
$(call compile_only_if,$(CONFIG_ENABLE_LCD_OUTPUT),lcd-real.o lcd-spi.o)
$(call compile_only_if_not,$(CONFIG_ENABLE_LCD_OUTPUT),lcd-dummy.o)

#We need font if either console or plot is enabled
$(call compile_only_if,$(or $(CONFIG_ENABLE_LCD_CONSOLE), $(CONFIG_ENABLE_LCD_PLOT)),  
    font.o)
```

Note the use of the Make ‘or’ function to include the font file. Other substitution functions, like ‘and’ and ‘if’ will also work here. Variables that do not come from menuconfig can also be used: ESP-IDF uses the default Make policy of judging a variable which is empty or contains only whitespace to be false while a variable with any non-whitespace in it is true.

(Note: Older versions of this document advised conditionally adding object file names to COMPONENT_OBJS. While this still is possible, this will only work when all object files for a component are named explicitly, and will not clean up deselected object files in a make clean pass.)

Source Code Generation

Some components will have a situation where a source file isn’t supplied with the component itself but has to be generated from another file. Say our component has a header file that consists of the converted binary data of a BMP file, converted using a hypothetical tool called bmp2h. The header file is then included in as C source file called graphics_lib.c:

```
COMPONENT_EXTRA_CLEAN := logo.h

graphics_lib.o: logo.h

logo.h: $(COMPONENT_PATH)/logo.bmp
    bmp2h -i $^ -o $@
```

In this example, graphics_lib.o and logo.h will be generated in the current directory (the build directory) while logo.bmp comes with the component and resides under the component path. Because logo.h is a generated file, it needs to be cleaned when make clean is called which why it is added to the COMPONENT_EXTRA_CLEAN variable.

Cosmetic Improvements

Because logo.h is a generated file, it needs to be cleaned when make clean is called which why it is added to the COMPONENT_EXTRA_CLEAN variable.

Adding logo.h to the graphics_lib.o dependencies causes it to be generated before graphics_lib.c is compiled.

If a source file in another component included logo.h, then this component’s name would have to be added to the other component’s COMPONENT_DEPENDS list to ensure that the components were built in-order.

Embedding Binary Data

Sometimes you have a file with some binary or text data that you'd like to make available to your component - but you don't want to reformat the file as C source.

You can set a variable COMPONENT_EMBED_FILES in component.mk, giving the names of the files to embed in this way:

```
COMPONENT_EMBED_FILES := server_root_cert.der
```

Or if the file is a string, you can use the variable COMPONENT_EMBED_TXTFILES. This will embed the contents of the text file as a null-terminated string:

```
COMPONENT_EMBED_TXTFILES := server_root_cert.pem
```

The file's contents will be added to the .rodata section in flash, and are available via symbol names as follows:

```
extern const uint8_t server_root_cert_pem_start[] asm("_binary_server_root_cert_pem_"
    ↪start");
extern const uint8_t server_root_cert_pem_end[]     asm("_binary_server_root_cert_pem_"
    ↪end");
```

The names are generated from the full name of the file, as given in COMPONENT_EMBED_FILES. Characters /, ., etc. are replaced with underscores. The _binary prefix in the symbol name is added by objcopy and is the same for both text and binary files.

For an example of using this technique, see [protocols/https_request](#) - the certificate file contents are loaded from the text .pem file at compile time.

Fully Overriding The Component Makefile

Obviously, there are cases where all these recipes are insufficient for a certain component, for example when the component is basically a wrapper around another third-party component not originally intended to be compiled under this build system. In that case, it's possible to forego the esp-idf build system entirely by setting COMPONENT_OWNBUILDTARGET and possibly COMPONENT_OWNCLEANTARGET and defining your own targets named `build` and `clean` in component .mk target. The build target can do anything as long as it creates \$(COMPONENT_LIBRARY) for the project make process to link into the app binary.

(Actually, even this is not strictly necessary - if the COMPONENT_ADD_LDFLAGS variable is overridden then the component can instruct the linker to link other binaries instead.)

Custom sdkconfig defaults

For example projects or other projects where you don't want to specify a full sdkconfig configuration, but you do want to override some key values from the esp-idf defaults, it is possible to create a file `sdkconfig.defaults` in the project directory. This file will be used when running `make defconfig`, or creating a new config from scratch.

To override the name of this file, set the `SDKCONFIG_DEFAULTS` environment variable.

Save flash arguments

There're some scenarios that we want to flash the target board without IDF. For this case we want to save the built binaries, esptool.py and esptool write_flash arguments. It's simple to write a script to save binaries and esptool.py. For flash arguments, we can add the following code to application project makefile:

```
print_flash_cmd:
    echo ${ESPTOOL_WRITE_FLASH_OPTIONS} ${ESPTOOL_ALL_FLASH_ARGS} | sed -e 's:'$PWD'/
↪build/':g'
```

the original ESPTOOL_ALL_FLASH_ARGS are absolute file name. Usually we want to save relative file name so we can move the bin folder to somewhere else. For this case we can use sed to convert to relative file name, like what we did in the example above.

When running make print_flash_cmd, it will print the flash arguments:

```
--flash_mode dio --flash_freq 40m --flash_size detect 0x1000 bootloader/bootloader.
↪bin 0x10000 example_app.bin 0x8000 partition_table_unit_test_app.bin
```

Then use flash arguments as the arguemnts for esptool write_flash arguments:

```
python esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 921600 --before default_
↪reset --after hard_reset write_flash -z --flash_mode dio --flash_freq 40m --flash_
↪size detect 0x1000 bootloader/bootloader.bin 0x10000 example_app.bin 0x8000_
↪partition_table_unit_test_app.bin
```

4.2.3 Building the Bootloader

The bootloader is built by default as part of “make all”, or can be built standalone via “make bootloader-clean”. There is also “make bootloader-list-components” to see the components included in the bootloader build.

The component in IDF components/bootloader is special, as the second stage bootloader is a separate .ELF and .BIN file to the main project. However it shares its configuration and build directory with the main project.

This is accomplished by adding a subproject under components/bootloader/subproject. This subproject has its own Makefile, but it expects to be called from the project’s own Makefile via some glue in the components/bootloader/Makefile.projectbuild file. See these files for more details.

4.3 Deep Sleep Wake Stubs

ESP32 supports running a “deep sleep wake stub” when coming out of deep sleep. This function runs immediately as soon as the chip wakes up - before any normal initialisation, bootloader, or ESP-IDF code has run. After the wake stub runs, the SoC can go back to sleep or continue to start ESP-IDF normally.

Deep sleep wake stub code is loaded into “RTC Fast Memory” and any data which it uses must also be loaded into RTC memory. RTC memory regions hold their contents during deep sleep.

4.3.1 Rules for Wake Stubs

Wake stub code must be carefully written:

- As the SoC has freshly woken from sleep, most of the peripherals are in reset states. The SPI flash is unmapped.
- The wake stub code can only call functions implemented in ROM or loaded into RTC Fast Memory (see below.)
- The wake stub code can only access data loaded in RTC memory. All other RAM will be unintialised and have random contents. The wake stub can use other RAM for temporary storage, but the contents will be overwritten when the SoC goes back to sleep or starts ESP-IDF.
- RTC memory must include any read-only data (.rodata) used by the stub.

- Data in RTC memory is initialised whenever the SoC restarts, except when waking from deep sleep. When waking from deep sleep, the values which were present before going to sleep are kept.
- Wake stub code is a part of the main esp-idf app. During normal running of esp-idf, functions can call the wake stub functions or access RTC memory. It is as if these were regular parts of the app.

4.3.2 Implementing A Stub

The wake stub in esp-idf is called `esp_wake_deep_sleep()`. This function runs whenever the SoC wakes from deep sleep. There is a default version of this function provided in esp-idf, but the default function is weak-linked so if your app contains a function named `esp_wake_deep_sleep()` then this will override the default.

If supplying a custom wake stub, the first thing it does should be to call `esp_default_wake_deep_sleep()`.

It is not necessary to implement `esp_wake_deep_sleep()` in your app in order to use deep sleep. It is only necessary if you want to have special behaviour immediately on wake.

If you want to swap between different deep sleep stubs at runtime, it is also possible to do this by calling the `esp_set_deep_sleep_wake_stub()` function. This is not necessary if you only use the default `esp_wake_deep_sleep()` function.

All of these functions are declared in the `esp_deepsleep.h` header under components/esp32.

4.3.3 Loading Code Into RTC Memory

Wake stub code must be resident in RTC Fast Memory. This can be done in one of two ways.

The first way is to use the `RTC_IRAM_ATTR` attribute to place a function into RTC memory:

```
void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    // Add additional functionality here
}
```

The second way is to place the function into any source file whose name starts with `rtc_wake_stub`. Files names `rtc_wake_stub*` have their contents automatically put into RTC memory by the linker.

The first way is simpler for very short and simple code, or for source files where you want to mix “normal” and “RTC” code. The second way is simpler when you want to write longer pieces of code for RTC memory.

4.3.4 Loading Data Into RTC Memory

Data used by stub code must be resident in RTC Slow Memory. This memory is also used by the ULP.

Specifying this data can be done in one of two ways:

The first way is to use the `RTC_DATA_ATTR` and `RTC_RODATA_ATTR` to specify any data (writeable or read-only, 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.

4.4 ESP32 Core Dump

4.4.1 Overview

ESP-IDF provides support to generate core dumps on unrecoverable software errors. This useful technique allows post-mortem analysis of software state at the moment of failure. Upon the crash system enters panic state, prints some information and halts or reboots depending configuration. User can choose to generate core dump in order to analyse the reason of failure on PC later on. Core dump contains snapshots of all tasks in the system at the moment of failure. Snapshots include tasks control blocks (TCB) and stacks. So it is possible to find out what task, at what instruction (line of code) and what callstack of that task lead to the crash. ESP-IDF provides special script *espcoredump.py* to help users to retrieve and analyse core dumps. This tool provides two commands for core dumps analysis:

- info_corefile - prints crashed task's registers, callstack, list of available tasks in the system, memory regions and contents of memory stored in core dump (TCBs and stacks)
- dbg_corefile - creates core dump ELF file and runs GDB debug session with this file. User can examine memory, variables and tasks states manually. Note that since not all memory is saved in core dump only values of variables allocated on stack will be meaningful

4.4.2 Configuration

There are a number of core dump related configuration options which user can choose in configuration menu of the application (*make menuconfig*).

1. Core dump data destination (*Components -> ESP32-specific config -> Core dump destination*):
 - Disable core dump generation
 - Save core dump to flash
 - Print core dump to UART
2. Logging level of core dump module (*Components -> ESP32-specific config -> Core dump module logging level*). Value is a number from 0 (no output) to 5 (most verbose).
3. Delay before core dump will be printed to UART (*Components -> ESP32-specific config -> Core dump print to UART delay*). Value is in ms.

4.4.3 Save core dump to flash

When this option is selected core dumps are saved to special partition on flash. When using default partition table files which are provided with ESP-IDF it automatically allocates necessary space on flash, But if user wants to use its own layout file together with core dump feature it should define separate partition for core dump as it is shown below:

```
# Name, Type, SubType, Offset, Size
# Note: if you change the phy_init or app partition offset, make sure to change the
# offset in Kconfig.projbuild
nvs, data, nvs, 0x9000, 0x6000
phy_init, data, phy, 0xf000, 0x1000
factory, app, factory, 0x10000, 1M
coredump, data, coredump,, 64K
```

There are no special requirements for partition name. It can be chosen according to the user application needs, but partition type should be ‘data’ and sub-type should be ‘coredump’. Also when choosing partition size note that core dump data structure introduces constant overhead of 20 bytes and per-task overhead of 12 bytes. This overhead does not include size of TCB and stack for every task. So partition size should be at least 20 + max tasks number x (12 + TCB size + max task stack size) bytes.

The example of generic command to analyze core dump from flash is: `espcoredump.py -p </path/to/serial/port> info_corefile </path/to/program/elf/file>` or `espcoredump.py -p </path/to/serial/port> dbg_corefile </path/to/program/elf/file>`

4.4.4 Print core dump to UART

When this option is selected base64-encoded core dumps are printed on UART upon system panic. In this case user should save core dump text body to some file manually and then run the following command: `espcoredump.py info_corefile -t b64 -c </path/to/saved/base64/text> </path/to/program/elf/file>` or `espcoredump.py dbg_corefile -t b64 -c </path/to/saved/base64/text> </path/to/program/elf/file>`

Base64-encoded body of core dump will be between the following header and footer:

```
===== CORE DUMP START =====
<body of base64-encoded core dump, save it to file on disk>
===== CORE DUMP END =====
```

4.4.5 Running ‘espcoredump.py’

Generic command syntax:

`espcoredump.py [options] command [args]`

Script Options

- `-chip,-c {auto,esp32}`. Target chip type. Supported values are `auto` and `esp32`.
- `-port,-p PORT`. Serial port device.
- `-baud,-b BAUD`. Serial port baud rate used when flashing/reading.

Commands

- `info_corefile`. Retrieve core dump and print useful info.
- `dbg_corefile`. Retrieve core dump and start GDB session with it.

Command Arguments

- `-gdb,-g` GDB. Path to gdb to use for data retrieval.
- `-core,-c` CORE. Path to core dump file to use (if skipped core dump will be read from flash).
- `-core-format,-t` CORE_FORMAT. Specifies that file passed with “`-c`” is an ELF (“elf”), dumped raw binary (“raw”) or base64-encoded (“b64”) format.
- `-off,-o` OFF. Offset of coredump partition in flash (type “make partition_table” to see it).
- `-save-core,-s` SAVE_CORE. Save core to file. Otherwise temporary core file will be deleted. Ignored with “`-c`”.
- `-print-mem,-m` Print memory dump. Used only with “info_corefile”.

4.5 Flash Encryption

Flash Encryption is a feature for encrypting the contents of the ESP32’s attached SPI flash. When flash encryption is enabled, physical readout of the SPI flash is not sufficient to recover most flash contents.

Flash Encryption is separate from the [Secure Boot](#) feature, and you can use flash encryption without enabling secure boot. However we recommend using both features together for a secure environment.

IMPORTANT: Enabling flash encryption limits your options for further updates of your ESP32. Make sure to read this document (including :ref:`flash-encryption-limitations`) and understand the implications of enabling flash encryption.

4.5.1 Background

- The contents of the flash are encrypted using AES with a 256 bit key. The flash encryption key is stored in efuse internal to the chip, and is (by default) protected from software access.
- Flash access is transparent via the flash cache mapping feature of ESP32 - any flash regions which are mapped to the address space will be transparently decrypted when read.
- Encryption is applied by flashing the ESP32 with plaintext data, and (if encryption is enabled) the bootloader encrypts the data in place on first boot.
- Not all of the flash is encrypted. The following kinds of flash data are encrypted:
 - Bootloader
 - Secure boot bootloader digest (if secure boot is enabled)
 - Partition Table
 - All “app” type partitions
 - Any partition marked with the “encrypt” flag in the partition table

It may be desirable for some data partitions to remain unencrypted for ease of access, or to use flash-friendly update algorithms that are ineffective if the data is encrypted. “NVS” partitions for non-volatile storage cannot be encrypted.

- The flash encryption key is stored in efuse key block 1, internal to the ESP32 chip. By default, this key is read-and write-protected so software cannot access it or change it.
- The *flash encryption algorithm* is AES-256, where the key is “tweaked” with the offset address of each 32 byte block of flash. This means every 32 byte block (two consecutive 16 byte AES blocks) is encrypted with a unique key derived from the flash encryption key.

- Although software running on the chip can transparently decrypt flash contents, by default it is made impossible for the UART bootloader to decrypt (or encrypt) data when flash encryption is enabled.
- If flash encryption may be enabled, the programmer must take certain precautions when writing code that *uses encrypted flash*.

4.5.2 Flash Encryption Initialisation

This is the default (and recommended) flash encryption initialisation process. It is possible to customise this process for development or other purposes, see [Flash Encryption Advanced Features](#) for details.

IMPORTANT: Once flash encryption is enabled on first boot, the hardware allows a maximum of 3 subsequent flash updates via serial re-flashing. A special procedure (documented in [Serial Flashing](#)) must be followed to perform these updates.

- If secure boot is enabled, no physical re-flashes are possible.
- OTA updates can be used to update flash content without counting towards this limit.
- When enabling flash encryption in development, use a *pregenerated flash encryption key* to allow physically re-flashing an unlimited number of times with pre-encrypted data.**

Process to enable flash encryption:

- The bootloader must be compiled with flash encryption support enabled. In `make menuconfig`, navigate to “Security Features” and select “Yes” for “Enable flash encryption on boot”.
- If enabling Secure Boot at the same time, it is best to simultaneously select those options now. Read the [Secure Boot](#) documentation first.
- Build and flash the bootloader, partition table and factory app image as normal. These partitions are initially written to the flash unencrypted.
- On first boot, the bootloader sees *FLASH_CRYPT_CNT efuse* is set to 0 (factory default) so it generates a flash encryption key using the hardware random number generator. This key is stored in efuse. The key is read and write protected against further software access.
- All of the encrypted partitions are then encrypted in-place by the bootloader. Encrypting in-place can take some time (up to a minute for large partitions.)

IMPORTANT: Do not interrupt power to the ESP32 while the first boot encryption pass is running. If power is interrupted, the flash contents will be corrupted and require flashing with unencrypted data again. A reflash like this will not count towards the flashing limit.

- Once flashing is complete, efuses are blown (by default) to disable encrypted flash access while the UART bootloader is running. See [Enabling UART Bootloader Encryption/Decryption](#) for advanced details.
- The *FLASH_CRYPT_CONFIG efuse* is also burned to the maximum value (0xF) to maximise the number of key bits which are tweaked in the flash algorithm. See [Setting FLASH_CRYPT_CONFIG](#) for advanced details.
- Finally, the *FLASH_CRYPT_CNT efuse* is burned with the initial value 1. It is this efuse which activates the transparent flash encryption layer, and limits the number of subsequent reflashes. See the [Updating Encrypted Flash](#) section for details about *FLASH_CRYPT_CNT efuse*.
- The bootloader resets itself to reboot from the newly encrypted flash.

4.5.3 Using Encrypted Flash

ESP32 app code can check if flash encryption is currently enabled by calling `esp_flash_encryption_enabled()`.

Once flash encryption is enabled, some care needs to be taken when accessing flash contents from code.

Scope of Flash Encryption

Whenever the `FLASH_CRYPT_CNT` efuse is set to a value with an odd number of bits set, all flash content which is accessed via the MMU's flash cache is transparently decrypted. This includes:

- Executable application code in flash (IROM).
- All read-only data stored in flash (DROM).
- Any data accessed via `esp_spi_flash_mmap()`.
- The software bootloader image when it is read by the ROM bootloader.

IMPORTANT: The MMU flash cache unconditionally decrypts all data. Data which is stored unencrypted in the flash will be “transparently decrypted” via the flash cache and appear to software like random garbage.

Reading Encrypted Flash

To read data without using a flash cache MMU mapping, we recommend using the partition read function `esp_partition_read()`. When using this function, data will only be decrypted when it is read from an encrypted partition. Other partitions will be read unencrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

Data which is read via other SPI read APIs are not decrypted:

- Data read via `esp_spi_flash_read()` is not decrypted
- Data read via ROM function `SPIRead()` is not decrypted (this function is not supported in esp-idf apps).
- Data stored using the Non-Volatile Storage (NVS) API is always stored and read decrypted.

Writing Encrypted Flash

Where possible, we recommend using the partition write function `esp_partition_write`. When using this function, data will only be encrypted when writing to encrypted partitions. Data will be written to other partitions unencrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

The `esp_spi_flash_write` function will write data when the `write_encrypted` parameter is set to true. Otherwise, data will be written unencrypted.

The ROM function `esp_rom_spiflash_write_encrypted` will write encrypted data to flash, the ROM function `SPIWrite` will write unencrypted to flash. (these function are not supported in esp-idf apps).

The minimum write size for unencrypted data is 4 bytes (and the alignment is 4 bytes). Because data is encrypted in blocks, the minimum write size for encrypted data is 16 bytes (and the alignment is 16 bytes.)

4.5.4 Updating Encrypted Flash

OTA Updates

OTA updates to encrypted partitions will automatically write encrypted, as long as the `esp_partition_write` function is used.

Serial Flashing

Provided secure boot is not used, the `FLASH_CRYPT_CNT` efuse allows the flash to be updated with new plaintext data via serial flashing (or other physical methods), up to 3 additional times.

The process involves flashing plaintext data, and then bumping the value of `FLASH_CRYPT_CNT` efuse which causes the bootloader to re-encrypt this data.

Limited Updates

Only 4 serial flash update cycles of this kind are possible, including the initial encrypted flash.

After the fourth time encryption is disabled, `FLASH_CRYPT_CNT` efuse has the maximum value `0xFF` and encryption is permanently disabled.

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.

Disabling Serial Updates

To prevent further plaintext updates via serial, use `espefuse.py` to write protect the `FLASH_CRYPT_CNT` efuse after flash encryption has been enabled (ie after first boot is complete):

```
espefuse.py --port PORT write_protect_efuse FLASH_CRYPT_CNT
```

This prevents any further modifications to disable or re-enable flash encryption.

Reflashing via Pregenerated Flash Encryption Key

It is possible to pregenerate a flash encryption key on the host computer and burn it into the ESP32's efuse key block. This allows data to be pre-encrypted on the host and flashed to the ESP32 without needing a plaintext flash update.

This is useful for development, because it removes the 4 time reflashing limit. It also allows reflashing with secure boot enabled, because the bootloader doesn't need to be reflashed each time.

IMPORTANT This method is intended to assist with development only, not for production devices. If pre-generating flash encryption for production, ensure the keys are generated from a high quality random number source and do not share the same flash encryption key across multiple devices.

Pregenerating a Flash Encryption Key

Flash encryption keys are 32 bytes of random data. You can generate a random key with espsecure.py:

```
espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin
```

(The randomness of this data is only as good as the OS and it's Python installation's random data source.)

Alternatively, if you're using *secure boot* and have a secure boot signing key then you can generate a deterministic SHA-256 digest of the secure boot private signing key and use this as the flash encryption key:

```
espsecure.py digest_private_key --keyfile secure_boot_signing_key.pem my_flash_
˓→encryption_key.bin
```

(The same 32 bytes is used as the secure boot digest key if you enable *reflashable mode* for secure boot.)

Generating the flash encryption key from the secure boot signing key in this way means that you only need to store one key file. However this method is **not at all suitable** for production devices.

Burning Flash Encryption Key

Once you have generated a flash encryption key, you need to burn it to the ESP32's efuse key block. **This must be done before first encrypted boot**, otherwise the ESP32 will generate a random key that software can't access or modify.

To burn a key to the device (one time only):

```
espefuse.py --port PORT burn_key flash_encryption my_flash_encryption_key.bin
```

First Flash with pregenerated key

After flashing the key, follow the same steps as for default *Flash Encryption Initialisation* and flash a plaintext image for the first boot. The bootloader will enable flash encryption using the pre-burned key and encrypt all partitions.

Reflashing with pregenerated key

After encryption is enabled on first boot, reflashing an encrypted image requires an additional manual step. This is where we pre-encrypt the data that we wish to update in flash.

Suppose that this is the normal command used to flash plaintext data:

```
esptool.py --port /dev/ttyUSB0 --baud 115200 write_flash 0x10000 build/my-app.bin
```

Binary app image build/my-app.bin is written to offset 0x10000. This file name and offset need to be used to encrypt the data, as follows:

```
espsecure.py encrypt_flash_data --keyfile my_flash_encryption_key.bin --address ↴0x10000 -o build/my-app-encrypted.bin build/my-app.bin
```

This example command will encrypts my-app.bin using the supplied key, and produce an encrypted file my-app-encrypted.bin. Be sure that the address argument matches the address where you plan to flash the binary.

Then, flash the encrypted binary with esptool.py:

```
esptool.py --port /dev/ttyUSB0 --baud 115200 write_flash 0x10000 build/my-app- ↴encrypted.bin
```

No further steps or efuse manipulation is necessary, because the data is already encrypted when we flash it.

4.5.5 Disabling Flash Encryption

If you've accidentally enabled flash encryption for some reason, the next flash of plaintext data will soft-brick the ESP32 (the device will reboot continuously, printing the error `flash read err, 1000`).

You can disable flash encryption again by writing `FLASH_CRYPT_CNT` efuse:

- First, run `make menuconfig` and uncheck “Enable flash encryption boot” under “Security Features”.
- Exit menuconfig and save the new configuration.
- Run `make menuconfig` again and double-check you really disabled this option! *If this option is left enabled, the bootloader will immediately re-enable encryption when it boots.*
- Run `make flash` to build and flash a new bootloader and app, without flash encryption enabled.
- **Run `espefuse.py` (in `components/esptool_py/esptool`) to disable the `FLASH_CRYPT_CNT` efuse:**
`espefuse.py burn_efuse FLASH_CRYPT_CNT`

Reset the ESP32 and flash encryption should be disabled, the bootloader will boot as normal.

4.5.6 Limitations of Flash Encryption

Flash Encryption prevents plaintext readout of the encrypted flash, to protect firmware against unauthorised readout and modification. It is important to understand the limitations of the flash encryption system:

- Flash encryption is only as strong as the key. For this reason, we recommend keys are generated on the device during first boot (default behaviour). If generating keys off-device (see [Reflashing via Pregenerated Flash Encryption Key](#)), ensure proper procedure is followed.
- Not all data is stored encrypted. If storing data on flash, check if the method you are using (library, API, etc.) supports flash encryption.

- Flash encryption does not prevent an attacker from understanding the high-level layout of the flash. This is because the same AES key is used for every pair of adjacent 16 byte AES blocks. When these adjacent 16 byte blocks contain identical content (such as empty or padding areas), these blocks will encrypt to produce matching pairs of encrypted blocks. This may allow an attacker to make high-level comparisons between encrypted devices (ie to tell if two devices are probably running the same firmware version).
- For the same reason, an attacker can always tell when a pair of adjacent 16 byte blocks (32 byte aligned) contain identical content. Keep this in mind if storing sensitive data on the flash, design your flash storage so this doesn't happen (using a counter byte or some other non-identical value every 16 bytes is sufficient).
- Flash encryption alone may not prevent an attacker from modifying the firmware of the device. To prevent unauthorised firmware from running on the device, use flash encryption in combination with [Secure Boot](#).

4.5.7 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.

4.5.8 Technical Details

The following sections provide some reference information about the operation of flash encryption.

`FLASH_CRYPT_CNT` efuse

`FLASH_CRYPT_CNT` is an 8-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 (0,2,4,6,8) 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.

- After all 8 bits are set (efuse value 0xFF): Transparent reading of encrypted flash is disabled, any encrypted data is permanently inaccessible. Bootloader will normally detect this condition and halt. To avoid use of this state to load unauthorised code, secure boot must be used or [`FLASH_CRYPT_CNT`](#) efuse must be write-protected.

Flash Encryption Algorithm

- AES-256 operates on 16 byte blocks of data. The flash encryption engine encrypts and decrypts data in 32 byte blocks, two AES blocks in series.
- AES algorithm is used inverted in flash encryption, so the flash encryption “encrypt” operation is AES decrypt and the “decrypt” operation is AES encrypt. This is for performance reasons and does not alter the effectiveness of the algorithm.
- The main flash encryption key is stored in efuse (BLOCK1) and by default is protected from further writes or software readout.
- Each 32 byte block (two adjacent 16 byte AES blocks) is encrypted with a unique key. The key is derived from the main flash encryption key in efuse, XORed with the offset of this block in the flash (a “key tweak”).
- The specific tweak depends on the setting of `FLASH_CRYPT_CONFIG` efuse. This is a 4 bit efuse, where each bit enables XORing of a particular range of the key bits:
 - Bit 1, bits 0-66 of the key are XORed.
 - Bit 2, bits 67-131 of the key are XORed.
 - Bit 3, bits 132-194 of the key are XORed.
 - Bit 4, bits 195-256 of the key are XORed.

It is recommended that `FLASH_CRYPT_CONFIG` is always left to set the default value `0xF`, so that all key bits are XORed with the block offset. See [`Setting FLASH_CRYPT_CONFIG`](#) for details.

- The high 19 bits of the block offset (bit 5 to bit 23) are XORed with the main flash encryption key. This range is chosen for two reasons: the maximum flash size is 16MB (24 bits), and each block is 32 bytes so the least significant 5 bits are always zero.
- There is a particular mapping from each of the 19 block offset bits to the 256 bits of the flash encryption key, to determine which bit is XORed with which. See the variable `_FLASH_ENCRYPTION_TWEAK_PATTERN` in the `espsecure.py` source code for the complete mapping.
- To see the full flash encryption algorithm implemented in Python, refer to the `_flash_encryption_operation()` function in the `espsecure.py` source code.

4.6 High-Level Interrupts

The Xtensa architecture has support for 32 interrupts, divided over 8 levels, plus an assortment of exceptions. On the ESP32, the interrupt mux allows most interrupt sources to be routed to these interrupts using the [*interrupt allocator*](#). Normally, interrupts will be written in C, but ESP-IDF allows high-level interrupts to be written in assembly as well, allowing for very low interrupt latencies.

4.6.1 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 iuninterrupt is implemented there.

4.6.2 Notes

- Do not call C code from a high-level interrupt; because these interrupts still run in critical sections, this can cause crashes. (The panic handler interrupt does call normal C code, but this is OK because there is no intention of returning to the normal code flow afterwards.)
- Make sure your assembly code gets linked in. If the interrupt handler symbol is the only symbol the rest of the code uses from this file, the linker will take the default ISR instead and not link the assembly file into the final project. To get around this, in the assembly file, define a symbol, like this:

```
.global ld_include_my_isr_file
ld_include_my_isr_file:
```

(The symbol is called `ld_include_my_isr_file` here but can have any arbitrary name not defined anywhere else.) Then, in the component.mk, add this file as an unresolved symbol to the ld command line arguments:

```
COMPONENT_ADD_LDFLAGS := -u ld_include_my_isr_file
```

This should cause the linker to always include a file defining `ld_include_my_isr_file`, causing the ISR to always be linked in.

- High-level interrupts can be routed and handled using `esp_intr_alloc` and associated functions. The handler and handler arguments to `esp_intr_alloc` must be NULL, however.
- In theory, medium priority interrupts could also be handled in this way. For now, ESP-IDF does not support this.

4.7 JTAG Debugging

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB. The document is structured as follows:

Introduction Introduction to the purpose of this guide.

How it Works? Description how ESP32, JTAG interface, OpenOCD and GDB are interconnected and working together to enable debugging of ESP32.

Selecting JTAG Adapter What are the criteria and options to select JTAG adapter hardware.

Setup of OpenOCD Procedure to install OpenOCD using prebuilt software packages for *Windows*, *Linux* and *MacOS* operating systems.

Configuring ESP32 Target Configuration of OpenOCD software and set up JTAG adapter hardware that will make together a debugging target.

Launching Debugger Steps to start up a debug session with GDB from *Eclipse* and from *Command Line*.

Debugging Examples If you are not familiar with GDB, check this section for debugging examples provided from *Eclipse* as well as from *Command Line*.

Building OpenOCD from Sources Procedure to build OpenOCD from sources for *Windows*, *Linux* and *MacOS* operating systems.

Tips and Quirks This section provides collection of tips and quirks related JTAG debugging of ESP32 with OpenOCD and GDB.

4.7.1 Introduction

The ESP32 has two powerful Xtensa cores, allowing for a great deal of variety of program architectures. The FreeRTOS OS that comes with ESP-IDF is capable of multi-core preemptive multithreading, allowing for an intuitive way of writing software.

The downside of the ease of programming is that debugging without the right tools is harder: figuring out a bug that is caused by two threads, running even simultaneously on two different CPU cores, can take a long time when all you have are printf statements. A better and in many cases quicker way to debug such problems is by using a debugger, connected to the processors over a debug port.

Espressif has ported OpenOCD to support the ESP32 processor and the multicore FreeRTOS, which will be the foundation of most ESP32 apps, and has written some tools to help with features OpenOCD does not support natively.

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB under Linux, Windows and MacOS. Except for OS specific installation procedures, the s/w user interface and use procedures are the same across all supported operating systems.

Note: Screenshots presented in this document have been made for Eclipse Neon 3 running on Ubuntu 16.04 LTE. There may be some small differences in what a particular user interface looks like, depending on whether you are using Windows, MacOS or Linux and / or a different release of Eclipse.

4.7.2 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.

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.

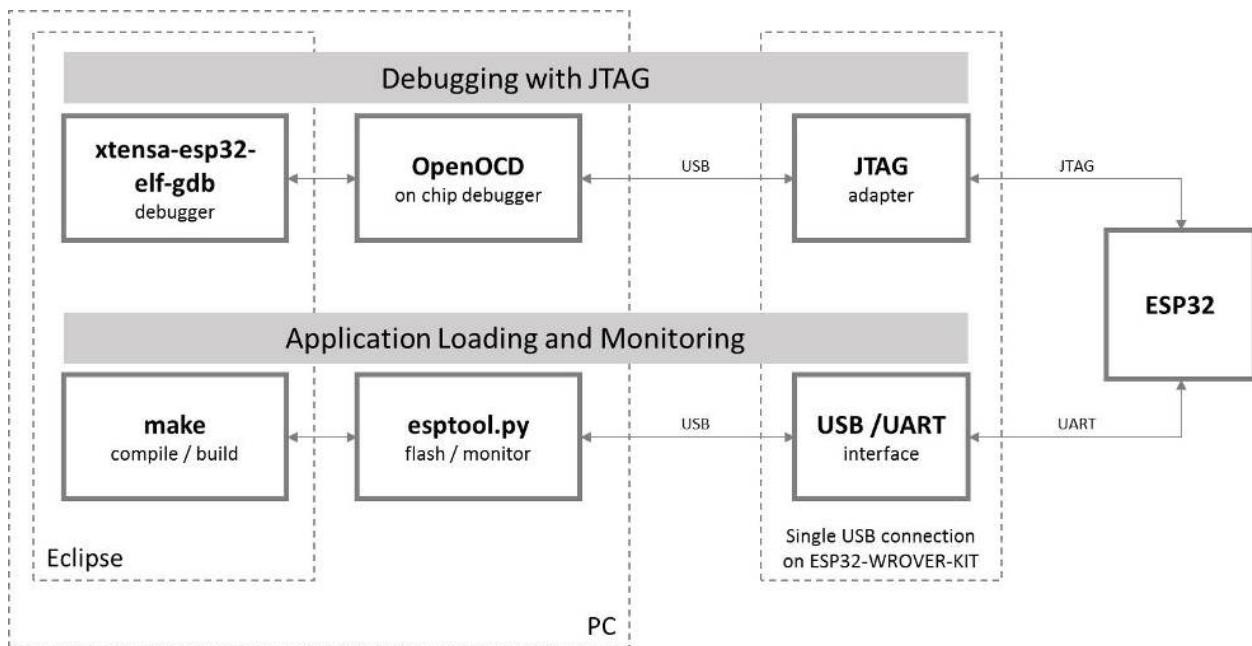


Fig. 4.1: JTAG debugging - overview diagram

Debugging using JTAG and application loading / monitoring is integrated under the [Eclipse](#) environment, to provide quick and easy transition from writing, compiling and loading the code to debugging, back to writing the code, and so on. All the software is available for Windows, Linux and MacOS platforms.

If the [ESP32 WROVER KIT](#) is used, then connection from PC to ESP32 is done effectively with a single USB cable thanks to FT2232H chip installed on WROVER, which provides two USB channels, one for JTAG and the second for UART connection.

Depending on user preferences, both *debugger* and *make* can be operated directly from terminal / command line, instead from Eclipse.

4.7.3 Selecting JTAG Adapter

The quickest and most convenient way to start with JTAG debugging is by using [ESP32 WROVER KIT](#). Each version of this development board has JTAG interface already build in. No need for an external JTAG adapter and extra wiring / cable to connect JTAG to ESP32. WROVER KIT is using FT2232H JTAG interface operating at 20 MHz clock speed, which is difficult to achieve with an external adapter.

If you decide to use separate JTAG adapter, look for one that is compatible with both the voltage levels on the ESP32 as well as with the OpenOCD software. The JTAG port on the ESP32 is an industry-standard JTAG port which lacks (and does not need) the TRST pin. The JTAG I/O pins all are powered from the VDD_3P3_RTC pin (which normally would be powered by a 3.3V rail) so the JTAG adapter needs to be able to work with JTAG pins in that voltage range.

On the software side, OpenOCD supports a fair amount of JTAG adapters. See <http://openocd.org/doc/html/Debug-Adapter-Hardware.html> for an (unfortunately slightly incomplete) list of the adapters OpenOCD works with. This page lists SWD-compatible adapters as well; take note that the ESP32 does not support SWD. JTAG adapters that are hardcoded to a specific product line, e.g. STM32 debugging adapters, will not work.

The minimal signalling to get a working JTAG connection are TDI, TDO, TCK, TMS and GND. Some JTAG debuggers also need a connection from the ESP32 power line to a line called e.g. Vtar to set the working voltage. SRST can optionally be connected to the CH_PD of the ESP32, although for now, support in OpenOCD for that line is pretty minimal.

4.7.4 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.

Setup OpenOCD for Windows

Setup OpenOCD

OpenOCD for Windows / MSYS2 is available for download from Espressif website:

<https://dl.espressif.com/dl/openocd-esp32-win32-a859564.zip>

Download this file and extract `openocd-esp32` folder inside to `~/esp/` directory.

Next Steps

To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

Related Documents

Building OpenOCD from Sources for Windows

The following instructions are alternative to downloading binary OpenOCD from Espressif website. To quickly setup the binary OpenOCD, instead of compiling it yourself, backup and proceed to section [Setup OpenOCD for Windows](#).

Download Sources of OpenOCD

The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp
git clone -recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Install Dependencies

Install packages that are required to compile OpenOCD:

Note: Install the following packages one by one, check if installation was successful and then proceed to the next package. Resolve reported problems before moving to the next step.

```
pacman -S libtool
pacman -S autoconf
pacman -S automake
pacman -S texinfo
```

```
pacman -S mingw-w64-i686-libusb-compat-git
pacman -S pkg-config
```

Note: Installation of `pkg-config` is breaking operation of esp-idf toolchain. After building of OpenOCD it should be uninstalled. It be covered at the end of this instruction. To build OpenOCD again, you will need to run `pacman -S pkg-config` once more. This issue does not concern other packages installed in this step (before `pkg-config`).

Build OpenOCD

Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
 - If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
 - If the `./configure` is successfully run, information of enabled JTAG will be printed under OpenOCD configuration summary.
 - If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
 - For details concerning compiling OpenOCD, please refer to `openocd-esp32/README.Windows`.
-

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/esp/openocd-esp32/src/openocd` directory.

Remove `pkg-config`, as discussed during installation of dependencies:

```
pacman -Rs pkg-config
```

Next Steps

To carry on with debugging environment setup, proceed to section *Configuring ESP32 Target*.

Setup OpenOCD for Linux

Setup OpenOCD

OpenOCD for 64-bit Linux is available for download from Espressif website:

<https://dl.espressif.com/dl/openocd-esp32-linux64-a859564.tar.gz>

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

```
cd ~/esp  
tar -xzf ~/Downloads/openocd-esp32-linux64-a859564.tar.gz
```

Next Steps

To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

Related Documents

Building OpenOCD from Sources for Linux

The following instructions are alternative to downloading binary OpenOCD from Espressif website. To quickly setup the binary OpenOCD, instead of compiling it yourself, backup and proceed to section [Setup OpenOCD for Linux](#).

Download Sources of OpenOCD

The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp  
git clone -recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Install Dependencies

Install packages that are required to compile OpenOCD.

Note: Install the following packages one by one, check if installation was successful and then proceed to the next package. Resolve reported problems before moving to the next step.

```
sudo apt-get install make  
sudo apt-get install libtool  
sudo apt-get install pkg-config  
sudo apt-get install autoconf  
sudo apt-get install automake  
sudo apt-get install texinfo  
sudo apt-get install libusb-1.0
```

Note:

- Version of `pkg-config` should be 0.2.3 or above.
- Version of `autoconf` should be 2.6.4 or above.
- Version of `automake` should be 1.9 or above.

- When using USB-Blaster, ASIX Presto, OpenJTAG and FT2232 as adapters, drivers libFTDI and FTD2XX need to be downloaded and installed.
 - When using CMSIS-DAP, HIDAPI is needed.
-

Build OpenOCD

Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `sudo make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
 - If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
 - If the `./configure` is successfully run, information of enabled JTAG will be printed under OpenOCD configuration summary.
 - If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
 - For details concerning compiling OpenOCD, please refer to `openocd-esp32/README`.
-

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/openocd-esp32/bin` directory.

Next Steps

To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

Setup OpenOCD for MacOS

Setup OpenOCD

OpenOCD for MacOS is available for download from Espressif website:

<https://dl.espressif.com/dl/openocd-esp32-macos-a859564.tar.gz>

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

```
cd ~/esp
tar -xzf ~/Downloads/openocd-esp32-macos-a859564.tar.gz
```

Next Steps

To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

Related Documents

Building OpenOCD from Sources for MacOS

The following instructions are alternative to downloading binary OpenOCD from Espressif website. To quickly setup the binary OpenOCD, instead of compiling it yourself, backup and proceed to section [Setup OpenOCD for MacOS](#).

Download Sources of OpenOCD

The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp
git clone -recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Install Dependencies

Install packages that are required to compile OpenOCD using Homebrew:

```
brew install automake libtool libusb wget gcc@4.9
```

Build OpenOCD

Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `sudo make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
- If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
- If the `./configure` is successfully run, information of enabled JTAG will be printed under OpenOCD configuration summary.
- If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.

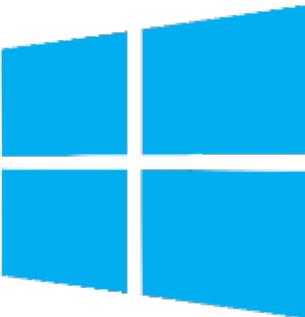
-
- For details concerning compiling OpenOCD, please refer to [openocd-esp32/README.OSX](#).
-

Once make process is successfully completed, the executable of OpenOCD will be saved in `~/esp/openocd-esp32/src/openocd` directory.

Next Steps

To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

Pick up your OS below and follow provided instructions to setup OpenOCD.

		
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](#).

4.7.5 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 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 (vxxxxx)” 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 (vxxxxx)” driver of this device should be replaced with “WinUSB (v6xxxxxx)”. To do so, select “Dual RS232-HS (Interface 0) and reinstall attached driver to the “WinUSB (v6xxxxxx)”, see picture above.

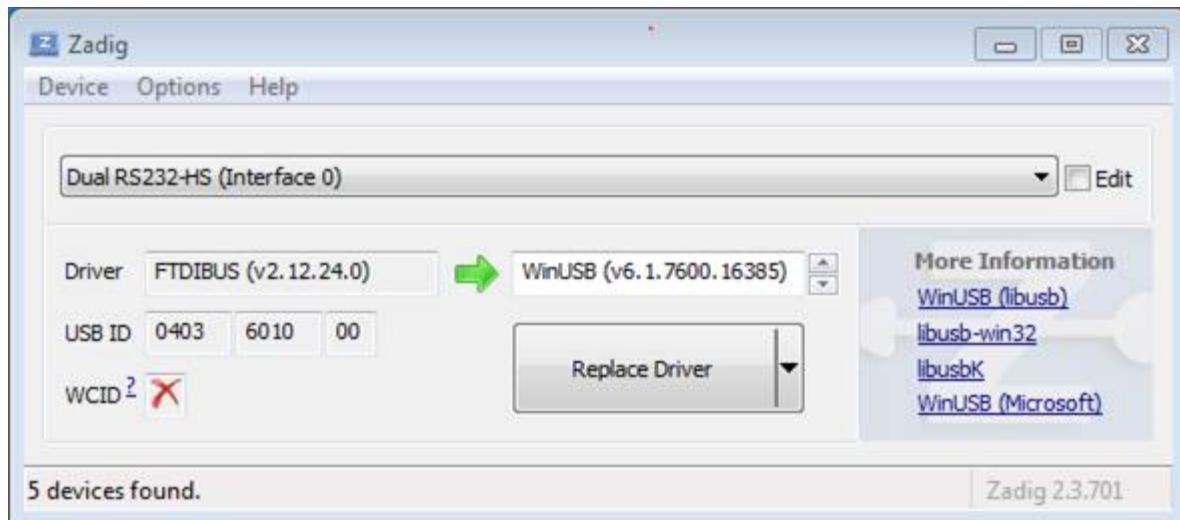


Fig. 4.2: Configuration of JTAG USB driver in Zadig tool

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.
2. Open a terminal, enter `ls -l /dev/ttyUSB*` command and check, if board’s USB ports are recognized by the OS. You are looking for similar result:

```
user-name@computer-name:~/esp$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 Jul 10 19:04 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 1 Jul 10 19:04 /dev/ttyUSB1
```

3. Following section “Permissions delegation” in OpenOCD’s README, set up the access permissions to both USB ports.
4. Log off and login, then cycle the power to the board to make the changes effective. In terminal enter again `ls -l /dev/ttyUSB*` command to verify, if group-owner has changed from `dialout` to `plugdev`:

```
user-name@computer-name:~/esp$ ls -l /dev/ttyUSB*
crw-rw-r-- 1 root plugdev 188, 0 Jul 10 19:07 /dev/ttyUSB0
crw-rw-r-- 1 root plugdev 188, 1 Jul 10 19:07 /dev/ttyUSB1
```

If you see similar result and you are a member of `plugdev` group, then the set up is complete.

The `/dev/ttyUSBn` interface with lower number is used for JTAG communication. The other interface is routed to ESP32’s serial port (UART) used for upload of application to ESP32’s flash.

Now ESP32 WROVER KIT's JTAG interface should be available to the OpenOCD. To carry on with debugging environment setup, proceed to section [Run OpenOCD](#).

MacOS

On macOS, using FT2232 for JTAG and serial port at the same time needs some additional steps. When the OS loads FTDI serial port driver, it does so for both channels of FT2232 chip. However only one of these channels is used as a serial port, while the other is used as JTAG. If the OS has loaded FTDI serial port driver for the channel used for JTAG, OpenOCD will not be able to connect to the chip. There are two ways around this:

1. Manually unload the FTDI serial port driver before starting OpenOCD, start OpenOCD, then load the serial port driver.
2. Modify FTDI driver configuration so that it doesn't load itself for channel B of FT2232 chip, which is the channel used for JTAG on WROVER KIT.

Manually unloading the driver

1. Install FTDI driver from <http://www.ftdichip.com/Drivers/VCP.htm>
2. Connect USB cable to the WROVER KIT.
3. Unload the serial port driver:

```
sudo kextunload -b com.FTDI.driver.FTDIUSBSerialDriver
```

In some cases you may need to unload Apple's FTDI driver as well:

```
sudo kextunload -b com.apple.driver.AppleUSBFTDI
```

4. Run OpenOCD (paths are given for downloadable OpenOCD archive):

```
bin/openocd -s share/openocd/scripts -f interface/ftdi/esp32_devkitj_v1.cfg -f board/esp-wroom-32.cfg
```

Or, if OpenOCD was built from source:

```
src/openocd -s tcl -f interface/ftdi/esp32_devkitj_v1.cfg -f board/esp-wroom-32.cfg
```

5. In another terminal window, load FTDI serial port driver again:

```
sudo kextload -b com.FTDI.driver.FTDIUSBSerialDriver
```

Note that if you need to restart OpenOCD, there is no need to unload FTDI driver again — just stop OpenOCD and start it again. The driver only needs to be unloaded if WROVER KIT was reconnected or power was toggled.

This procedure can be wrapped into a shell script, if desired.

Modifying FTDI driver

In a nutshell, this approach requires modification to FTDI driver configuration file, which prevents the driver from being loaded for channel B of FT2232H.

Note: Other boards may use channel A for JTAG, so use this option with caution.

Warning: This approach also needs signature verification of drivers to be disabled, so may not be acceptable for all users.

1. Open FTDI driver configuration file using a text editor (note sudo):

```
sudo nano /Library/Extensions/FTDIUSBSerialDriver.kext/Contents/Info.plist
```

2. Find and delete the following lines:

```
<key>FT2232H_B</key>
<dict>
    <key>CFBundleIdentifier</key>
    <string>com.FTDI.driver.FTDIUSBSerialDriver</string>
    <key>IOClass</key>
    <string>FTDIUSBSerialDriver</string>
    <key>IOProviderClass</key>
    <string>IOUSBInterface</string>
    <key>bConfigurationValue</key>
    <integer>1</integer>
    <key>bInterfaceNumber</key>
    <integer>1</integer>
    <key>bcdDevice</key>
    <integer>1792</integer>
    <key>idProduct</key>
    <integer>24592</integer>
    <key>idVendor</key>
    <integer>1027</integer>
</dict>
```

3. Save and close the file
4. Disable driver signature verification:
 - (a) Open Apple logo menu, choose “Restart...”
 - (b) When you hear the chime after reboot, press CMD+R immediately
 - (c) Once Recovery mode starts up, open Terminal
 - (d) Run the command:

```
csrutil enable --without kext
```

 - (e) 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 programming.
- `exit` - Optional. Finally exit OpenOCD.

You are now ready to start application debugging. Follow steps described in section below.

4.7.6 Launching Debugger

The toolchain for ESP32 features GNU Debugger, in short GDB. It is available with other toolchain programs under filename `xtensa-esp32-elf-gdb`. GDB can be called and operated directly from command line in a terminal. Another option is to call it from within IDE (like Eclipse, Visual Studio Code, etc.) and operate indirectly with help of GUI instead of typing commands in a terminal.

Both options of using debugger are discussed under links below.

- [Eclipse](#)
- [Command Line](#)

It is recommended to first check if debugger works from [Command Line](#) and then move to using [Eclipse](#).

4.7.7 Debugging Examples

This section is intended for users not familiar with GDB. It presents example debugging session from [Eclipse](#) using simple application available under [get-started/blink](#) and covers the following debugging actions:

1. [Navigating though the code, call stack and threads](#)
2. [Setting and clearing breakpoints](#)
3. [Halting the target manually](#)
4. [Stepping through the code](#)
5. [Checking and setting memory](#)
6. [Watching and setting program variables](#)
7. [Setting conditional breakpoints](#)

Similar debugging actions are provided using GDB from [Command Line](#).

Before proceeding to examples, set up your ESP32 target and load it with [get-started/blink](#).

4.7.8 Building OpenOCD from Sources

Please refer to separate documents listed below, that describe build process.

Note: Examples of invoking OpenOCD in this document assume using pre-built binary distribution described in section [Setup of OpenOCD](#). To use binaries build locally from sources, change the path to OpenOCD executable to `src/openocd` and the path to configuration files to `-s tcl`.

Example of invoking OpenOCD build locally from sources:

```
src/openocd -s tcl -f interface/ftdi/esp32_devkitj_v1.cfg -f board/esp-wroom-32.cfg
```

4.7.9 Tips and Quirks

This section provides collection of links to all tips and quirks referred to from various parts of this guide.

- [Breakpoints and watchpoints available](#)
- [What else should I know about breakpoints?](#)
- [Why stepping with “next” does not bypass subroutine calls?](#)
- [Support options for OpenOCD at compile time](#)
- [FreeRTOS support](#)
- [Why to set SPI flash voltage in OpenOCD configuration?](#)
- [Optimize JTAG speed](#)
- [What is the meaning of debugger’s startup commands?](#)

- *Configuration of OpenOCD for specific target*
- *How debugger resets ESP32?*
- *Do not use JTAG pins for something else*
- *Reporting issues with OpenOCD / GDB*

4.7.10 Related Documents

Using Debugger

This section covers configuration and running debugger either from *Eclipse* or *Command Line*. It is recommended to first check if debugger works from *Command Line* and then move to using Eclipse.

Eclipse

Debugging functionality is provided out of box in standard Eclipse installation. Another option is to use pluggins like “GDB Hardware Debugging” plugin. We have found this plugin quite convenient and decided to use throughout this guide.

To begin with, install “GDB Hardware Debugging” plugin by opening Eclipse and going to *Help > Install New Software*.

Once installation is complete, configure debugging session following steps below. Please note that some of configuration parameters are generic and some are project specific. This will be shown below by configuring debugging for “blink” example project. If not done already, add this project to Eclipse workspace following guidance in section *Build and Flash with Eclipse IDE*. The source of `get-started/blink` application is available in `examples` directory of ESP-IDF repository.

1. In Eclipse go to *Run > Debug Configuration*. A new window will open. In the window’s left pane double click “GDB Hardware Debugging” (or select “GDB Hardware Debugging” and press the “New” button) to create a new configuration.
2. In a form that will show up on the right, enter the “Name:” of this configuration, e.g. “Blink checking”.
3. On the “Main” tab below, under “Project:”, press “Browse” button and select the “blink” project.
4. In next line “C/C++ Application:” press “Browse” button and select “blink.elf” file. If “blink.elf” is not there, then likely this project has not been build yet. See *Build and Flash with Eclipse IDE* how to do it.
5. Finally, under “Build (if required) before launching” click “Disable auto build”.

A sample window with settings entered in points 1 - 5 is shown below.

6. Click “Debugger” tab. In field “GDB Command” enter `xtensa-esp32-elf-gdb` to invoke debugger.
7. Change default configuration of “Remote host” by entering `3333` under the “Port number”.

Configuration entered in points 6 and 7 is shown on the following picture.

8. The last tab to that requires changing of default configuration is “Startup”. Under “Initialization Commands” uncheck “Reset and Delay (seconds)” and “Halt”. Then, in entry field below, type `mon reset halt` and `x $a1=0` (in two separate lines).

Note: If you want to update image in the flash automatically before starting new debug session add the following lines of commands at the beginning of “Initialization Commands” textbox:

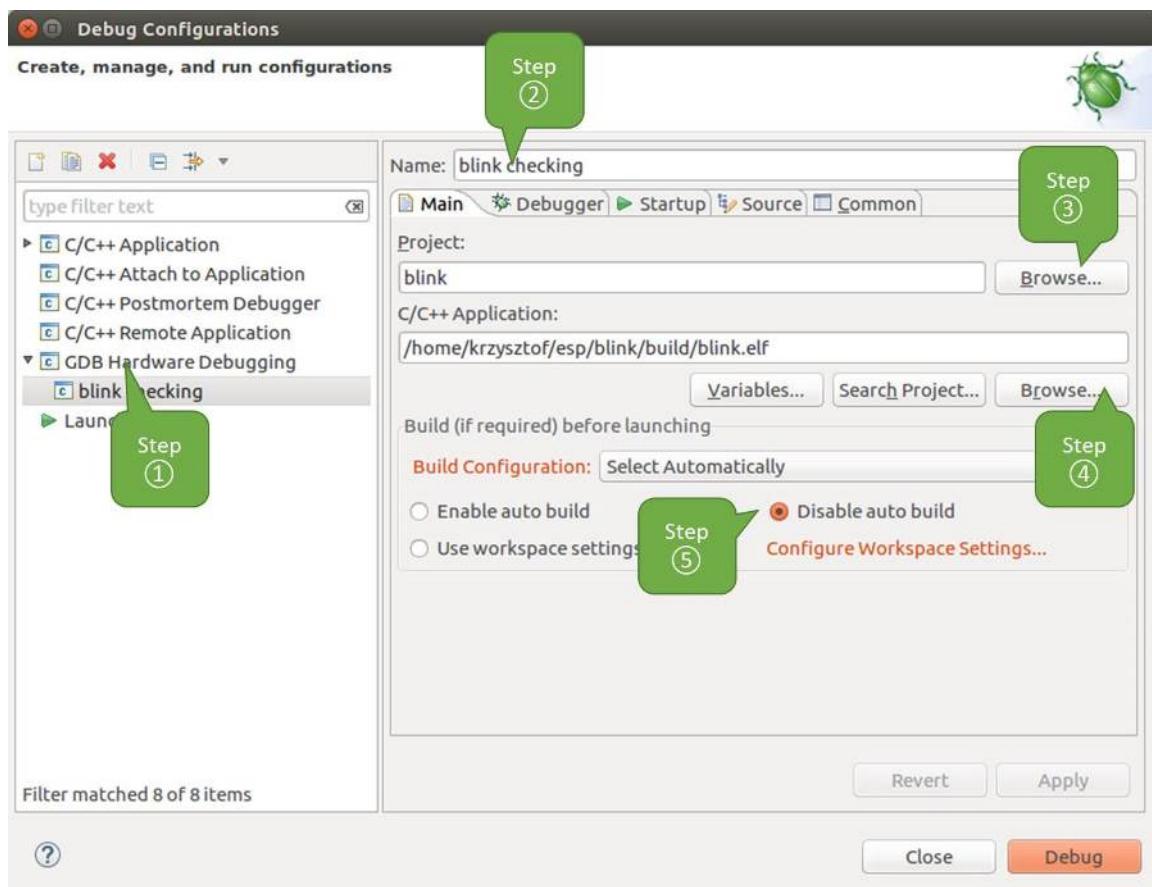


Fig. 4.3: Configuration of GDB Hardware Debugging - Main tab

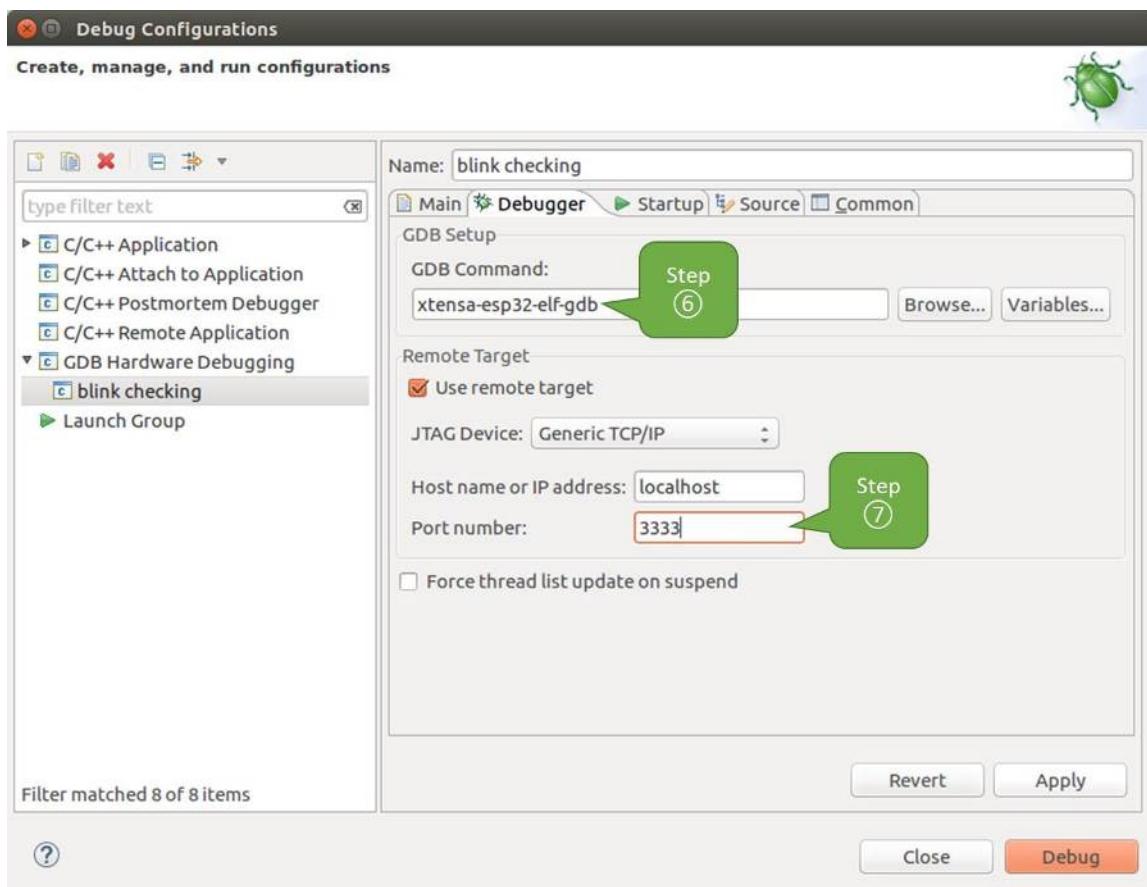


Fig. 4.4: Configuration of GDB Hardware Debugging - Debugger tab

```
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 breakpoint 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.

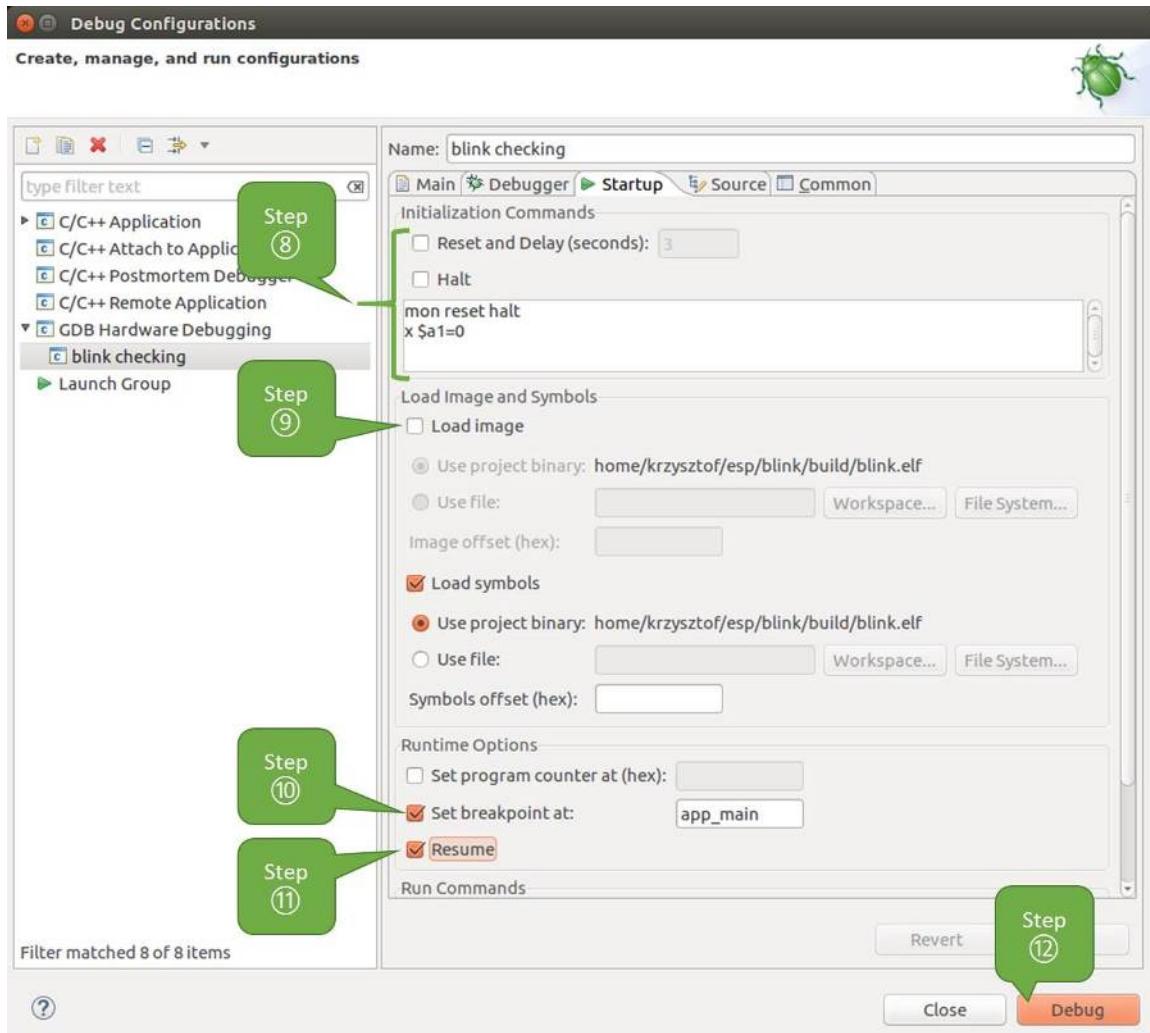


Fig. 4.5: Configuration of GDB Hardware Debugging - Startup tab

If the “Startup” sequence looks convoluted and respective “Initialization Commands” are not clear to you, check [What is the meaning of debugger’s startup commands?](#) for additional explanation.

12. If you previously completed [Configuring ESP32 Target](#) steps described above, so the target is running and ready to talk to debugger, go right to debugging by pressing “Debug” button. Otherwise press “Apply” to save changes, go back to [Configuring ESP32 Target](#) and return here to start debugging.

Once all 1 - 12 configuration steps are satisfied, the new Eclipse perspective called “Debug” will open as shown on example picture below.

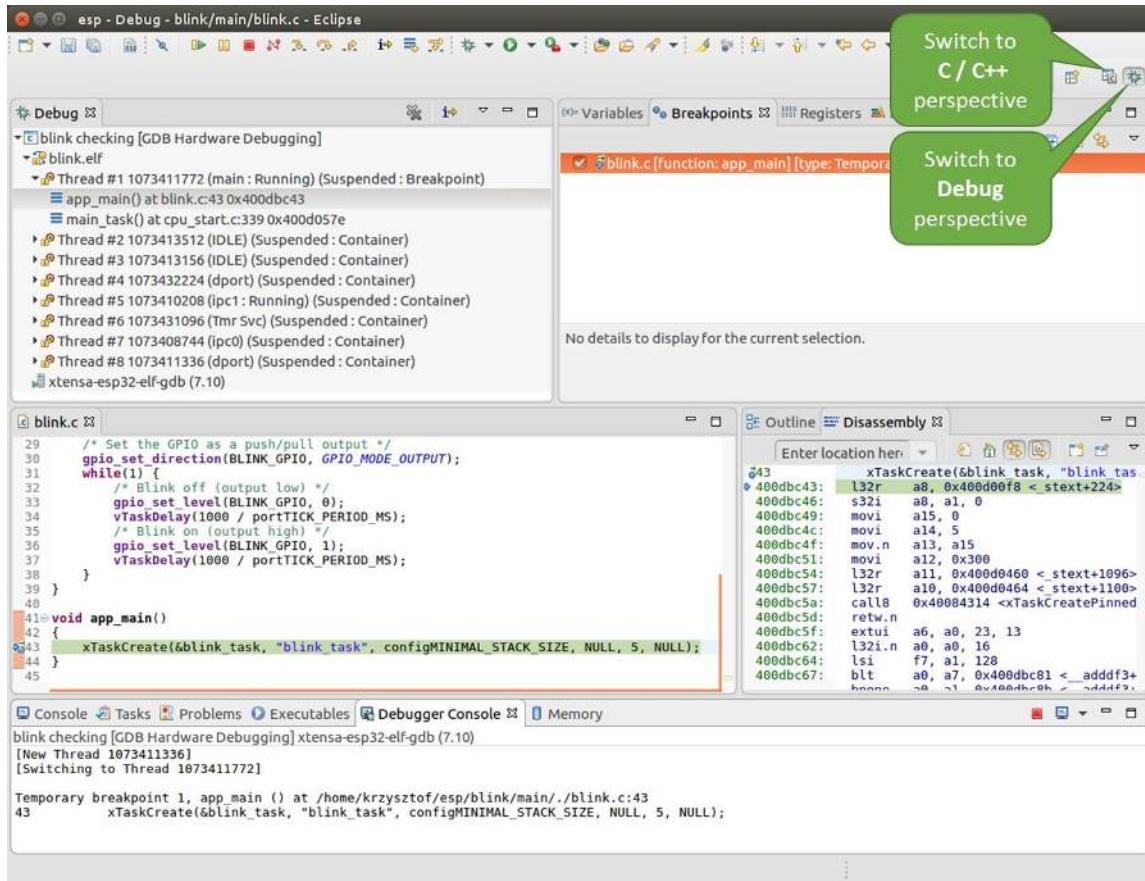


Fig. 4.6: Debug Perspective in Eclipse

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
thb app_main
```

```
x $a1=0  
c
```

Save this file in current directory.

For more details what's inside `gdbinit` file, see [What is the meaning of debugger's startup commands?](#)

4. Now you are ready to launch GDB. Type the following in terminal:

```
xtensa-esp32-elf-gdb -x gdbinit build/blink.elf
```

5. If previous steps have been done correctly, you will see a similar log concluded with (gdb) prompt:

```
user-name@computer-name:~/esp/blink$ xtensa-esp32-elf-gdb -x gdbinit build/blink.  
↳elf  
GNU gdb (crosstool-NG crosstool-ng-1.22.0-61-gab8375a) 7.10  
Copyright (C) 2015 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "--host=x86_64-build_pc-linux-gnu --target=xtensa-  
↳esp32-elf".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word"....  
Reading symbols from build/blink.elf...done.  
0x400d10d8 in esp_vApplicationIdleHook () at /home/user-name/esp/esp-idf/  
↳components/esp32/.freertos_hooks.c:52  
52         asm("waiti 0");  
JTAG tap: esp32.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica), part:↳  
↳0x2003, ver: 0x1)  
JTAG tap: esp32.slave tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica), part:↳  
↳0x2003, ver: 0x1)  
esp32: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).  
esp32: Core was reset (pwrstat=0x5F, after clear 0x0F).  
Target halted. PRO_CPU: PC=0x5000004B (active) APP_CPU: PC=0x00000000  
esp32: target state: halted  
esp32: Core was reset (pwrstat=0x1F, after clear 0x0F).  
Target halted. PRO_CPU: PC=0x40000400 (active) APP_CPU: PC=0x40000400  
esp32: target state: halted  
Hardware assisted breakpoint 1 at 0x400db717: file /home/user-name/esp/blink/main/  
↳./blink.c, line 43.  
0x0: 0x00000000  
Target halted. PRO_CPU: PC=0x400DB717 (active) APP_CPU: PC=0x400D10D8  
[New Thread 1073428656]  
[New Thread 1073413708]  
[New Thread 1073431316]  
[New Thread 1073410672]  
[New Thread 1073408876]  
[New Thread 1073432196]  
[New Thread 1073411552]  
[Switching to Thread 1073411996]  
  
Temporary breakpoint 1, app_main () at /home/user-name/esp/blink/main./blink.c:43
```

```
43     xTaskCreate(&blink_task, "blink_task", 512, NULL, 5, NULL);
(gdb)
```

Note the third line from bottom that shows debugger halting at breakpoint established in `gdbinit` file at function `app_main()`. Since the processor is halted, the LED should not be blinking. If this is what you see as well, you are ready to start debugging.

If you are not quite sure how to use GDB, check [Command Line](#) example debugging session in section [Debugging Examples](#).

Debugging Examples

This section describes debugging with GDB from [Eclipse](#) as well as from [Command Line](#).

Eclipse

Verify if your target is ready and loaded with [get-started/blink](#) example. Configure and start debugger following steps in section [Eclipse](#). Pick up where target was left by debugger, i.e. having the application halted at breakpoint established at `app_main()`.

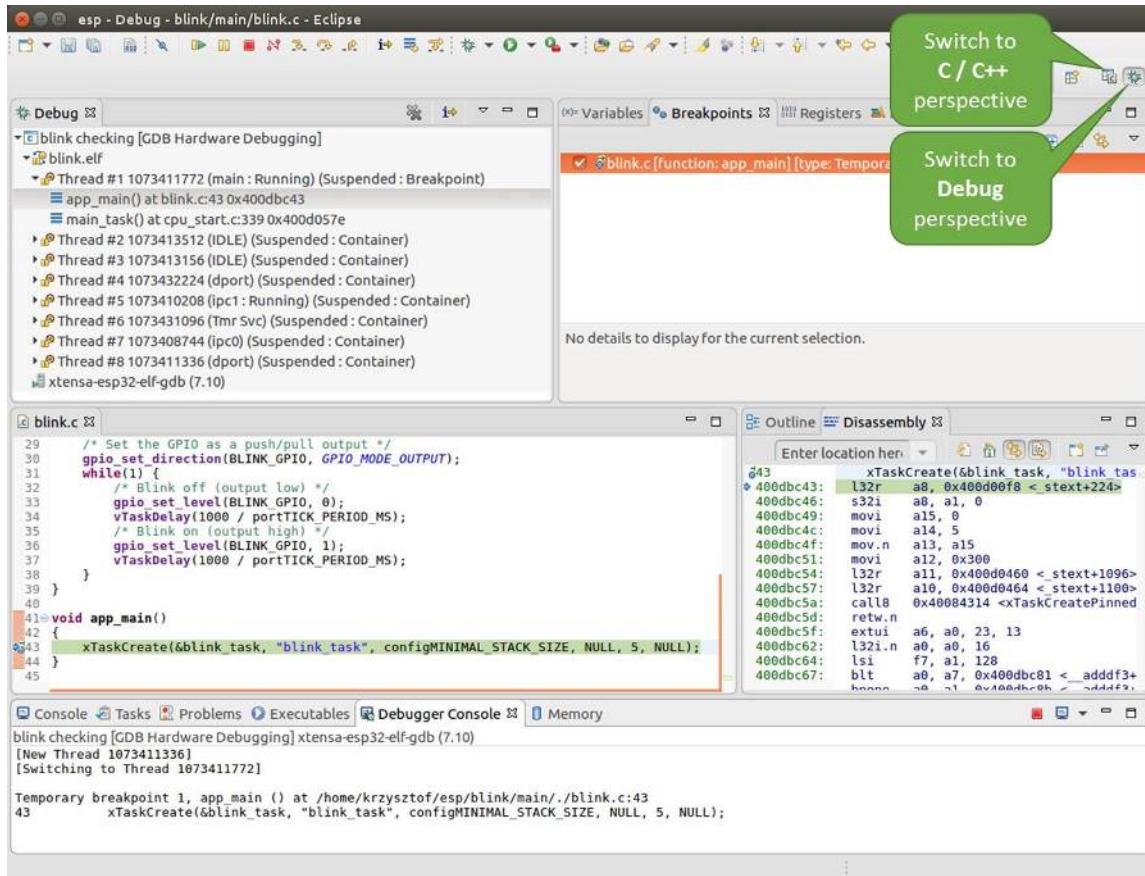


Fig. 4.7: Debug Perspective in Eclipse

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.

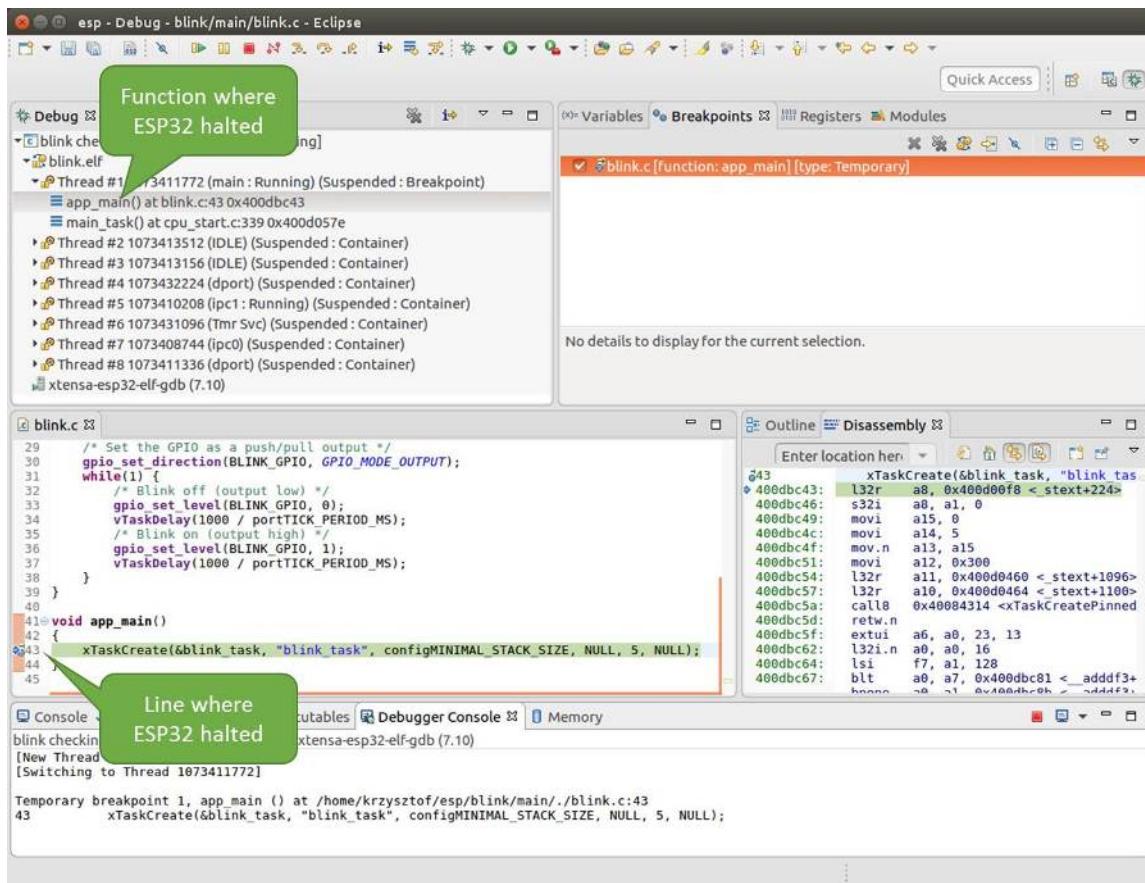


Fig. 4.8: Target halted during debugging

Specific thread where the program halted is expanded showing the call stack. It represents function calls that lead up to the highlighted line of code, where the target halted. The first line of call stack under Thread #1 contains the last called function `app_main()`, that in turn was called from function `main_task()` shown in a line below. Each line

of the stack also contains the file name and line number where the function was called. By clicking / highlighting the stack entries, in window below, you will see contents of this file.

By expanding threads you can navigate throughout the application. Expand Thread #5 that contains much longer call stack. You will see there, besides function calls, numbers like 0x4000000c. They represent addresses of binary code not provided in source form.

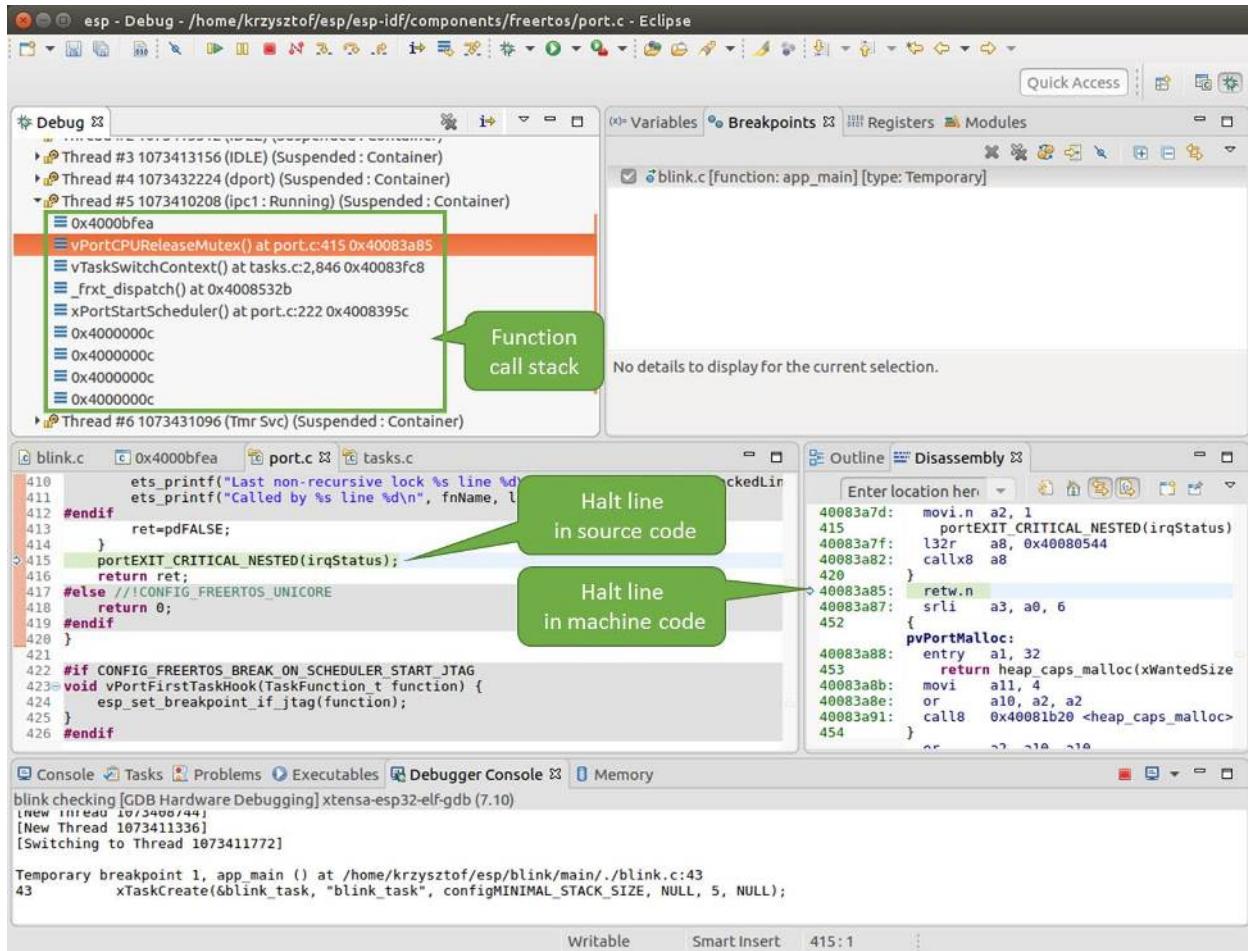


Fig. 4.9: Navigate through the call stack

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 click on number 33 in file `blink.c` file. A dialog

will open where you can confirm your selection by pressing “OK” button. If you do not like to see the dialog just double click the line number. Set another breakpoint in line 36.

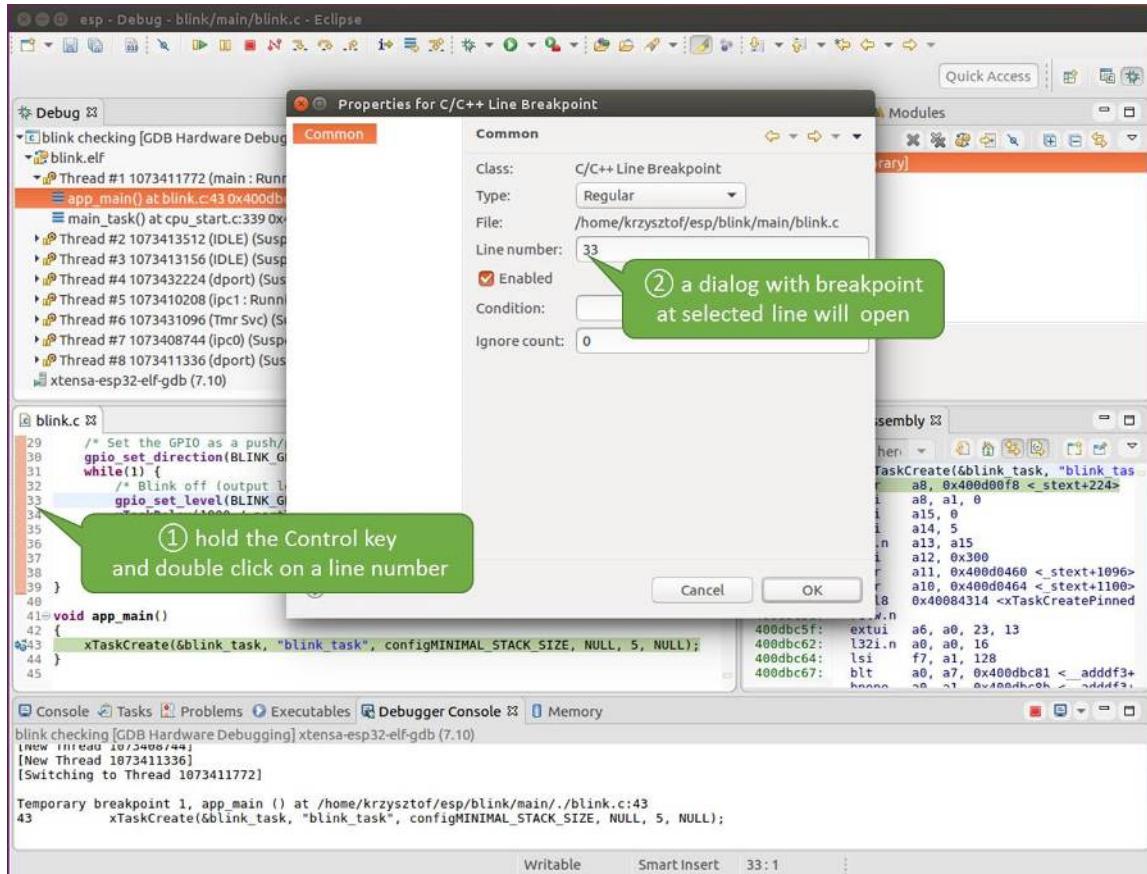


Fig. 4.10: Setting a breakpoint

Information how many breakpoints are set and where is shown in window “Breakpoints” on top right. Click “Show Breakpoints Supported by Selected Target” to refresh this list. Besides the two just set breakpoints the list may contain temporary breakpoint at function `app_main()` established at debugger start. As maximum two breakpoints are allowed (see [Breakpoints and watchpoints available](#)), you need to delete it, or debugging will fail.

If you now click “Resume” (click `blink_task()` under “Tread #8”, if “Resume” button is grayed out), the processor will run and halt at a breakpoint. Clicking “Resume” another time will make it run again, halt on second breakpoint, and so on.

You will be also able to see that LED is changing the state after each click to “Resume” program execution.

Read more about breakpoints under [Breakpoints and watchpoints available](#) and [What else should I know about breakpoints?](#)

Halting the target manually

When debugging, you may resume application and enter code waiting for some event or staying in infinite loop without any break points defined. In such case, to go back to debugging mode, you can break program execution manually by pressing “Suspend” button.

To check it, delete all breakpoints and click “Resume”. Then click “Suspend”. Application will be halted at some random point and LED will stop blinking. Debugger will expand tread and highlight the line of code where application

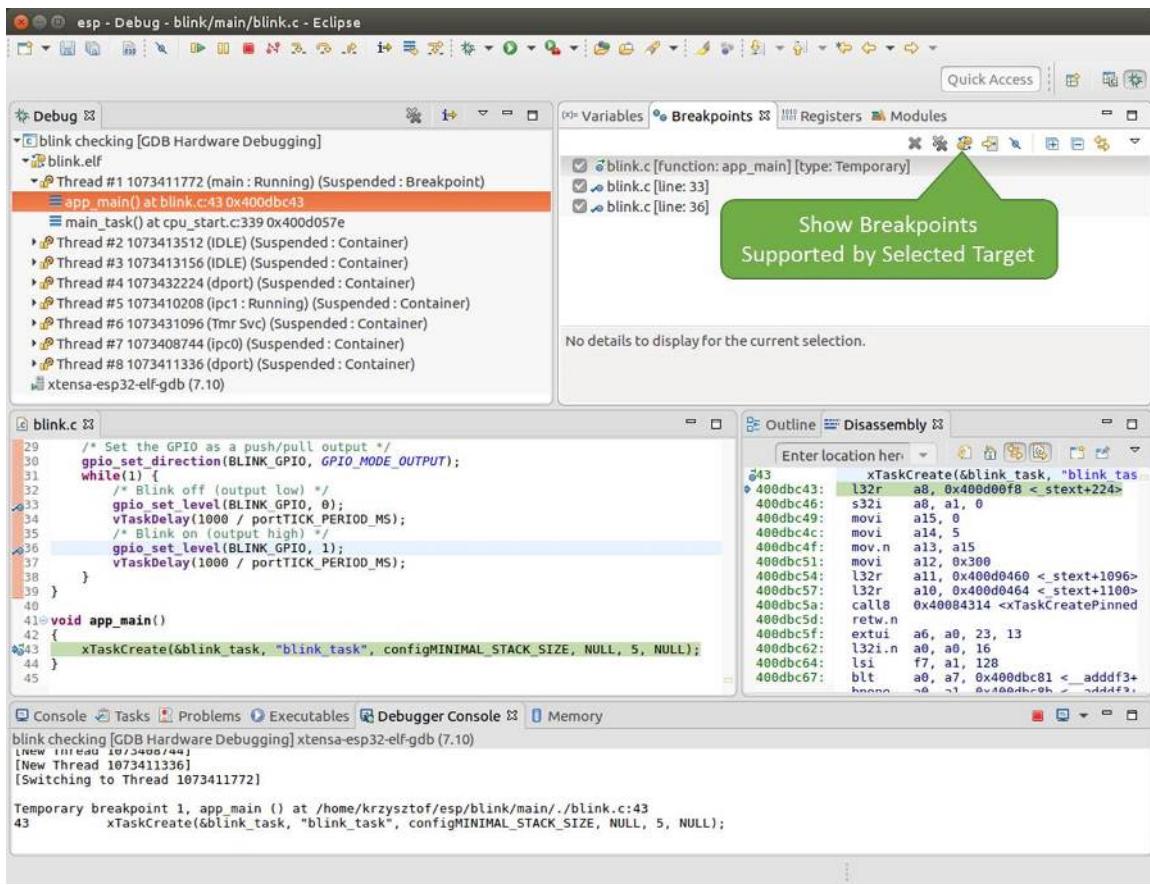


Fig. 4.11: Three breakpoints are set / maximum two are allowed

halted.

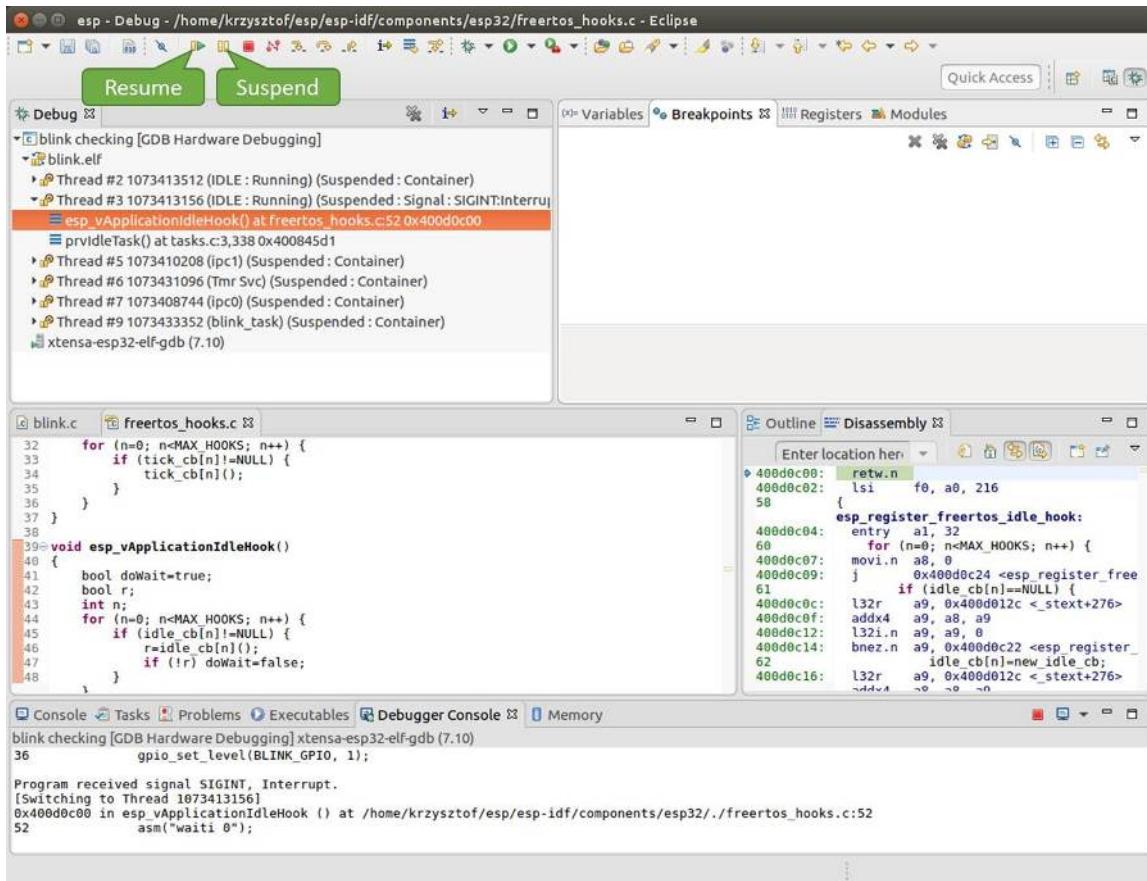


Fig. 4.12: Target halted manually

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.

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.

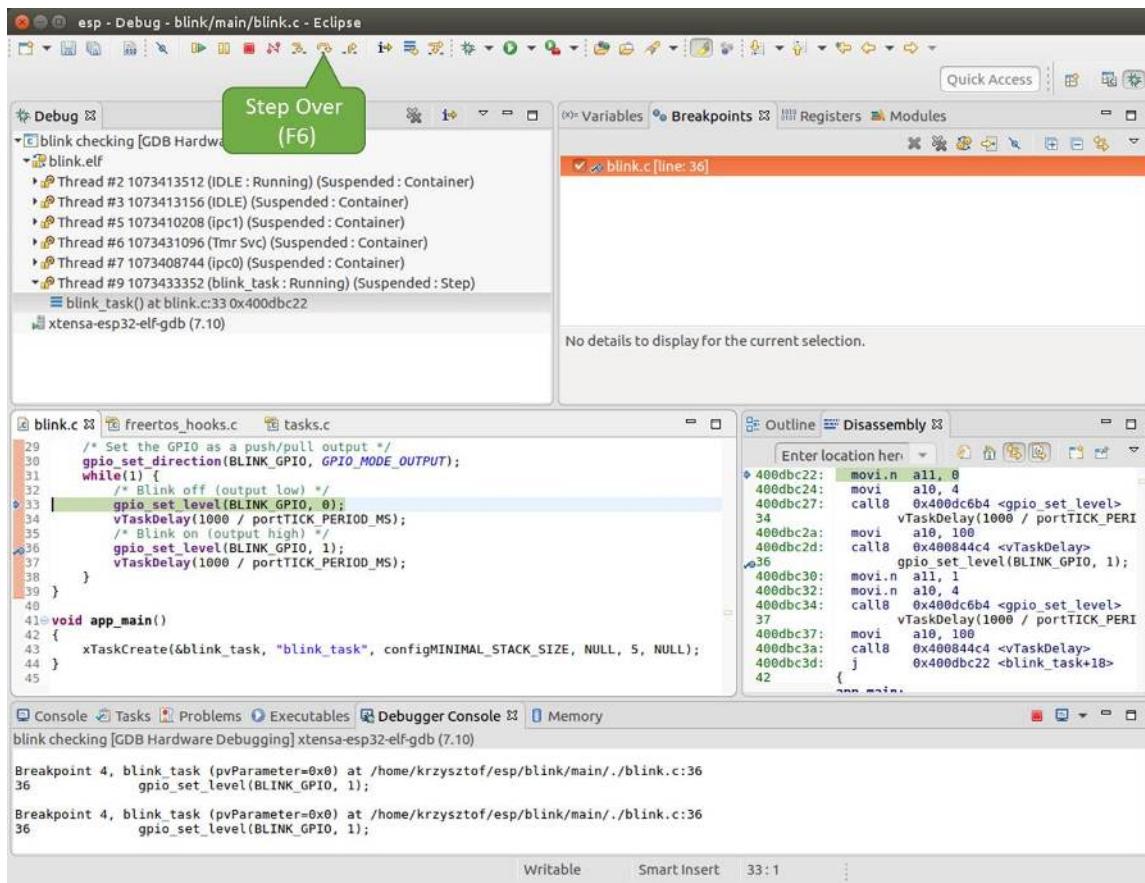


Fig. 4.13: Stepping through the code with “Step Over (F6)”

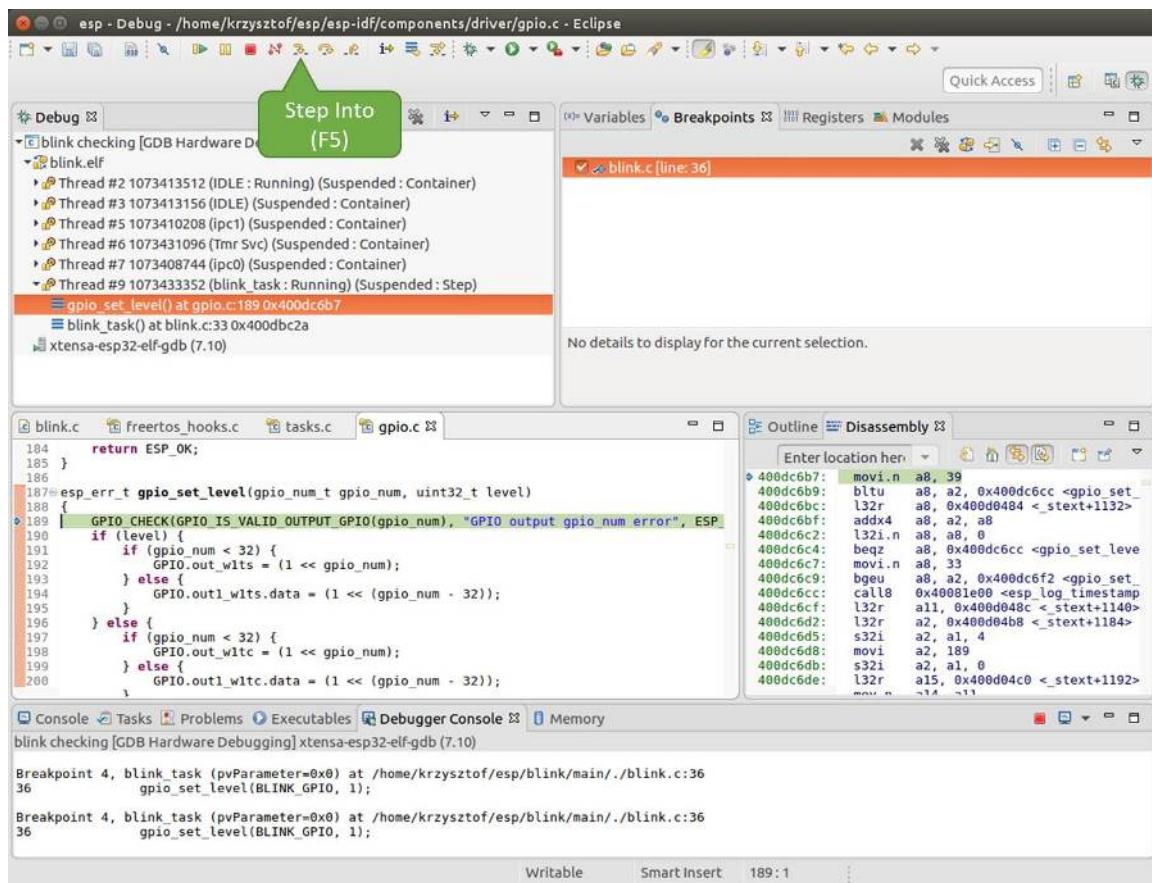


Fig. 4.14: Stepping through the code with “Step Into (F5)”

Checking and setting memory

To display or set contents of memory use “Memory” tab at the bottom of “Debug” perspective.

With the “Memory” tab, we will read from and write to the memory location 0x3FF44004 labeled as GPIO_OUT_REG used to set and clear individual GPIO’s. For more information please refer to [ESP32 Technical Reference Manual](#), chapter IO_MUX and GPIO Matrix.

Being in the same `blink.c` project as before, set two breakpoints right after `gpio_set_level` instruction. Click “Memory” tab and then “Add Memory Monitor” button. Enter 0x3FF44004 in provided dialog.

Now resume program by pressing F8 and observe “Monitor” tab.

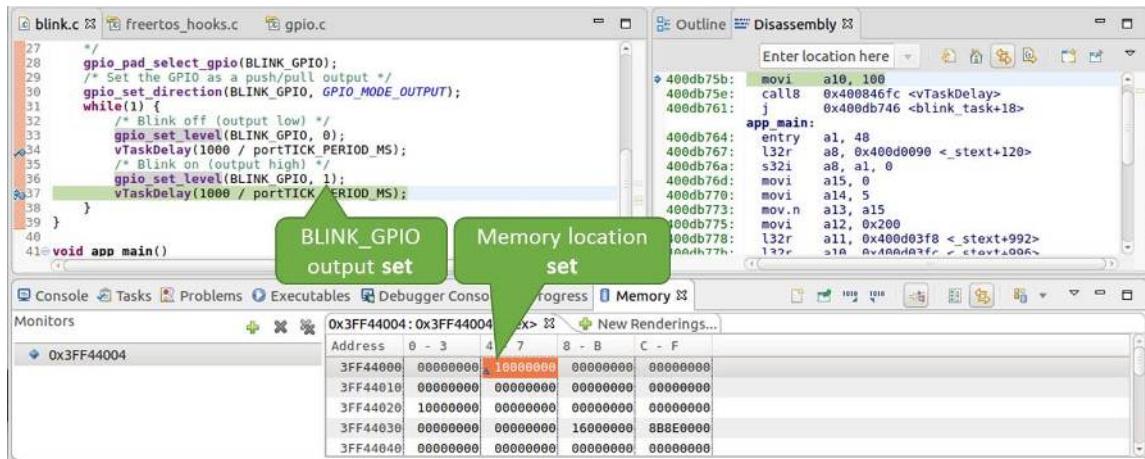


Fig. 4.15: Observing memory location 0x3FF44004 changing one bit to ON”

You should see one bit being flipped over at memory location 0x3FF44004 (and LED changing the state) each time F8 is pressed.

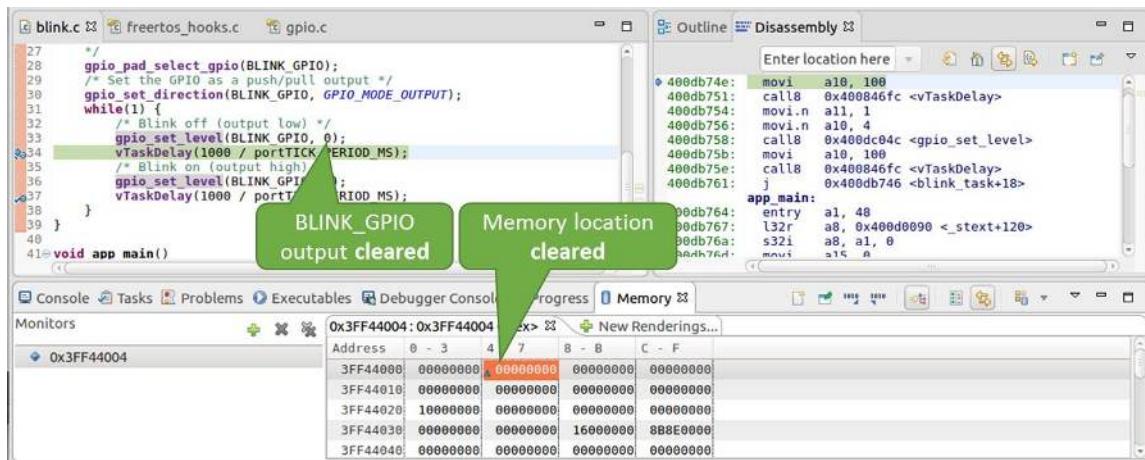


Fig. 4.16: Observing memory location 0x3FF44004 changing one bit to ON”

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 task is checking the value of a program variable as the program runs. To be able to demonstrate this functionality, update file `blink.c` by adding a declaration of a global variable `int i` above definition of function `blink_task`. Then add `i++` inside `loop(1)` of this function to get `i` incremented on each blink.

Exit debugger, so it is not confused with new code, build and flash the code to the ESP and restart debugger. There is no need to restart OpenOCD.

Once application is halted, enter 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.

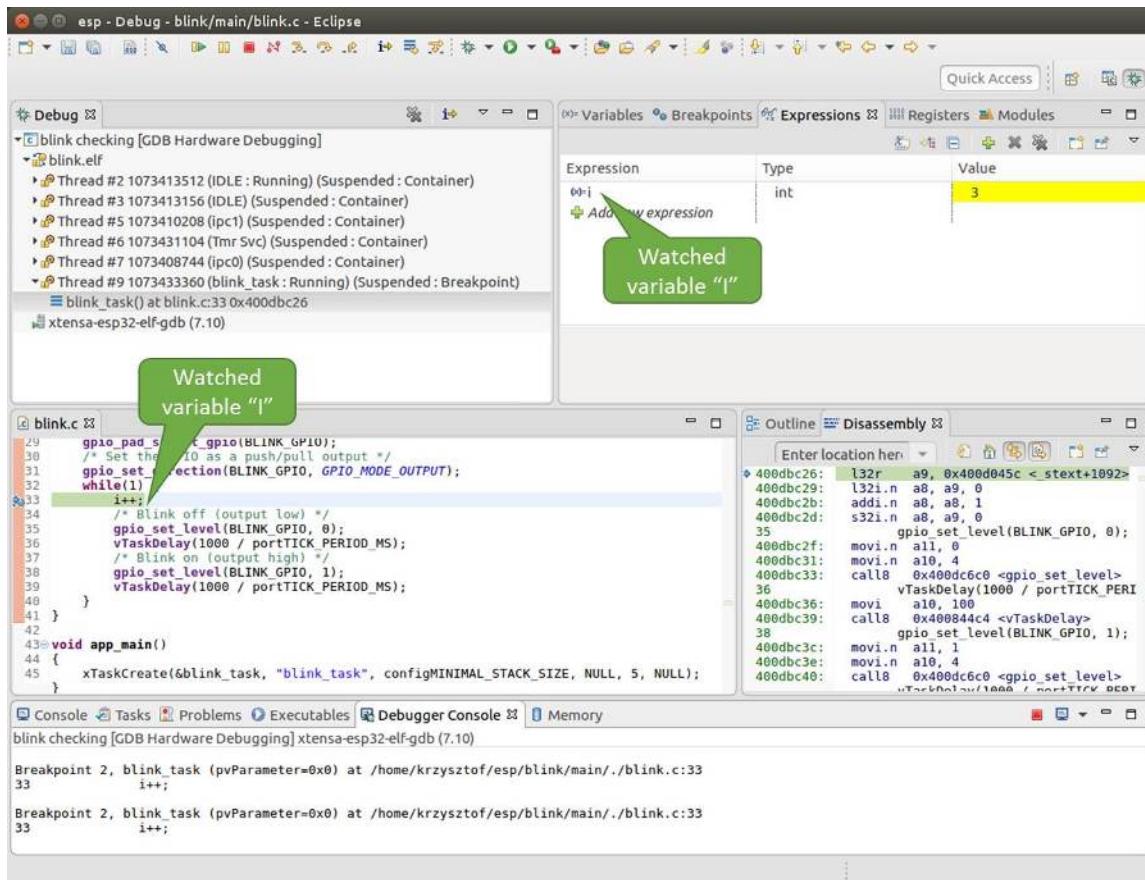


Fig. 4.17: Watching program variable “i”

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`.

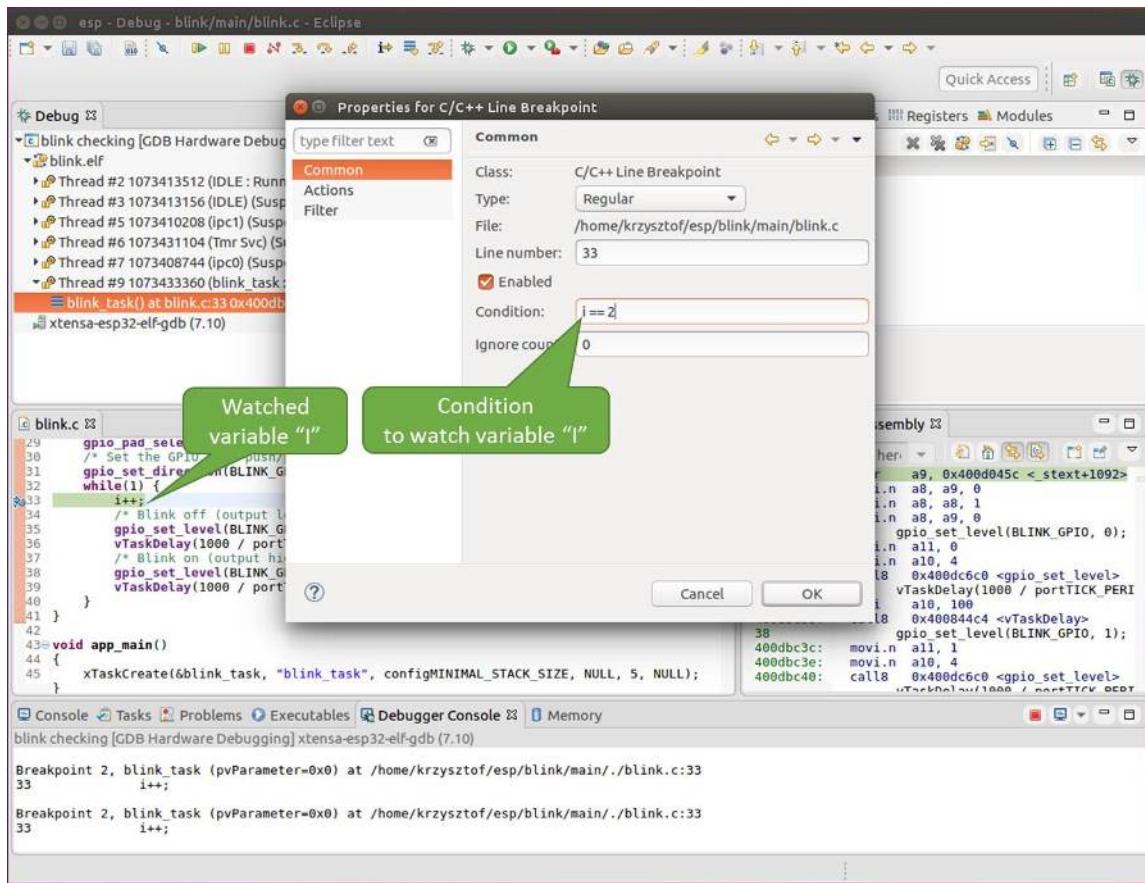


Fig. 4.18: Setting a conditional breakpoint

If current value of `i` is less than 2 (change it if required) and program is resumed, it will blink LED in a loop until condition `i == 2` gets true and then finally halt.

Command Line

Verify if your target is ready and loaded with [get-started/blink](#) example. Configure and start debugger following steps in section [Command Line](#). Pick up where target was left by debugger, i.e. having the application halted at breakpoint established at `app_main()`:

```
Temporary breakpoint 1, app_main () at /home/user-name/esp/blink/main./blink.c:43
43          xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5,_
˓→NULL);
(gdb)
```

Examples in this section

1. *Navigating 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 through the code, call stack and threads

When you see the (gdb) prompt, the application is halted. LED should not be blinking.

To find out where exactly the code is halted, enter `l` or `list`, and debugger will show couple of lines of code around the halt point (line 43 of code in file `blink.c`)

```
(gdb) l
38         }
39     }
40
41     void app_main()
42     {
43         xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5, u
~NULL);
44     }
(gdb)
```

Check how code listing works by entering, e.g. 1 30, 40 to see particular range of lines of code.

You can use `bt` or `backtrace` to see what function calls lead up to this code:

```
(gdb) bt
#0  app_main () at /home/user-name/esp/blink/main./blink.c:43
#1  0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
→esp32./cpu_start.c:339
(gdb)
```

Line #0 of output provides the last function call before the application halted, i.e. `app_main()` we have listed previously. The `app_main()` was in turn called by function `main_task` from line 339 of code located in file `cpu_start.c`.

To get to the context of `main_task` in file `cpu_start.c`, enter frame N, where N = 1, because the `main_task` is listed under #1:

```
(gdb) frame 1
#1 0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
→esp32./cpu_start.c:339
339     app_main();
(gdb)
```

Enter 1 and this will reveal the piece of code that called `app_main()` (in line 339):

```
(gdb) l
334         ;
335     }
336 #endif
337     //Enable allocation in region where the startup stacks were located.
338     heap_caps_enable_nonos_stack_heaps();
339     app_main();
340     vTaskDelete(NULL);
341 }
342
(gdb)
```

By listing some lines before, you will see the function name `main_task` we have been looking for:

```
(gdb) l 326, 341
326     static void main_task(void* args)
327 {
328     // Now that the application is about to start, disable boot watchdogs
329     REG_CLR_BIT(TIMG_WDTCONFIG0_REG(0), TIMG_WDT_FLASHBOOT_MOD_EN_S);
330     REG_CLR_BIT(RTC_CNTL_WDTCONFIG0_REG, RTC_CNTL_WDT_FLASHBOOT_MOD_EN);
331 #if !CONFIG_FREERTOS_UNICORE
332     // Wait for FreeRTOS initialization to finish on APP CPU, before_
→replacing its startup stack
333     while (port_xSchedulerRunning[1] == 0) {
334         ;
335     }
336 #endif
337     //Enable allocation in region where the startup stacks were located.
338     heap_caps_enable_nonos_stack_heaps();
339     app_main();
340     vTaskDelete(NULL);
341 }
(gdb)
```

To see the other code, enter `i threads`. This will show the list of threads running on target:

```
(gdb) i threads
Id  Target Id      Frame
 8  Thread 1073411336 (dport) 0x400d0848 in dport_access_init_core (arg=<optimized_
→out>)
    at /home/user-name/esp/esp-idf/components/esp32./dport_access.c:170
 7  Thread 1073408744 (ipc0) xQueueGenericReceive (xQueue=0x3ffae694, pvBuffer=0x0,
→ xTicksToWait=1644638200,
```

```
xJustPeeking=0) at /home/user-name/esp/esp-idf/components/freertos./queue.c:1452
6   Thread 1073431096 (Tmr Svc) prvTimerTask (pvParameters=0x0)
   at /home/user-name/esp/esp-idf/components/freertos./timers.c:445
5   Thread 1073410208 (ipc1 : Running) 0x4000bfea in ?? ()
4   Thread 1073432224 (dport) dport_access_init_core (arg=0x0)
   at /home/user-name/esp/esp-idf/components/esp32./dport_access.c:150
3   Thread 1073413156 (IDLE) prvIdleTask (pvParameters=0x0)
   at /home/user-name/esp/esp-idf/components/freertos./tasks.c:3282
2   Thread 1073413512 (IDLE) prvIdleTask (pvParameters=0x0)
   at /home/user-name/esp/esp-idf/components/freertos./tasks.c:3282
* 1   Thread 1073411772 (main : Running) app_main () at /home/user-name/esp/blink/
  ↵main./blink.c:43
(gdb)
```

The thread list shows the last function calls per each thread together with the name of C source file if available.

You can navigate to specific thread by entering `thread N`, where N is the thread Id. To see how it works go to thread thread 5:

```
(gdb) thread 5
[Switching to thread 5 (Thread 1073410208)]
#0 0x4000bfea in ?? ()
(gdb)
```

Then check the backtrace:

```
(gdb) bt
#0 0x4000bfea in ?? ()
#1 0x40083a85 in vPortCPUReleaseMutex (mux=<optimized out>) at /home/user-name/esp/
  ↵esp-idf/components/freertos./port.c:415
#2 0x40083fc8 in vTaskSwitchContext () at /home/user-name/esp/esp-idf/components/
  ↵freertos./tasks.c:2846
#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 now enter `c`, the processor will run and halt at a breakpoint. Entering `c` another time will make it run again, halt on second breakpoint, and so on:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB6F6 (active) APP_CPU: PC=0x400D10D8

Breakpoint 2, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/.blink.
→c:33
33     gpio_set_level(BLINK_GPIO, 0);
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB6F8 (active) APP_CPU: PC=0x400D10D8
Target halted. PRO_CPU: PC=0x400DB704 (active) APP_CPU: PC=0x400D10D8

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/.blink.
→c:36
36     gpio_set_level(BLINK_GPIO, 1);
(gdb)
```

You will be also able to see that LED is changing the state only if you resume program execution by entering `c`.

To examine how many breakpoints are set and where, use command `info break`:

```
(gdb) info break
Num      Type            Disp Enb Address      What
2        breakpoint      keep y  0x400db6f6 in blink_task at /home/user-name/esp/blink/
→main/.blink.c:33
      breakpoint already hit 1 time
3        breakpoint      keep y  0x400db704 in blink_task at /home/user-name/esp/blink/
→main/.blink.c:36
      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
(gdb) c
```

```

Continuing.
Target halted. PRO_CPU: PC=0x400DB751 (active)      APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB75B (active)      APP_CPU: PC=0x400D1128

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main./blink.
→c:37
37          vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000010
(gdb)

```

If your are blinking LED connected to GPIO4, then you should see fourth bit being flipped each time the LED changes the state:

```

0x3ff44004: 0x00000000
...
0x3ff44004: 0x00000010

```

Now, when the LED is off, that corresponds to 0x3ff44004: 0x00000000 being displayed, try using set command to set this bit by writing 0x00000010 to the same memory location:

```

(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000000
(gdb) set {unsigned int}0x3FF44004=0x000010

```

You should see the LED to turn on immediately after entering set {unsigned int}0x3FF44004=0x000010 command.

Watching and setting program variables

A common debugging 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
33          i++;
(gdb)

```

Resume application couple more times so `i` gets incremented. Now you can enter `print i` (in short `p i`) to check the current value of `i`:

```
(gdb) p i
$1 = 3
(gdb)
```

To modify the value of `i` use `set` command as below (you can then print it out to check if it has been indeed changed):

```
(gdb) set var i = 0
(gdb) p i
$3 = 0
(gdb)
```

You may have up to two watchpoints, see [Breakpoints and watchpoints available](#).

Setting conditional breakpoints

Here comes more interesting part. You may set a breakpoint to halt the program execution, if certain condition is satisfied. Delete existing breakpoints and try this:

```
(gdb) break blink.c:34 if (i == 2)
Breakpoint 3 at 0x400db753: file /home/user-name/esp/blink/main./blink.c, line 34.
(gdb)
```

Above command sets conditional breakpoint to halt program execution in line 34 of `blink.c` if `i == 2`.

If current value of `i` is less than 2 and program is resumed, it will blink LED in a loop until condition `i == 2` gets true and then finally halt:

```
(gdb) set var i = 0
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB755 (active)      APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB753 (active)      APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB755 (active)      APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB753 (active)      APP_CPU: PC=0x400D112C

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main./blink.
→c:34
34          gpio_set_level(BLINK_GPIO, 0);
(gdb)
```

Obtaining help on commands

Commands presented so far should provide are very basis and intended to let you quickly get started with JTAG debugging. Check help what are the other commands at your disposal. To obtain help on syntax and functionality of particular command, being at `(gdb)` prompt type `help` and command name:

```
(gdb) help next
Step program, proceeding through subroutine calls.
Usage: next [N]
Unlike "step", if the current source line calls a subroutine,
```

```
this command does not enter the subroutine, but instead steps over  
the call, in effect treating it as a single source line.  
(gdb)
```

By typing just `help`, you will get top level list of command classes, to aid you drilling down to more details. Optionally refer to available GDB cheat sheets, for instance <http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>. Good to have as a reference (even if not all commands are applicable in an embedded environment).

Ending debugger session

To quit debugger enter `q`:

```
(gdb) q  
A debugging session is active.  
  
Inferior 1 [Remote target] will be detached.  
  
Quit anyway? (y or n) y  
Detaching from program: /home/user-name/esp/blink/build/blink.elf, Remote target  
Ending remote debugging.  
user-name@computer-name:~/esp/blink$
```

Tips and Quirks

This section provides collection of all tips and quirks referred to from various parts of this guide.

Breakpoints and watchpoints available

The ESP32 supports 2 hardware breakpoints. It also supports two watchpoints, so two variables can be watched for change or read by the GDB command `watch myVariable`. Note that menuconfig option `FREERTOS_WATCHPOINT_END_OF_STACK` uses the 2nd watchpoint and will not provide expected results, if you also try to use it within OpenOCD / GDB. See menuconfig's help for detailed description.

What else should I know about breakpoints?

Normal GDB breakpoints (`b myFunction`) can only be set in IRAM, because that memory is writable. Setting these types of breakpoints in code in flash will not work. Instead, use a hardware breakpoint (`hb myFunction`).

Why stepping with “next” does not bypass subroutine calls?

When stepping through the code with `next` command, GDB is internally setting a breakpoint (one out of two available) ahead in the code to bypass the subroutine calls. This functionality will not work, if the two available breakpoints are already set elsewhere in the code. If this is the case, delete breakpoints to have one “spare”. With both breakpoints already used, stepping through the code with `next` command will work as like with `step` command and debugger will step inside subroutine calls.

Support options for OpenOCD at compile time

The ESP-IDF code has various support options for OpenOCD set at compile time: it can stop execution when the first thread is started and break the system if a panic or unhandled exception is thrown. First option is disabled and second enabled by default and both can be changed using the esp-idf configuration menu. Please see the [make menuconfig](#) menu for more details.

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 its internal voltage regulator used to supply power to external SPI flash chip. If binary level on MTDI pin on power up is low, the voltage regulator is set to deliver 3.3V, if it is high, then the voltage is set to 1.8V. The MTDI pin should have a pull-up or may rely on internal weak pull down resistor (see ESP32 Datasheet for details), depending on the type of SPI chip used. Once JTAG is connected, it overrides the pull-up or pull-down resistor that is supposed to do the bootstrapping.

To handle this issue OpenOCD's board configuration file (e.g. `boards\esp-wroom-32.cfg` for ESP-WROOM-32 module) provides `ESP32_FLASH_VOLTAGE` parameter to set the idle state of the TDO line to a specified binary level, therefore reducing the chance of a bad bootup of application due to incorrect flash voltage.

Check specification of ESP32 module connected to JTAG, what is the power supply voltage of SPI flash chip. Then set `ESP32_FLASH_VOLTAGE` accordingly. Most WROOM modules use 3.3V flash, while WROVER modules use 1.8V flash.

Optimize JTAG speed

In order to achieve higher data rates and minimize number of dropped packets it is recommended to optimize setting of JTAG clock frequency, so it is at maximum and still provides stable operation of JTAG. To do so use the following tips.

1. The upper limit of JTAG clock frequency is 20 MHz if CPU runs at 80 MHz, or 26 MHz if CPU runs at 160 MHz or 240 MHz.
2. Depending on particular JTAG adapter and the length of connecting cables, you may need to reduce JTAG frequency below 20 / 26 MHz.
3. In particular reduce frequency, if you get DSR/DIR errors (and they do not relate to OpenOCD trying to read from a memory range without physical memory being present there).
4. ESP-WROVER-KIT operates stable at 20 / 26 MHz.

What is the meaning of debugger's startup commands?

On startup, debugger is issuing sequence of commands to reset the chip and halt it at specific line of code. This sequence (shown below) is user defined to pick up at most convenient / appropriate line and start debugging.

- `mon reset halt` — reset the chip and keep the CPUs halted
- `thb app_main` — insert a temporary hardware breakpoint at `app_main`, put here another function name if required
- `x $a1=0` — this is the tricky part. As far as we can tell, there is no way for a `mon` command to tell GDB that the target state has changed. GDB will assume that whatever stack the target had before `mon reset halt` will still be valid. In fact, after reset the target state will change and executing `x $a1=0` is a way to force GDB to get new state from the target.
- `c` — resume the program. It will then stop at breakpoint inserted at `app_main`.

Configuration of OpenOCD for specific target

OpenOCD needs to be told what JTAG adapter **interface** to use, as well as what type of **board** and processor the JTAG adapter is connected to. To do so, use existing configuration files located in OpenOCD's `share/openocd/scripts/interface` and `share/openocd/scripts/board` folders.

For example, if you connect to ESP-WROVER-KIT with ESP-WROOM-32 module installed (see section [ESP32 WROVER KIT](#)), use the following configuration files:

- `interface/ftdi/esp32_devkitj_v1.cfg`
- `board/esp-wroom-32.cfg`

Optionally prepare configuration by yourself. To do so, you can check existing files and modify them to match your specific hardware. Below is the summary of available configuration parameters for **board** configuration.

Adapter's clock speed

```
adapter_khz 20000
```

See [Optimize JTAG speed](#) for guidance how to set this value.

Single core debugging

```
set ESP32_ONLYCPU 1
```

Comment out this line for dual core debugging.

Disable RTOS support

```
set ESP32RTOS 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 (FFFFFF) indicates target still busy!
cpu0: xtensa_resume (line 431): DSR (FFFFFF) indicates DIR instruction generated an_
<exception>
cpu0: xtensa_resume (line 431): DSR (FFFFFF) indicates DIR instruction generated an_
<overrun>
```

```
cpul: xtensa_resume (line 431): DSR (FFFFFFF) indicates target still busy!
cpul: xtensa_resume (line 431): DSR (FFFFFFF) indicates DIR instruction generated an_
↳exception!
cpul: xtensa_resume (line 431): DSR (FFFFFFF) 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.

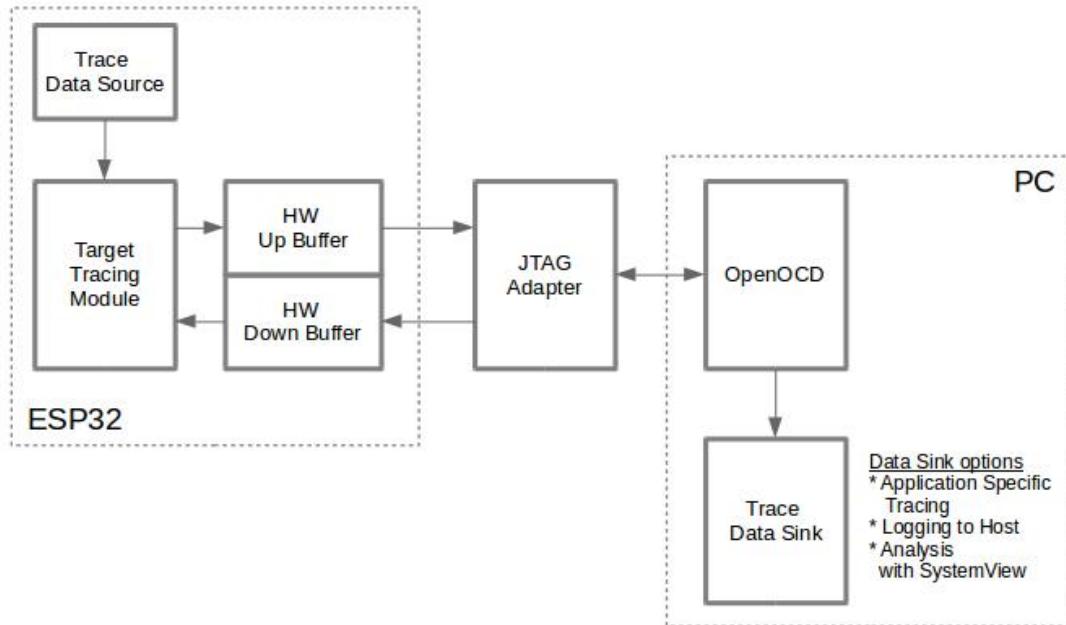


Fig. 4.19: Tracing Components when Working Over JTAG

Modes of Operation

The library supports two modes of operation:

Post-mortem mode. This is the default mode. The mode does not need interaction from the host side. In this mode tracing module does not check whether host has read all the data from *HW UP BUFFER* buffer and overwrites old data with the new ones. This mode is useful when only the latest trace data are interesting to the user, e.g. for analyzing program's behaviour just before the crash. Host can read the data later on upon user request, e.g. via special OpenOCD command in case of working via JTAG interface.

Streaming mode. Tracing module enters this mode when host connects to ESP32. In this mode before writing new data to *HW UP BUFFER* tracing module checks that there is enough space in it and if necessary waits for the host to read data and free enough memory. Maximum waiting time is controlled via timeout values passed by users to corresponding API routines. So when application tries to write data to trace buffer using finite value of the maximum waiting time it is possible situation that this data will be dropped. Especially this is true for tracing from time critical code (ISRs, OS scheduler code etc.) when infinite timeouts can lead to system malfunction. In order to avoid loss of such critical data developers can enable additional data buffering via menuconfig option `ESP32_APPTRACE_PENDING_DATA_SIZE_MAX`. This macro specifies the size of data which can be buffered in above conditions. The option can also help to overcome situation when data transfer to the host is temporarily slowed down, e.g due to USB bus congestions etc. But it will not help when average bitrate of trace data stream exceeds HW interface capabilities.

Configuration Options and Dependencies

Using of this feature depends on two components:

1. **Host side:** Application tracing is done over JTAG, so it needs OpenOCD to be set up and running on host machine. For instructions how to set it up, please, see [JTAG Debugging](#) for details.
2. **Target side:** Application tracing functionality can be enabled in menuconfig. *Component config > Application Level Tracing* menu allows selecting destination for the trace data (HW interface for transport). Choosing any of the destinations automatically enables `CONFIG_ESP32_APPTRACE_ENABLE` option.

Note: In order to achieve higher data rates and minimize number of dropped packets it is recommended to optimize setting of JTAG clock frequency, so it is at maximum and still provides stable operation of JTAG, see [Optimize JTAG speed](#).

There are two additional menuconfig options not mentioned above:

1. *Threshold for flushing last trace data to host on panic* (`ESP32_APPTRACE_POSTMORTEM_FLUSH_THRESHOLD`).
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 than 512 bytes of the recent trace data, so if there is less than 512 bytes of pending data at the moment of panic they will not be flushed and will not overwrite previous 16KB. The option is only meaningful in post-mortem mode and when working over JTAG.
2. *Timeout for flushing last trace data to host on panic* (`ESP32_APPTRACE_ONPANIC_HOST_FLUSH_TIMEOUT`).
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 */
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 than 0 then command runs in non-blocking mode. By default 1 ms.

trace_size Maximum size of data to collect (in bytes). Tracing is stopped after specified amount of data is received. By default -1 (trace size stop trigger is disabled).

stop_tmo Idle timeout (in sec). Tracing is stopped if there is no data for specified period of time. By default -1 (disable this stop trigger). Optionally set it to value longer than longest pause between tracing commands from target.

wait4halt If 0 start tracing immediately, otherwise command waits for the target to be halted (after reset, by breakpoint etc.) and then automatically resumes it and starts tracing. By default 0.

skip_size Number of bytes to skip at the start. By default 0.

Note: If `poll_period` is 0, OpenOCD telnet command line will not be available until tracing is stopped. You must stop it manually by resetting the board or pressing Ctrl+C in OpenOCD window (not one with the telnet session). Another option is to set `trace_size` and wait until this size of data is collected. At this point tracing stops automatically.

Command usage examples:

1. Collect 2048 bytes of tracing data to a file “trace.log”. The file will be saved in “openocd-esp32” directory.

```
esp32 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
```

There is an option to configure target to halt after reset on start of scheduler. To do so, go to menuconfig and enable option *Stop program on scheduler start when JTAG/OCD is detected* under *Component config > FreeRTOS*.

Logging to Host

IDF implements useful feature: logging to host via application level tracing library. This is a kind of semihosting when all `ESP_LOGx` calls sends strings to be printed to the host instead of UART. This can be useful because “printing to host” eliminates some steps performed when logging to UART. The most part of work is done on the host.

By default IDF’s logging library uses `vprintf`-like function to write formatted output to dedicated UART. In general it involves the following steps:

1. Format string is parsed to obtain type of each argument.
2. According to its type every argument is converted to string representation.
3. Format string combined with converted arguments is sent to UART.

Though implementation of `vprintf`-like function can be optimised to a certain level, all steps above have to be performed in any case and every step takes some time (especially item 3). So it is frequent situation when addition of extra logging to the program to diagnose some problem changes its behaviour and problem disappears or in the worst cases program can not work normally at all and ends up with an error or even hangs.

Possible ways to overcome this problem are to use higher UART bitrates (or another faster interface) and/or move string formatting procedure to the host.

Application level tracing feature can be used to transfer log information to host using `esp_apptrace_vprintf` function. This function does not perform full parsing of the format string and arguments, instead it just calculates number of arguments passed and sends them along with the format string address to the host. On the host log data are processed and printed out by a special Python script.

Limitations

Current implementation of logging over JTAG has some limitations:

1. Tracing from `ESP_EARLY_LOGx` macros is not supported.
2. No support for printf arguments which size exceeds 4 bytes (e.g. `double` and `uint64_t`).
3. Only strings from .rodata section are supported as format strings and arguments.
4. Maximum number of printf arguments is 256.

How To Use It

In order to use logging via trace module user needs to perform the following steps:

1. On target side special `vprintf`-like function needs to be installed. As it was mentioned earlier this function is `esp_apptrace_vprintf`. It sends log data to the host. Example code is provided in [system/app_trace_to_host](#).
2. Follow instructions in items 2-5 in [Application Specific Tracing](#).
3. To print out collected log records, run the following command in terminal: `$IDF_PATH/tools/esp_app_trace/logtrace_proc.py /path/to/trace/file /path/to/program/elf/file`.

Log Trace Processor Command Options

Command usage:

```
logtrace_proc.py [-h] [--no-errors] <trace_file> <elf_file>
```

Positional arguments:

trace_file Path to log trace file

elf_file Path to program ELF file

Optional arguments:

-h, --help show this help message and exit

--no-errors, -n Do not print errors

System Behaviour Analysis with SEGGER SystemView

Another useful IDF feature built on top of application tracing library is the system level tracing which produces traces compatible with SEGGER SystemView tool (see [SystemView](#)). SEGGER SystemView is a real-time recording and visualization tool that allows to analyze runtime behavior of an application.

Note: Currently IDF-based application is able to generate SystemView compatible traces in form of files to be opened in SystemView application. The tracing process can not yet be controlled using that tool.

How To Use It

Support for this feature is enabled by *Component config > Application Level Tracing > FreeRTOS SystemView Tracing ([SYSVIEW_ENABLE](#))* menuconfig option. There are several other options enabled under the same menu:

1. *ESP32 timer to use as SystemView timestamp source ([SYSVIEW_TS_SOURCE](#))* selects the source of timestamps for SystemView events. In single core mode timestamps are generated using ESP32 internal cycle counter running at maximum 240 Mhz (~4 ns granularity). In dual-core mode external timer working at 40Mhz is used, so timestamp granularity is 25 ns.
2. Individually enabled or disabled collection of SystemView events (CONFIG_SYSVIEW_EVT_XXX):
 - Trace Buffer Overflow Event
 - ISR Enter Event
 - ISR Exit Event
 - ISR Exit to Scheduler Event
 - Task Start Execution Event
 - Task Stop Execution Event
 - Task Start Ready State Event
 - Task Stop Ready State Event
 - Task Create Event
 - Task Terminate Event
 - System Idle Event
 - Timer Enter Event
 - Timer Exit Event

IDF has all the code required to produce SystemView compatible traces, so user can just configure necessary project options (see above), build, download the image to target and use OpenOCD to collect data as described in the previous sections.

OpenOCD SystemView Tracing Command Options

Command usage:

```
esp32 sysview [start <options>] | [stop] | [status]
```

Sub-commands:

start Start tracing (continuous streaming).

stop Stop tracing.

status Get tracing status.

Start command syntax:

```
start <outfile1> [<outfile2>] [<poll_period> [<trace_size> [<stop_tmo>]]]
```

outfile1 Path to file to save data from PRO CPU. This argument should have the following format: `file://path/to/file`.

outfile2 Path to file to save data from APP CPU. This argument should have the following format: `file://path/to/file`.

poll_period Data polling period (in ms) for available trace data. If greater than 0 then command runs in non-blocking mode. By default 1 ms.

trace_size Maximum size of data to collect (in bytes). Tracing is stopped after specified amount of data is received. By default -1 (trace size stop trigger is disabled).

stop_tmo Idle timeout (in sec). Tracing is stopped if there is no data for specified period of time. By default -1 (disable this stop trigger).

Note: If `poll_period` is 0 OpenOCD telnet command line will not be available until tracing is stopped. You must stop it manually by resetting the board or pressing Ctrl+C in OpenOCD window (not one with the telnet session). Another option is to set `trace_size` and wait until this size of data is collected. At this point tracing stops automatically.

Command usage examples:

1. Collect SystemView tracing data to files “pro-cpu.SVDat” and “app-cpu.SVDat”. The files will be saved in “openocd-esp32” directory.

```
esp32 sysview start file://pro-cpu.SVDat file://app-cpu.SVDat
```

The tracing data will be retrieved and saved in non-blocking mode. To stop data this process enter `esp32 apptrace stop` command on OpenOCD telnet prompt, Optionally pressing Ctrl+C in OpenOCD window.

2. Retrieve tracing data and save them indefinitely.

```
esp32 sysview start file://pro-cpu.SVDat file://app-cpu.SVDat 0 -1 -1
```

OpenOCD telnet command line prompt will not be available until tracing is stopped. To stop tracing, press Ctrl+C in OpenOCD window.

Data Visualization

After trace data are collected user can use special tool to visualize the results and inspect behaviour of the program. Unfortunately SystemView does not support tracing from multiple cores. So when tracing from ESP32 working in dual-core mode two files are generated: one for PRO CPU and another one for APP CPU. User can load every file into separate instance of the tool.

It is uneasy and awkward to analyze data for every core in separate instance of the tool. Fortunately there is Eclipse plugin called *Impulse* which can load several trace files and makes 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.

4.8 Partition Tables

4.8.1 Overview

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

Partition table length is 0xC00 bytes (maximum 95 partition table entries). If the partition table is signed due to *secure boot*, the signature is appended after the table data.

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

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

- “Single factory app, no OTA”
- “Factory app, two OTA definitions”

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

4.8.2 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, app, factory, 0x10000, 1M
ota_0, app, ota_0, , 1M
ota_1, app, ota_1, , 1M
```

- There are now three app partition definitions.
- The type of all three are set as “app”, but the subtype varies between the factory app at 0x10000 and the next two “OTA” apps.
- There is also a new “ota data” slot, which holds the data for OTA updates. The bootloader consults this data in order to know which app to execute. If “ota data” is empty, it will execute the factory app.

4.8.3 Creating Custom Tables

If you choose “Custom partition table CSV” in menuconfig then you can also enter the name of a CSV file (in the project directory) to use for your partition table. The CSV file can describe any number of definitions for the table you need.

The CSV format is the same format as printed in the summaries shown above. However, not all fields are required in the CSV. For example, here is the “input” CSV for the OTA partition table:

```
# Name, Type, SubType, Offset, Size
nvs, data, nvs, 0x9000, 0x4000
otadata, data, ota, 0xd000, 0x2000
phy_init, data, phy, 0xf000, 0x1000
factory, app, factory, 0x10000, 1M
ota_0, app, ota_0, , 1M
ota_1, app, ota_1, , 1M
```

- Whitespace between fields is ignored, and so is any line starting with # (comments).
- Each non-comment line in the CSV file is a partition definition.
- Only the offset for the first partition is supplied. The gen_esp32part.py tool fills in each remaining offset to start after the preceding partition.

Name field

Name field can be any meaningful name. It is not significant to the ESP32. Names longer than 16 characters will be truncated.

Type field

Partition type field can be specified as app (0) or data (1). Or it can be a number 0-254 (or as hex 0x00-0xFE). Types 0x00-0x3F are reserved for esp-idf core functions.

If your application needs to store data, please add a custom partition type in the range 0x40-0xFE.

The bootloader ignores any partition types other than app (0) & data (1).

Subtype

The 8-bit subtype field is specific to a given partition type.

esp-idf currently only specifies the meaning of the subtype field for “app” and “data” partition types.

App Subtypes

When type is “app”, the subtype field can be specified as factory (0), ota_0 (0x10) … ota_15 (0x1F) or test (0x20).

- factory (0) is the default app partition. The bootloader will execute the factory app unless there it sees a partition of type data/ota, in which case it reads this partition to determine which OTA image to boot.
 - OTA never updates the factory partition.
 - If you want to conserve flash usage in an OTA project, you can remove the factory partition and use ota_0 instead.
- ota_0 (0x10) … ota_15 (0x1F) are the OTA app slots. Refer to the [OTA documentation](#) for more details, which then use the OTA data partition to configure which app slot the bootloader should boot. If using OTA, an application should have at least two OTA application slots (ota_0 & ota_1). Refer to the [OTA documentation](#) for more details.
- test (0x2) is a reserved subtype for factory test procedures. It is not currently supported by the esp-idf bootloader.

Data Subtypes

When type is “data”, the subtype field can be specified as ota (0), phy (1), nvs (2).

- ota (0) is the [OTA data partition](#) which stores information about the currently selected OTA application. This partition should be 0x2000 bytes in size. Refer to the [OTA documentation](#) for more details.
- phy (1) is for storing PHY initialisation data. This allows PHY to be configured per-device, instead of in firmware.
 - In the default configuration, the phy partition is not used and PHY initialisation data is compiled into the app itself. As such, this partition can be removed from the partition table to save space.
 - To load PHY data from this partition, run `make menuconfig` and enable [`ESP32_PHY_INIT_DATA_IN_PARTITION`](#) option. You will also need to flash your devices with phy init data as the esp-idf build system does not do this automatically.
- nvs (2) is for the [Non-Volatile Storage \(NVS\) API](#).
 - NVS is used to store per-device PHY calibration data (different to initialisation data).
 - NVS is used to store WiFi data if the [`esp_wifi_set_storage\(WIFI_STORAGE_FLASH\)`](#) initialisation function is used.
 - The NVS API can also be used for other application data.

- It is strongly recommended that you include an NVS partition of at least 0x3000 bytes in your project.
- If using NVS API to store a lot of data, increase the NVS partition size from the default 0x6000 bytes.

Other data subtypes are reserved for future esp-idf uses.

Offset & Size

Only the first offset field is required (we recommend using 0x10000). Partitions with blank offsets will start after the previous partition.

App partitions have to be at offsets aligned to 0x10000 (64K). If you leave the offset field blank, the tool will automatically align the partition. If you specify an unaligned offset for an app partition, the tool will return an error.

Sizes and offsets can be specified as decimal numbers, hex numbers with the prefix 0x, or size multipliers K or M (1024 and 1024*1024 bytes).

4.8.4 Generating Binary Partition Table

The partition table which is flashed to the ESP32 is in a binary format, not CSV. The tool `partition_table/gen_esp32part.py` is used to convert between CSV and binary formats.

If you configure the partition table CSV name in `make menuconfig` and then `make partition_table`, this conversion is done as part of the build process.

To convert CSV to Binary manually:

```
python gen_esp32part.py --verify input_partitions.csv binary_partitions.bin
```

To convert binary format back to CSV:

```
python gen_esp32part.py --verify binary_partitions.bin input_partitions.csv
```

To display the contents of a binary partition table on stdout (this is how the summaries displayed when running `make partition_table` are generated):

```
python gen_esp32part.py binary_partitions.bin
```

`gen_esp32part.py` takes one optional argument, `--verify`, which will also verify the partition table during conversion (checking for overlapping partitions, unaligned partitions, etc.)

4.8.5 Flashing the partition table

- `make partition_table-flash`: will flash the partition table with `esptool.py`.
- `make flash`: Will flash everything including the partition table.

A manual flashing command is also printed as part of `make partition_table`.

Note that updating the partition table doesn't erase data that may have been stored according to the old partition table. You can use `make erase_flash` (or `esptool.py erase_flash`) to erase the entire flash contents.

4.9 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.

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.

4.9.1 Background

- Most data is stored in flash. Flash access does not need to be protected from physical access in order for secure boot to function, because critical data is stored (non-software-accessible) in Efuses internal to the chip.
- Efuses are used to store the secure bootloader key (in efuse BLOCK2), and also a single Efuse bit (ABS_DONE_0) is burned (written to 1) to permanently enable secure boot on the chip. For more details about efuse, see Chapter 11 “eFuse Controller” in the Technical Reference Manual.
- To understand the secure boot process, first familiarise yourself with the standard [ESP-IDF boot process](#).
- Both stages of the boot process (initial software bootloader load, and subsequent partition & app loading) are verified by the secure boot process, in a “chain of trust” relationship.

4.9.2 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](#).

- When running in secure boot mode, the software bootloader uses the secure boot signing key (the public key of which is embedded in the bootloader itself, and therefore validated as part of the bootloader) to verify the signature appended to all subsequent partition tables and app images before they are booted.

4.9.3 Keys

The following keys are used by the secure boot process:

- “secure bootloader key” is a 256-bit AES key that is stored in Efuse block 2. The bootloader can generate this key itself from the internal hardware random number generator, the user does not need to supply it (it is optionally possible to supply this key, see [Re-Flashable Software Bootloader](#)). The Efuse holding this key is read & write protected (preventing software access) before secure boot is enabled.
- “secure boot signing key” is a standard ECDSA public/private key pair (see [Image Signing Algorithm](#)) in PEM format.
 - The public key from this key pair (for signature verification but not signature creation) is compiled into the software bootloader and used to verify the second stage of booting (partition table, app image) before booting continues. The public key can be freely distributed, it does not need to be kept secret.
 - The private key from this key pair *must be securely kept private*, as anyone who has this key can authenticate to any bootloader that is configured with secure boot and the matching public key.

4.9.4 How To Enable Secure Boot

- 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](#).)
- 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.
- 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
- 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.

- 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`.
- 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!**
- Run `make flash` to build and flash the partition table and the just-built app image. The app image will be signed using the signing key you generated in step 4.

NOTE: `make flash` doesn’t flash the bootloader if secure boot is enabled.

8. Reset the ESP32 and it will boot the software bootloader you flashed. The software bootloader will enable secure boot on the chip, and then it verifies the app image signature and boots the app. You should watch the serial console output from the ESP32 to verify that secure boot is enabled and no errors have occurred due to the build configuration.

NOTE Secure boot won't be enabled until after a valid partition table and app image have been flashed. This is to prevent accidents before the system is fully configured.

9. On subsequent boots, the secure boot hardware will verify the software bootloader has not changed (using the secure bootloader key) and then the software bootloader will verify the signed partition table and app image (using the public key portion of the secure boot signing key).

4.9.5 Re-Flashable Software Bootloader

Configuration “Secure Boot: One-Time Flash” is the recommended configuration for production devices. In this mode, each device gets a unique key that is never stored outside the device.

However, an alternative mode “Secure Boot: Reflashable” is also available. This mode allows you to supply a 256-bit key file that is used for the secure bootloader key. As you have the key file, you can generate new bootloader images and secure boot digests for them.

In the esp-idf build process, this 256-bit key file is derived from the app signing key generated during the `generate_signing_key` step above. The private key's SHA-256 digest is used as the 256-bit secure bootloader key. This is a convenience so you only need to generate/protect a single private key.

NOTE: Although it's possible, we strongly recommend not generating one secure boot key and flashing it to every device in a production environment. The “One-Time Flash” option is recommended for production environments.

To enable a reflashable bootloader:

1. In the `make menuconfig` step, select “Bootloader Config” -> “Secure Boot” -> “Reflashable”.
2. Follow the steps shown above to choose a signing key file, and generate the key file.
3. Run `make bootloader`. A 256-bit key file will be created, derived from the private key that is used for signing. Two sets of flashing steps will be printed - the first set of steps includes an `espefuse.py burn_key` command which is used to write the bootloader key to efuse. (Flashing this key is a one-time-only process.) The second set of steps can be used to reflash the bootloader with a pre-calculated digest (generated during the build process).
4. Resume from [Step 6 of the one-time flashing process](#), to flash the bootloader and enable secure boot. Watch the console log output closely to ensure there were no errors in the secure boot configuration.

4.9.6 Generating Secure Boot Signing Key

The build system will prompt you with a command to generate a new signing key via `espsecure.py generate_signing_key`. This uses the `python-ecdsa` library, which in turn uses Python's `os.urandom()` as a random number source.

The strength of the signing key is proportional to (a) the random number source of the system, and (b) the correctness of the algorithm used. For production devices, we recommend generating signing keys from a system with a quality entropy source, and using the best available EC key generation utilities.

For example, to generate a signing key using the `openssl` command line:

```
` openssl ecparam -name prime256v1 -genkey -noout -out my_secure_boot_signing_key.pem`
```

Remember that the strength of the secure boot system depends on keeping the signing key private.

4.9.7 Remote Signing of Images

For production builds, it can be good practice to use a remote signing server rather than have the signing key on the build machine (which is the default esp-idf secure boot configuration). The espsecure.py command line program can be used to sign app images & partition table data for secure boot, on a remote system.

To use remote signing, disable the option “Sign binaries during build”. The private signing key does not need to be present on the build system. However, the public (signature verification) key is required because it is compiled into the bootloader (and can be used to verify image signatures during OTA updates).

To extract the public key from the private key:

```
espsecure.py extract_public_key --keyfile PRIVATE_SIGNING_KEY PUBLIC_VERIFICATION_KEY
```

The path to the public signature verification key needs to be specified in the menuconfig under “Secure boot public signature verification key” in order to build the secure bootloader.

After the app image and partition table are built, the build system will print signing steps using espsecure.py:

```
espsecure.py sign_data --keyfile PRIVATE_SIGNING_KEY BINARY_FILE
```

The above command appends the image signature to the existing binary. You can use the *–output* argument to write the signed binary to a separate file:

```
espsecure.py sign_data --keyfile PRIVATE_SIGNING_KEY --output SIGNED_BINARY_FILE  
→ BINARY_FILE
```

4.9.8 Secure Boot Best Practices

- Generate the signing key on a system with a quality source of entropy.
- Keep the signing key private at all times. A leak of this key will compromise the secure boot system.
- Do not allow any third party to observe any aspects of the key generation or signing process using espsecure.py. Both processes are vulnerable to timing or other side-channel attacks.
- Enable all secure boot options in the Secure Boot Configuration. These include flash encryption, disabling of JTAG, disabling BASIC ROM interpreter, and disabling the UART bootloader encrypted flash access.
- Use secure boot in combination with *flash encryption* to prevent local readout of the flash contents.

4.9.9 Technical Details

The following sections contain low-level reference descriptions of various secure boot elements:

Secure Boot Hardware Support

The first stage of secure boot verification (checking the software bootloader) is done via hardware. The ESP32’s Secure Boot support hardware can perform three basic operations:

1. Generate a random sequence of bytes from a hardware random number generator.
2. Generate a digest from data (usually the bootloader image from flash) using a key stored in Efuse block 2. The key in Efuse can (& should) be read/write protected, which prevents software access. For full details of this algorithm see *Secure Bootloader Digest Algorithm*. The digest can only be read back by software if Efuse ABS_DONE_0 is *not* burned (ie still 0).

3. Generate a digest from data (usually the bootloader image from flash) using the same algorithm as step 2 and compare it to a pre-calculated digest supplied in a buffer (usually read from flash offset 0x0). The hardware returns a true/false comparison without making the digest available to software. This function is available even when Efuse ABS_DONE_0 is burned.

Secure Bootloader Digest Algorithm

Starting with an “image” of binary data as input, this algorithm generates a digest as output. The digest is sometimes referred to as an “abstract” in hardware documentation.

For a Python version of this algorithm, see the `espsecure.py` tool in the `components/esptool_py` directory (specifically, the `digest_secure_bootloader` command).

Items marked with (^) are to fulfill hardware restrictions, as opposed to cryptographic restrictions.

1. Prefix the image with a 128 byte randomly generated IV.
2. If the image length is not modulo 128, pad the image to a 128 byte boundary with 0xFF. (^)
3. For each 16 byte plaintext block of the input image:
 - Reverse the byte order of the plaintext input block (^)
 - Apply AES256 in ECB mode to the plaintext block.
 - Reverse the byte order of the ciphertext output block. (^)
 - Append to the overall ciphertext output.
4. Byte-swap each 4 byte word of the ciphertext (^)
5. Calculate SHA-512 of the ciphertext.

Output digest is 192 bytes of data: The 128 byte IV, followed by the 64 byte SHA-512 digest.

Image Signing Algorithm

Deterministic ECDSA as specified by [RFC 6979](#).

- Curve is NIST256p (openssl calls this curve “prime256v1”, it is also sometimes called secp256r1).
- Hash function is SHA256.
- Key format used for storage is PEM.
 - In the bootloader, the public key (for signature verification) is flashed as 64 raw bytes.
- Image signature is 68 bytes - a 4 byte version word (currently zero), followed by a 64 bytes of signature data. These 68 bytes are appended to an app image or partition table data.

Manual Commands

Secure boot is integrated into the esp-idf build system, so `make` will automatically sign an app image if secure boot is enabled. `make bootloader` will produce a bootloader digest if menuconfig is configured for it.

However, it is possible to use the `espsecure.py` tool to make standalone signatures and digests.

To sign a binary image:

```
espsecure.py sign_data --keyfile ./my_signing_key.pem --output ./image_signed.bin  
→image-unsigned.bin
```

Keyfile is the PEM file containing an ECDSA private signing key.

To generate a bootloader digest:

```
espsecure.py digest_secure_bootloader --keyfile ./securebootkey.bin --output ./  
bootloader-digest.bin build/bootloader/bootloader.bin
```

Keyfile is the 32 byte raw secure boot key for the device. To flash this digest onto the device:

```
esptool.py write_flash 0x0 bootloader-digest.bin
```

4.10 ULP coprocessor programming

4.10.1 ULP coprocessor instruction set

This document provides details about the instructions used by ESP32 ULP coprocessor assembler.

ULP coprocessor has 4 16-bit general purpose registers, labeled R0, R1, R2, R3. It also has an 8-bit counter register (stage_cnt) which can be used to implement loops. Stage count register is accessed using special instructions.

ULP coprocessor can access 8k bytes of RTC_SLOW_MEM memory region. Memory is addressed in 32-bit word units. It can also access peripheral registers in RTC_CNTL, RTC_IO, and SENS peripherals.

All instructions are 32-bit. Jump instructions, ALU instructions, peripheral register and memory access instructions are executed in 1 cycle. Instructions which work with peripherals (TSENS, ADC, I2C) take variable number of cycles, depending on peripheral operation.

The instruction syntax is case insensitive. Upper and lower case letters can be used and intermixed arbitrarily. This is true both for register names and instruction names.

Note about addressing

ESP32 ULP coprocessor's JUMP, ST, LD instructions which take register as an argument (jump address, store/load base address) expect the argument to be expressed in 32-bit words.

Consider the following example program:

```
entry:  
    NOP  
    NOP  
    NOP  
    NOP  
loop:  
    MOVE R1, loop  
    JUMP R1
```

When this program is assembled and linked, address of label *loop* will be equal to 16 (expressed in bytes). However *JUMP* instruction expects the address stored in register to be expressed in 32-bit words. To account for this common use case, assembler will convert the address of label *loop* from bytes to words, when generating *MOVE* instruction, so the code generated code will be equivalent to:

0000	NOP
0004	NOP
0008	NOP
000c	NOP
0010	MOVE R1, 4
0014	JUMP R1

The other case is when the argument of MOVE instruction is not a label but a constant. In this case assembler will use the value as is, without any conversion:

```
.set      val, 0x10
MOVE     R1, val
```

In this case, value loaded into R1 will be 0x10.

Similar considerations apply to LD and ST instructions. Consider the following code:

```
.global array
array: .long 0
       .long 0
       .long 0
       .long 0

MOVE R1, array
MOVE R2, 0x1234
ST R2, R1, 0          // write value of R2 into the first array element,
                      // i.e. array[0]

ST R2, R1, 4          // write value of R2 into the second array element
                      // (4 byte offset), i.e. array[1]

ADD R1, R1, 2          // this increments address by 2 words (8 bytes)
ST R2, R1, 0          // write value of R2 into the third array element,
                      // i.e. array[2]
```

NOP - no operation

Syntax: NOP

Operands: None

Description: No operation is performed. Only the PC is incremented.

Example:

```
1: NOP
```

ADD - Add to register

Syntax: ADD Rd_{dst}, R_{src1}, R_{src2}

ADD Rd_{dst}, R_{src1}, imm

Operands:

- Rd_{dst} - Register R[0..3]
- R_{src1} - Register R[0..3]
- R_{src2} - Register R[0..3]
- Imm - 16-bit signed value

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

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

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.

Example:

```

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

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

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

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

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

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

Description The instruction loads lower 16-bit half-word from memory with address *Rsrc*+*offset* into the destination register *Rdst*:

```
Rdst [15:0] = Mem[Rsrc + offset / 4][15:0]
```

Examples:

```
1:      LD   R1, R2, 0x12          //R1 = MEM[R2+0x12]

2:      .data                  //Data section definition
Addr1: .word    123           // Define label Addr1 16 bit
      .set     offs, 0x00        // Define constant offs
      .text                //Text section definition
      MOVE     R1, 1            // R1 = 1
      MOVE     R2, Addr1        // R2 = Addr1 / 4 (address of label is_
→converted into words)
      LD       R1, R2, offs      // R1 = MEM[R2 + 0]
                                // R1 will be 123
```

JUMP – Jump to an absolute address

Syntax JUMP *Rdst*

JUMP *ImmAddr*

JUMP *Rdst, Condition*

JUMP *ImmAddr; Condition*

Operands

- *Rdst* – Register R[0..3] containing address to jump to (expressed in 32-bit words)
- *ImmAddr* – 13 bits address (expressed in bytes), aligned to 4 bytes
- **Condition:**
 - EQ – jump if last ALU operation result was zero
 - OV – jump if last ALU has set overflow flag

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 \geq threshold
 - *LT* (less than) – jump if value in R0 $<$ threshold

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  $\geq$  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  $\geq$  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
 - *GT* (greater than) – jump if value in stage_cnt > threshold

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

Description The instruction sets the stage count register to 0

Examples:

```
1:      STAGE_RST      // Reset stage count register
```

STAGE_INC – Increment stage count register

Syntax STAGE_INC *Value*

Operands

- *Value* – 8 bits value

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

Description The instruction decrements stage count register by given value.

Examples:

```

1:      STAGE_DEC      10          // stage_cnt -= 10;

2:          // Down counting loop example
        STAGE_RST           // set stage_cnt to 0
        STAGE_INC  16         // increment stage_cnt to 16

```

```

label: STAGE_DEC 1           // stage_cnt--;
    NOP                   // do something
    JUMPS     label, 0, GT // jump to label if stage_cnt > 0

```

HALT – End the program

Syntax HALT

Operands No operands

Description The instruction halt the processor to the power down mode

Examples:

```

1:      HALT      // Move chip to powerdown

```

WAKE – wakeup the chip

Syntax WAKE

Operands No operands

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.

Examples:

```

1:      WAKE          // Trigger wake up
    REG_WR 0x006, 24, 24, 0 // Stop ULP timer (clear RTC_CNTL_ULP_CP_SLP_
    ↵TIMER_EN)
    HALT          // Stop the ULP program
    // After these instructions, SoC will wake up,
    // and ULP will not run again until started by the main program.

```

SLEEP – set ULP wakeup timer period

Syntax SLEEP *sleep_reg*

Operands

- *sleep_reg* – 0..4, selects one of SENS_ULP_CP_SLEEP_CYCx_REG registers.

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

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

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

Description The instruction makes measurements from ADC.

Examples:

```
1:      ADC      R1,  0,  1    // Measure value using ADC1 pad 2 and store result
                                         into R1
```

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

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, and SENS peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

```
addr_ulp = (addr_dport - DR_REG_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

Description The instruction writes up to 8 bits from a general purpose register into a peripheral register. REG[*Addr*][*High*:*Low*] = *data*

This instruction can access registers in RTC_CNTL, RTC_IO, and SENS peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

```
addr_ulp = (addr_dport - DR_REG_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_ctrl_reg.h`, `soc/rtc_io_reg.h`, and `soc/sens_reg.h`.

READ_RTC_REG(rtc_reg, low_bit, bit_width) Read up to 16 bits from rtc_reg[low_bit + bit_width - 1 : low_bit] into R0. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_ctrl_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_TSSENS_OUT field of SENS_SAR_SLAVE_ADDR3_REG into R0 */
READ_RTC_FIELD(SENS_SAR_SLAVE_ADDR3_REG, SENS_TSSENS_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_ctrl_reg.h"

/* Set RTC_CNTL_ULP_CP_SLP_TIMER_EN field of RTC_CNTL_STATE0_REG to 0 */
WRITE_RTC_FIELD(RTC_CNTL_STATE0_REG, RTC_CNTL_ULP_CP_SLP_TIMER_EN, 0)
```

4.10.2 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.

Load and store instructions use addresses expressed in 32-bit words. Address 0 corresponds to the first word of RTC_SLOW_MEM (which is address 0x50000000 as seen by the main CPUs).

To generate branch instructions, special M_ preprocessor defines are used. M_LABEL define can be used to define a branch target. Label identifier is a 16-bit integer. M_Bxxx defines can be used to generate branch instructions with target set to a particular label.

Implementation note: these M_ preprocessor defines will be translated into two ulp_insn_t values: one is a token value which contains label number, and the other is the actual instruction. ulp_process_macros_and_load function resolves the label number to the address, modifies the branch instruction to use the correct address, and removes the the extra ulp_insn_t token which contains the label numer.

Here is an example of using labels and branches:

```
const ulp_insn_t program[] = {
    I_MOVI(R0, 34),           // R0 <- 34
    M_LABEL(1),                // label_1
    I_MOVI(R1, 32),           // R1 <- 32
    I_LD(R1, R1, 0),          // R1 <- RTC_SLOW_MEM[R1]
    I_MOVI(R2, 33),           // R2 <- 33
    I_LD(R2, R2, 0),          // R2 <- RTC_SLOW_MEM[R2]
    I_SUBR(R3, R1, R2),       // R3 <- R1 - R2
    I_ST(R3, R0, 0),          // R3 -> RTC_SLOW_MEM[R0 + 0]
    I_ADDI(R0, R0, 1),         // R0++
    M_BL(1, 64),              // if (R0 < 64) goto label_1
    I_HALT(),
};

RTC_SLOW_MEM[32] = 42;
RTC_SLOW_MEM[33] = 18;
size_t load_addr = 0;
size_t size = sizeof(program)/sizeof(ulp_insn_t);
ulp_process_macros_and_load(load_addr, program, &size);
ulp_run(load_addr);
```

Functions

`esp_err_t ulp_process_macros_and_load(uint32_t load_addr, const ulp_insn_t *program, size_t *psize)`

Resolve all macro references in a program and load it into RTC memory.

Return

- ESP_OK on success
- ESP_ERR_NO_MEM if auxiliary temporary structure can not be allocated
- one of ESP_ERR_ULP_xxx if program is not valid or can not be loaded

Parameters

- `load_addr`: address where the program should be loaded, expressed in 32-bit words
- `program`: `ulp_insn_t` array with the program
- `psize`: size of the program, expressed in 32-bit words

`esp_err_t ulp_run(uint32_t entry_point)`

Run the program loaded into RTC memory.

Return ESP_OK on success

Parameters

- `entry_point`: entry point, expressed in 32-bit words

Error codes

`ESP_ERR_ULP_BASE`

Offset for ULP-related error codes

`ESP_ERR_ULP_SIZE_TOO_BIG`

Program doesn't fit into RTC memory reserved for the ULP

`ESP_ERR_ULP_INVALID_LOAD_ADDR`

Load address is outside of RTC memory reserved for the ULP

`ESP_ERR_ULP_DUPLICATE_LABEL`

More than one label with the same number was defined

`ESP_ERR_ULP_UNDEFINED_LABEL`

Branch instructions references an undefined label

`ESP_ERR_ULP_BRANCH_OUT_OF_RANGE`

Branch target is out of range of B instruction (try replacing with BX)

ULP coprocessor registers

ULP co-processor has 4 16-bit general purpose registers. All registers have same functionality, with one exception. R0 register is used by some of the compare-and-branch instructions as a source register.

These definitions can be used for all instructions which require a register.

`R0`

general purpose register 0

`R1`

general purpose register 1

`R2`

general purpose register 2

`R3`

general purpose register 3

ULP coprocessor instruction defines

`I_DELAY(cycles_)`

Delay (nop) for a given number of cycles

`I_HALT`

Halt the coprocessor.

This instruction halts the coprocessor, but keeps ULP timer active. As such, ULP program will be restarted again by timer. To stop the program and prevent the timer from restarting the program, use `I_END(0)` instruction.

`I_END`

Stop ULP program timer.

This is a convenience macro which disables the ULP program timer. Once this instruction is used, ULP program will not be restarted anymore until `ulp_run` function is called.

ULP program will continue running after this instruction. To stop the currently running program, use `I_HALT()`.

I_ST (reg_val, reg_addr, offset_)

Store value from register `reg_val` into RTC memory.

The value is written to an offset calculated by adding value of `reg_addr` register and `offset_` field (this offset is expressed in 32-bit words). 32 bits written to RTC memory are built as follows:

- bits [31:21] hold the PC of current instruction, expressed in 32-bit words
- bits [20:16] = 5'b1
- bits [15:0] are assigned the contents of `reg_val`

`RTC_SLOW_MEM[addr + offset_] = { 5'b0, insn_PC[10:0], val[15:0] }`

I_LD (reg_dest, reg_addr, offset_)

Load value from RTC memory into `reg_dest` register.

Loads 16 LSBs from RTC memory word given by the sum of value in `reg_addr` and value of `offset_`.

I_WR_REG (reg, low_bit, high_bit, val)

Write literal value to a peripheral register

`reg[high_bit : low_bit] = val` This instruction can access RTC_CNTL_, RTC_IO_, and SENS_ peripheral registers.

I_RD_REG (reg, low_bit, high_bit)

Read from peripheral register into R0

R0 = `reg[high_bit : low_bit]` This instruction can access RTC_CNTL_, RTC_IO_, and SENS_ peripheral registers.

I_BL (pc_offset, imm_value)

Branch relative if R0 less than immediate value.

`pc_offset` is expressed in words, and can be from -127 to 127 `imm_value` is a 16-bit value to compare R0 against

I_BGE (pc_offset, imm_value)

Branch relative if R0 greater or equal than immediate value.

`pc_offset` is expressed in words, and can be from -127 to 127 `imm_value` is a 16-bit value to compare R0 against

I_BXR (reg_pc)

Unconditional branch to absolute PC, address in register.

`reg_pc` is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXI (imm_pc)

Unconditional branch to absolute PC, immediate address.

Address `imm_pc` is expressed in 32-bit words.

I_BXZR (reg_pc)

Branch to absolute PC if ALU result is zero, address in register.

`reg_pc` is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXZI (imm_pc)

Branch to absolute PC if ALU result is zero, immediate address.

Address `imm_pc` is expressed in 32-bit words.

I_BXFR (reg_pc)

Branch to absolute PC if ALU overflow, address in register

`reg_pc` is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXFI (imm_pc)

Branch to absolute PC if ALU overflow, immediate address

Address imm_pc is expressed in 32-bit words.

I_ADDR (reg_dest, reg_src1, reg_src2)

Addition: dest = src1 + src2

I_SUBR (reg_dest, reg_src1, reg_src2)

Subtraction: dest = src1 - src2

I_ANDR (reg_dest, reg_src1, reg_src2)

Logical AND: dest = src1 & src2

I_ORR (reg_dest, reg_src1, reg_src2)

Logical OR: dest = src1 | src2

I_MOVR (reg_dest, reg_src)

Copy: dest = src

I_LSHR (reg_dest, reg_src, reg_shift)

Logical shift left: dest = src << shift

I_RSHR (reg_dest, reg_src, reg_shift)

Logical shift right: dest = src >> shift

I_ADDI (reg_dest, reg_src, imm_)

Add register and an immediate value: dest = src1 + imm

I_SUBI (reg_dest, reg_src, imm_)

Subtract register and an immediate value: dest = src - imm

I_ANDI (reg_dest, reg_src, imm_)

Logical AND register and an immediate value: dest = src & imm

I_ORI (reg_dest, reg_src, imm_)

Logical OR register and an immediate value: dest = src | imm

I_MOVI (reg_dest, imm_)

Copy an immediate value into register: dest = imm

I_LSHI (reg_dest, reg_src, imm_)

Logical shift left register value by an immediate: dest = src << imm

I_RSHI (reg_dest, reg_src, imm_)

Logical shift right register value by an immediate: dest = val >> imm

M_LABEL (label_num)

Define a label with number label_num.

This is a macro which doesn't generate a real instruction. The token generated by this macro is removed by ulp_process_macros_and_load function. Label defined using this macro can be used in branch macros defined below.

M_BL (label_num, imm_value)

Macro: branch to label label_num if R0 is less than immediate value.

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BGE (label_num, imm_value)

Macro: branch to label label_num if R0 is greater or equal than immediate value

This macro generates two `ulp_insn_t` values separated by a comma, and should be used when defining contents of `ulp_insn_t` arrays. First value is not a real instruction; it is a token which is removed by `ulp_process_macros_and_load` function.

M_BX (label_num)

Macro: unconditional branch to label

This macro generates two `ulp_insn_t` values separated by a comma, and should be used when defining contents of `ulp_insn_t` arrays. First value is not a real instruction; it is a token which is removed by `ulp_process_macros_and_load` function.

M_BXZ (label_num)

Macro: branch to label if ALU result is zero

This macro generates two `ulp_insn_t` values separated by a comma, and should be used when defining contents of `ulp_insn_t` arrays. First value is not a real instruction; it is a token which is removed by `ulp_process_macros_and_load` function.

M_BXF (label_num)

Macro: branch to label if ALU overflow

This macro generates two `ulp_insn_t` values separated by a comma, and should be used when defining contents of `ulp_insn_t` arrays. First value is not a real instruction; it is a token which is removed by `ulp_process_macros_and_load` function.

Defines

RTC_SLOW_MEM

RTC slow memory, 8k size

ULP (Ultra Low Power) coprocessor is a simple FSM which is designed to perform measurements using ADC, temperature sensor, and external I2C sensors, while main processors are in deep sleep mode. ULP coprocessor can access `RTC_SLOW_MEM` memory region, and registers in `RTC_CNTL`, `RTC_IO`, and `SARADC` peripherals. ULP coprocessor uses fixed-width 32-bit instructions, 32-bit memory addressing, and has 4 general purpose 16-bit registers.

4.10.3 Installing the toolchain

ULP coprocessor code is written in assembly and compiled using the `binutils-esp32ulp` toolchain.

1. Download the toolchain using the links listed on this page: <https://github.com/espressif/binutils-esp32ulp/wiki#downloads>
2. Extract the toolchain into a directory, and add the path to the `bin/` directory of the toolchain to the `PATH` environment variable.

4.10.4 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:

- (a) **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).
- (b) **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.
- (c) **Run linker script template through C preprocessor.** The template is located in components/ulp/lld directory.
- (d) **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.
- (e) **Dump contents of the ELF file into binary (ulp_app_name.bin)** for embedding into the application.
- (f) **Generate list of global symbols (ulp_app_name.sym)** in the ELF file using esp32ulp-elf-nm.
- (g) **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.
- (h) **Add the generated binary to the list of binary files** to be embedded into the application.

4.10.5 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. These files include each global symbol defined in the ULP program, prefixed with `ulp_`.

The header file contains declaration of the symbol:

```
extern uint32_t ulp_measurement_count;
```

Note that all symbols (variables, arrays, functions) are declared as `uint32_t`. For functions and arrays, take address of the symbol and cast to the appropriate type.

The generated linker script file defines locations of symbols in `RTC_SLOW_MEM`:

```
PROVIDE ( ulp_measurement_count = 0x50000060 );
```

To access ULP program variables from the main program, include the generated header file and use variables as one normally would:

```
#include "ulp_app_name.h"

// later
void init_ulp_vars() {
    ulp_measurement_count = 64;
}
```

Note that ULP program can only use lower 16 bits of each 32-bit word in RTC memory, because the registers are 16-bit, and there is no instruction to load from high part of the word.

Likewise, ULP store instruction writes register value into the lower 16 bit part of the 32-bit word. Upper 16 bits are written with a value which depends on the address of the store instruction, so when reading variables written by the ULP, main application needs to mask upper 16 bits, e.g.:

```
printf("Last measurement value: %d\n", ulp_last_measurement & UINT16_MAX);
```

4.10.6 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 - 16) 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) / sizeof(uint32_t)) );
```

`esp_err_t ulp_run(uint32_t entry_point)`
 Run the program loaded into RTC memory.

Return `ESP_OK` on success

Parameters

- `entry_point`: entry point, expressed in 32-bit words

Declaration of the entry point symbol comes from the above mentioned generated header file, `$ (ULP_APP_NAME) .h`. In assembly source of the ULP application, this symbol must be marked as `.global`:

```
.global entry
entry:
/* code starts here */
```

4.10.7 ULP program flow

ULP coprocessor is started by a timer. The timer is started once `ulp_run` is called. The timer counts a number of RTC_SLOW_CLK ticks (by default, produced by an internal 150kHz RC oscillator). The number of ticks is set using `SENS_ULP_CP_SLEEP_CYCx_REG` registers ($x = 0..4$). When starting the ULP for the first time, `SENS_ULP_CP_SLEEP_CYC0_REG` will be used to set the number of timer ticks. Later the ULP program can select another `SENS_ULP_CP_SLEEP_CYCx_REG` register using `sleep` instruction.

The application can set ULP timer period values (`SENS_ULP_CP_SLEEP_CYCx_REG`, $x = 0..4$) using `ulp_wakeup_period_set` function.

`esp_err_t ulp_set_wakeup_period(size_t period_index, uint32_t period_us)`

Set one of ULP wakeup period values.

ULP coprocessor starts running the program when the wakeup timer counts up to a given value (called period). There are 5 period values which can be programmed into `SENS_ULP_CP_SLEEP_CYCx_REG` registers, $x = 0..4$. By default, wakeup timer will use the period set into `SENS_ULP_CP_SLEEP_CYC0_REG`, i.e. period number 0. ULP program code can use `SLEEP` instruction to select which of the `SENS_ULP_CP_SLEEP_CYCx_REG` should be used for subsequent wakeups.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if `period_index` is out of range

Parameters

- `period_index`: wakeup period setting number (0 - 4)
- `period_us`: wakeup period, us

Once the timer counts the number of ticks set in the selected `SENS_ULP_CP_SLEEP_CYCx_REG` register, ULP coprocessor powers up and starts running the program from the entry point set in the call to `ulp_run`.

The program runs until it encounters a `halt` instruction or an illegal instruction. Once the program halts, ULP coprocessor powers down, and the timer is started again.

To disable the timer (effectively preventing the ULP program from running again), clear the `RTC_CNTL_ULP_CP_SLP_TIMER_EN` bit in the `RTC_CNTL_STATE0_REG` register. This can be done both from ULP code and from the main program.

4.11 Unit Testing in ESP32

ESP-IDF comes with a unit test app based on Unity - unit test framework. Unit tests are integrated in the ESP-IDF repository and are placed in `test` subdirectory of each component respectively.

4.11.1 Adding unit tests

Unit tests are added in the `test` subdirectory of the respective component. Tests are added in C files, a single C file can include multiple test cases. Test files start with the word “test”.

The test file should include `unity.h` and the header for the C module to be tested.

Tests are added in a function in the C file as follows:

```
TEST_CASE("test name", "[module name]"
{
    // Add test here
}
```

First argument is a descriptive name for the test, second argument is an identifier in square brackets. Identifiers are used to group related test, or tests with specific properties.

There is no need to add a main function with `UNITY_BEGIN()` and `UNITY_END()` in each test case. `unity_platform.c` will run `UNITY_BEGIN()`, run the tests cases, and then call `UNITY_END()`.

Each *test* subdirectory needs to include component.mk file with at least the following line of code:

```
COMPONENT_ADD_LDFLAGS = -Wl,--whole-archive -l$(COMPONENT_NAME) -Wl,--no-whole-archive
```

See <http://www.throwtheswitch.org/unity> for more information about writing tests in Unity.

4.11.2 Building unit test app

Follow the setup instructions in the top-level esp-idf README. Make sure that `IDF_PATH` environment variable is set to point to the path of esp-idf top-level directory.

Change into tools/unit-test-app directory to configure and build it:

- `make menuconfig` - configure unit test app.
- `make TESTS_ALL=1` - build unit test app with tests for each component having tests in the `test` subdirectory.
- `make TEST_COMPONENTS='xxx'` - build unit test app with tests for specific components.

When the build finishes, it will print instructions for flashing the chip. You can simply run `make flash` to flash all build output.

You can also run `make flash TESTS_ALL=1` or `make TEST_COMPONENTS='xxx'` to build and flash. Everything needed will be rebuilt automatically before flashing.

Use menuconfig to set the serial port for flashing.

4.11.3 Running unit tests

After flashing reset the ESP32 and it will boot the unit test app.

Unit test app prints a test menu with all available tests.

Test cases can be run by inputting one of the following:

- Test case name in quotation marks to run a single test case
- Test case index to run a single test case
- Module name in square brackets to run all test cases for a specific module
- An asterisk to run all test cases

4.12 Console

ESP-IDF provides `console` component, which includes building blocks needed to develop an interactive console over serial port. This component includes following facilities:

- Line editing, provided by `linenoise` library. This includes handling of backspace and arrow keys, scrolling through command history, command auto-completion, and argument hints.
- Splitting of command line into arguments.
- Argument parsing, provided by `argtable3` library. This library includes APIs useful for parsing GNU style command line arguments.
- Functions for registration and dispatching of commands.

These facilities can be used together or independently. For example, it is possible to use line editing and command registration features, but use `getopt` or custom code for argument parsing, instead of `argtable3`. Likewise, it is possible to use simpler means of command input (such as `fgets`) together with the rest of the means for command splitting and argument parsing.

4.12.1 Line editing

Line editing feature lets users compose commands by typing them, erasing symbols using ‘backspace’ key, navigating within the command using left/right keys, navigating to previously typed commands using up/down keys, and performing autocompletion using ‘tab’ key.

Note: This feature relies on ANSI escape sequence support in the terminal application. As such, serial monitors which display raw UART data can not be used together with the line editing library. If you see `[6n` or similar escape sequence when running `get_started/console` example instead of a command prompt (`[esp32]>`), it means that the serial monitor does not support escape sequences. Programs which are known to work are GNU screen, minicom, and `idf_monitor.py` (which can be invoked using `make monitor` from project directory).

Here is an overview of functions provided by `linenoise` library.

Configuration

`linenoise` library does not need explicit initialization. However, some configuration defaults may need to be changed before invoking the main line editing function.

`linenoiseClearScreen` Clear terminal screen using an escape sequence and position the cursor at the top left corner.

`linenoiseSetMultiLine` Switch between single line and multi line editing modes. In single line mode, if the length of the command exceeds the width of the terminal, the command text is scrolled within the line to show the end of the text. In this case the beginning of the text is hidden. Single line needs less data to be sent to refresh screen on each key press, so exhibits less glitching compared to the multi line mode. On the flip side, editing commands and copying command text from terminal in single line mode is harder. Default is single line mode.

Main loop

`linenoise` In most cases, console applications have some form of read/eval loop. `linenoise` is the single function which handles user’s key presses and returns completed line once ‘enter’ key is pressed. As such, it handles the ‘read’ part of the loop.

`linenoiseFree` This function must be called to release the command line buffer obtained from `linenoise` function.

Hints and completions

linenoiseSetCompletionCallback When user presses ‘tab’ key, linenoise library invokes completion callback. The callback should inspect the contents of the command typed so far and provide a list of possible completions using calls to linenoiseAddCompletion function. linenoiseSetCompletionCallback function should be called to register this completion callback, if completion feature is desired.

console component provides a ready made function to provide completions for registered commands, esp_console_get_completion (see below).

linenoiseAddCompletion Function to be called by completion callback to inform the library about possible completions of the currently typed command.

linenoiseSetHintsCallback Whenever user input changes, linenoise invokes hints callback. This callback can inspect the command line typed so far, and provide a string with hints (which can include list of command arguments, for example). The library then displays the hint text on the same line where editing happens, possibly with a different color.

linenoiseSetFreeHintsCallback If hint string returned by hints callback is dynamically allocated or needs to be otherwise recycled, the function which performs such cleanup should be registered via linenoiseSetFreeHintsCallback.

History

linenoiseHistorySetMaxLen This function sets the number of most recently typed commands to be kept in memory. Users can navigate the history using up/down arrows.

linenoiseHistoryAdd Linenoise does not automatically add commands to history. Instead, applications need to call this function to add command strings to the history.

linenoiseHistorySave Function saves command history from RAM to a text file, for example on an SD card or on a filesystem in flash memory.

linenoiseHistoryLoad Counterpart to linenoiseHistorySave, loads history from a file.

linenoiseHistoryFree Releases memory used to store command history. Call this function when done working with linenoise library.

4.12.2 Splitting of command line into arguments

console component provides esp_console_split_argv function to split command line string into arguments. The function returns the number of arguments found (argc) and fills an array of pointers which can be passed as argv argument to any function which accepts arguments in argc, argv format.

The command line is split into arguments according to the following rules:

- Arguments are separated by spaces
- If spaces within arguments are required, they can be escaped using \ (backslash) character.
- Other escape sequences which are recognized are \\ (which produces literal backslash) and \\", which produces a double quote.
- Arguments can be quoted using double quotes. Quotes may appear only in the beginning and at the end of the argument. Quotes within the argument must be escaped as mentioned above. Quotes surrounding the argument are stripped by esp_console_split_argv function.

Examples:

- abc def 1 20 .3 [abc, def, 1, 20, .3]

- abc "123 456" def [abc, 123 456, def]
- `a\ b\\c\" [a b\c"]

4.12.3 Argument parsing

For argument parsing, `console` component includes `argtable3` library. Please see [tutorial](#) for an introduction to `argtable3`. Github repository also includes [examples](#).

4.12.4 Command registration and dispatching

`console` component includes utility functions which handle registration of commands, matching commands typed by the user to registered ones, and calling these commands with the arguments given on the command line.

Application first initializes command registration module using a call to `esp_console_init`, and calls `esp_console_cmd_register` function to register command handlers.

For each command, application provides the following information (in the form of `esp_console_cmd_t` structure):

- Command name (string without spaces)
- Help text explaining what the command does
- Optional hint text listing the arguments of the command. If application uses Argtable3 for argument parsing, hint text can be generated automatically by providing a pointer to `argtable` argument definitions structure instead.
- The command handler function.

A few other functions are provided by the command registration module:

`esp_console_run` This function takes the command line string, splits it into argc/argv argument list using `esp_console_split_argv`, looks up the command in the list of registered components, and if it is found, executes its handler.

`esp_console_split_argv` Adds help command to the list of registered commands. This command prints the list of all the registered commands, along with their arguments and help texts.

`esp_console_get_completion` Callback function to be used with `linenoiseSetCompletionCallback` from `linenoise` library. Provides completions to `linenoise` based on the list of registered commands.

`esp_console_get_hint` Callback function to be used with `linenoiseSetHintsCallback` from `linenoise` library. Provides argument hints for registered commands to `linenoise`.

4.12.5 Example

Example application illustrating usage of the `console` component is available in `examples/system/console` directory. This example shows how to initialize UART and VFS functions, set up `linenoise` library, read and handle commands from UART, and store command history in Flash. See `README.md` in the example directory for more details.

4.13 ESP32 ROM console

When an ESP32 is unable to boot from flash ROM (and the fuse disabling it hasn't been blown), it boots into a rom console. The console is based on TinyBasic, and statements entered should be in the form of BASIC statements. As

is common in the BASIC language, without a preceding line number, commands entered are executed immediately; lines with a prefixed line number are stored as part of a program.

4.13.1 Full list of supported statements and functions

System

- BYE - exits Basic, reboots ESP32, retries booting from flash
- END - stops execution from the program, also “STOP”
- MEM - displays memory usage statistics
- NEW - clears the current program
- RUN - executes the current program

IO, Documentation

- PEEK(address) - get a 32-bit value from a memory address
- POKE - write a 32-bit value to memory
- USR(addr, arg1, ..) - Execute a machine language function
- PRINT expression - print out the expression, also “?”
- PHEX expression - print expression as a hex number
- REM stuff - remark/comment, also “””

Expressions, Math

- A=V, LET A=V - assign value to a variable
- +, -, *, / - Math
- <, <=, =, >=, !=, > - Comparisons
- ABS(expression) - returns the absolute value of the expression
- RSEED(v) - sets the random seed to v
- RND(m) - returns a random number from 0 to m
- A=1234 - * Assign a decimal value*
- A=&h1A2 - * Assign a hex value*
- A=&b1001 - Assign a binary value

Control

- IF expression THEN statement - perform statement if expression is true
- FOR variable = start TO end - start for block
- FOR variable = start TO end STEP value - start for block with step
- NEXT - end of for block

- GOTO 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*

4.13.2 Example programs

Here are a few example commands and programs to get you started...

Read UART_DATE register of uart0

```
> PHEX PEEK(&h3FF40078)  
15122500
```

Set GPIO2 using memory writes to GPIO_OUT_REG

Note: you can do this easier with the IOSET command

```
> POKE &h3FF44004,PEEK(&h3FF44004) OR &b100
```

Get value of GPIO0

```
> IODIR 0,0  
> PRINT IOGET(0)  
0
```

Blink LED

Hook up an LED between GPIO2 and ground. When running the program, the LED should blink 10 times.

```
10 IODIR 2,1  
20 FOR A=1 TO 10  
30 IOSET 2,1  
40 DELAY 250  
50 IOSET 2,0  
60 DELAY 250  
70 NEXT A  
RUN
```

4.13.3 Credits

The ROM console is based on “TinyBasicPlus” by Mike Field and Scott Lawrence, which is based on “68000 Tiny-Basic” by Gordon Brandy

4.14 Wi-Fi Driver

4.14.1 Important Notes

- This document describes the implementation of only the **latest** IDF release. Backward compatibility with older versions of ESP-IDF is not guaranteed.
- This document describes the features which have already been implemented in the **latest** IDF release. For features that are now in developing/testing status, we also provide brief descriptions, while indicating the release versions in which these features will be eventually implemented.
- If you find anything wrong/ambiguous/hard to understand or inconsistent with the implementation, feel free to let us know about it on our IDF GitHub page.

4.14.2 ESP32 Wi-Fi Feature List

- Supports Station-only mode, SoftAP-only mode, Station/SoftAP-coexistence mode
- Supports IEEE-802.11B, IEEE-802.11G, IEEE802.11N and APIs to configure the protocol mode
- Supports WPA/WPA2/WPA2-Enterprise and WPS
- Supports AMPDU, HT40, QoS and other key features
- Supports Modem-sleep
- Supports an Espressif-specific protocol which, in turn, supports up to **1 km** of data traffic
- Up to 20 MBit/sec TCP throughput and 30 MBit/sec UDP throughput over the air
- Supports Sniffer
- Support set fast_crypto algorithm and normal algorithm switch which used in wifi connect

4.14.3 How To Write a Wi-Fi Application

Preparation

Generally, the most effective way to begin your own Wi-Fi application is to select an example which is similar to your own application, and port the useful part into your project. It is not a MUST but it is strongly recommended that you take some time to read this article first, especially if you want to program a robust Wi-Fi application. This article is supplementary to the Wi-Fi APIs/Examples. It describes the principles of using the Wi-Fi APIs, the limitations of the current Wi-Fi API implementation, and the most common pitfalls in using Wi-Fi. This article also reveals some design details of the Wi-Fi driver. We recommend that you become familiar at least with the following sections: <[ESP32 Wi-Fi API Error Code](#)>, <[ESP32 Wi-Fi Programming Model](#)>, and <[ESP32 Wi-Fi Event Description](#)>.

Setting Wi-Fi Compile-time Options

Refer to <[Wi-Fi Menuconfig](#)>

Init Wi-Fi

Refer to [<ESP32 Wi-Fi Station General Scenario>](#), [<ESP32 Wi-Fi soft-AP General Scenario>](#).

Start/Connect Wi-Fi

Refer to [<ESP32 Wi-Fi Station General Scenario>](#), [<ESP32 Wi-Fi soft-AP General Scenario>](#).

Event-Handling

Generally, it is easy to write code in “sunny-day” scenarios, such as [<SYSTEM_EVENT_STA_START>](#), [<SYSTEM_EVENT_STA_CONNECTED>](#) etc. The hard part is to write routines in “rainy-day” scenarios, such as [<SYSTEM_EVENT_STA_DISCONNECTED>](#) etc. Good handling of “rainy-day” scenarios is fundamental to robust Wi-Fi applications. Refer to [<ESP32 Wi-Fi Event Description>](#), [<ESP32 Wi-Fi Station General Scenario>](#), [<ESP32 Wi-Fi soft-AP General Scenario>](#)

Write Error-Recovery Routines Correctly at All Times

Just like the handling of “rainy-day” scenarios, a good error-recovery routine is also fundamental to robust Wi-Fi applications. Refer to [<ESP32 Wi-Fi API Error Code>](#)

4.14.4 ESP32 Wi-Fi API Error Code

All of the ESP32 Wi-Fi APIs have well-defined return values, namely, the error code. The error code can be categorized into:

- No errors, e.g. `ESP_ERR_WIFI_OK` means that the API returns successfully
- Recoverable errors, such as `ESP_ERR_WIFI_NO_MEM`, etc.
- Non-recoverable, non-critical errors
- Non-recoverable, critical errors

Whether the error is critical or not depends on the API and the application scenario, and it is defined by the API user.

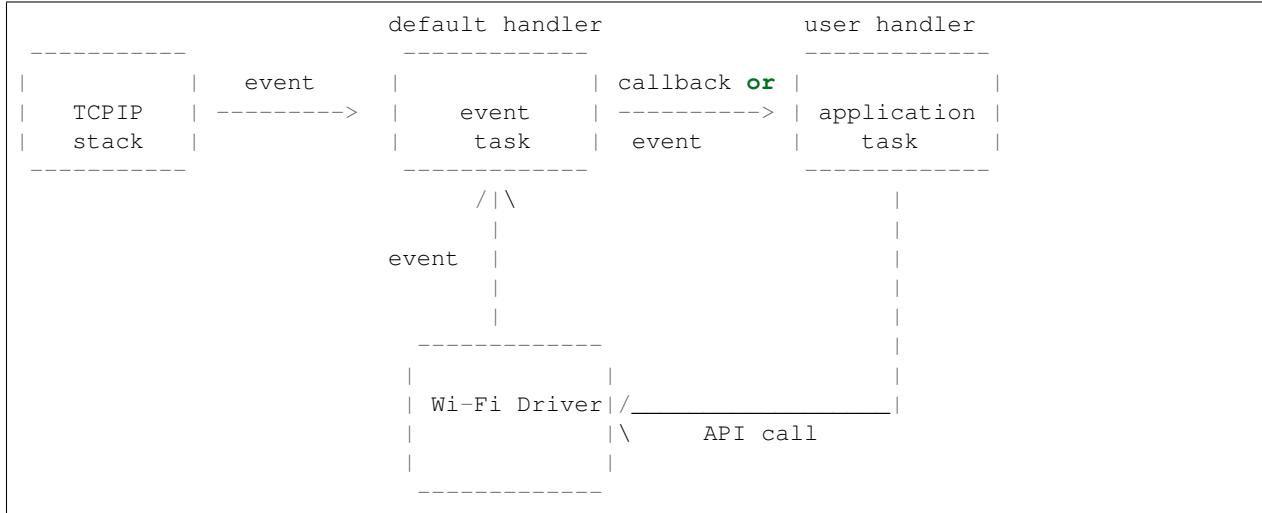
The primary principle to write a robust application with Wi-Fi API is to always check the error code and write the error-handling code. Generally, the error-handling code can be used:

- for recoverable errors, in which case you can write a recoverable-error code. For example, when `esp_wifi_start` returns `ESP_ERR_WIFI_NO_MEM`, the recoverable-error code `vTaskDelay` can be called, in order to get a microseconds’ delay for another try.
- for non-recoverable, yet non-critical, errors, in which case printing the error code is a good method for error handling.
- for non-recoverable, critical errors, in which case “assert” may be a good method for error handling. For example, if `esp_wifi_set_mode` returns `ESP_ERR_WIFI_NOT_INIT`, it means that the Wi-Fi driver is not initialized by `esp_wifi_init` successfully. You can detect this kind of error very quickly in the application development phase.

In `esp_err.h`, `ESP_ERROR_CHECK` checks the return values. It is a rather commonplace error-handling code and can be used as the default error-handling code in the application development phase. However, we strongly recommend that the API user writes their own error-handling code.

4.14.5 ESP32 Wi-Fi Programming Model

The ESP32 Wi-Fi programming model is depicted as follows:



The Wi-Fi driver can be considered a black box that knows nothing about high-layer code, such as the TCPIP stack, application task, event task, etc. All the Wi-Fi driver can do is receive API calls from the high layer, or post an event-queue to a specified queue which is initialized by API `esp_wifi_init()`.

The event task is a daemon task which receives events from the Wi-Fi driver or from other subsystems, such as the TCPIP stack. The event task will call the default callback function upon receiving the event. For example, upon receiving `SYSTEM_EVENT_STA_CONNECTED`, it will call `tcpip_adapter_start()` to start the DHCP client in its default handler.

An application can register its own event callback function by using API `esp_event_init`. Then, the application callback function will be called after the default callback. Also, if the application does not want to execute the callback in the event task, it needs to post the relevant event to the application task in the application callback function.

The application task (code) generally mixes all these things together: it calls APIs to initialize the system/Wi-Fi and handle the events when necessary.

4.14.6 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.

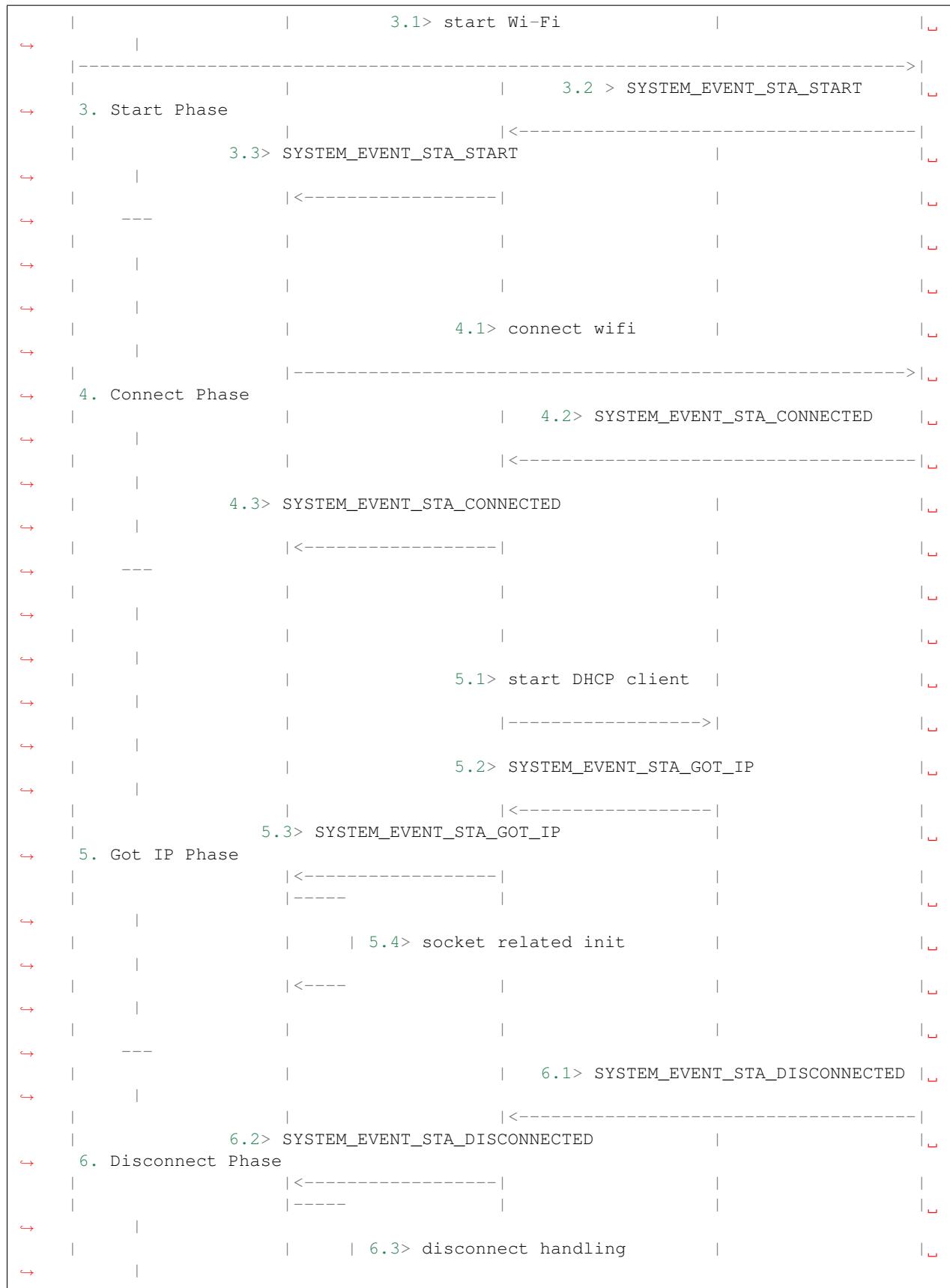
SYSTEM_EVENT_AP_PROBEREQRECVED

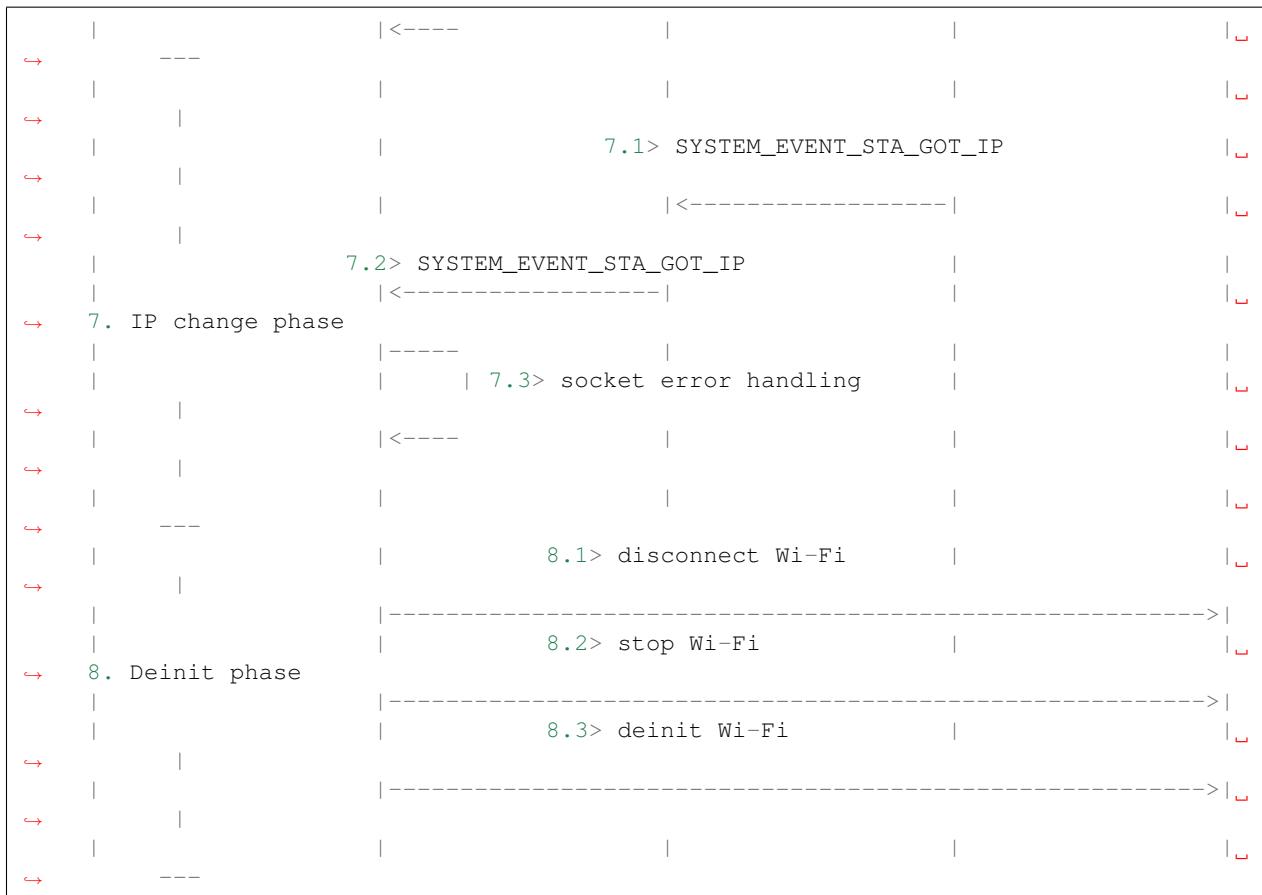
Currently, the ESP32 implementation will never generate this event. It may be removed in future releases.

4.14.7 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 Wi-Fi driver at this stage, because the configuration APIs (such as `esp_wifi_set_protocol`) will cause the Wi-Fi to reconnect, which may not be desirable.

If the Wi-Fi NVS flash is enabled by menuconfig, all Wi-Fi configuration in this phase, or later phases, will be stored into flash. When the board powers on/reboots, you do not need to configure the Wi-Fi driver from scratch. You only need to call `esp_wifi_get_xxx` APIs to fetch the configuration stored in flash previously. You can also configure the Wi-Fi driver if the previous configuration is not what you want.

3. Wi-Fi Start Phase

- s3.1: Call `esp_wifi_start` to start the Wi-Fi driver.
- s3.2: The Wi-Fi driver posts `<SYSTEM_EVENT_STA_START>` to the event task; then, the event task will do some common things and will call the application event callback function.
- s3.3: The application event callback function relays the `<SYSTEM_EVENT_STA_START>` to the application task. We recommend that you call `esp_wifi_connect()`. However, you can also call `esp_wifi_connect()` in other phrases after the `<SYSTEM_EVENT_STA_START>` arises.

4. Wi-Fi Connect Phase

- s4.1: Once `esp_wifi_connect()` is called, the Wi-Fi driver will start the internal scan/connection process.
- s4.2: If the internal scan/connection process is successful, the `<SYSTEM_EVENT_STA_CONNECTED>` will be generated. In the event task, it starts the DHCP client, which will finally trigger the DHCP process.
- s4.3: In the above-mentioned scenario, the application event callback will relay the event to the application task. Generally, the application needs to do nothing, and you can do whatever you want, e.g., print a log, etc.

In step 4.2, the Wi-Fi connection may fail because, for example, the password is wrong, the AP is not found, etc. In a case like this, `<SYSTEM_EVENT_STA_DISCONNECTED>` will arise and the reason for such a failure will be provided. For handling events that disrupt Wi-Fi connection, please refer to phase 6.

5. Wi-Fi ‘Got IP’ Phase

- s5.1: Once the DHCP client is initialized in step 4.2, the *got IP* phase will begin.
- s5.2: If the IP address is successfully received from the DHCP server, then `<SYSTEM_EVENT_STA_GOT_IP>` will arise and the event task will perform common handling.
- s5.3: In the application event callback, `<SYSTEM_EVENT_STA_GOT_IP>` is relayed to the application task. For LwIP-based applications, this event is very special and means that everything is ready for the application to begin its tasks, e.g. creating the TCP/UDP socket, etc. A very common mistake is to initialize the socket before `<SYSTEM_EVENT_STA_GOT_IP>` is received. **DO NOT start the socket-related work before the IP is received.**

6. Wi-Fi Disconnect Phase

- s6.1: When the Wi-Fi connection is disrupted, e.g. because the AP is powered off, the RSSI is poor, etc., `<SYSTEM_EVENT_STA_DISCONNECTED>` will arise. This event may also arise in phase 3. Here, the event task will notify the LwIP task to clear/remove all UDP/TCP connections. Then, all application sockets will be in a wrong status. In other words, no socket can work properly when this event happens.

- s6.2: In the scenario described above, the application event callback function relays `<SYSTEM_EVENT_STA_DISCONNECTED>` to the application task. We recommend that `esp_wifi_connect()` be called to reconnect the Wi-Fi, close all sockets and re-create them if necessary. Refer to `<SYSTEM_EVENT_STA_DISCONNECTED>`.

7. Wi-Fi IP Change Phase

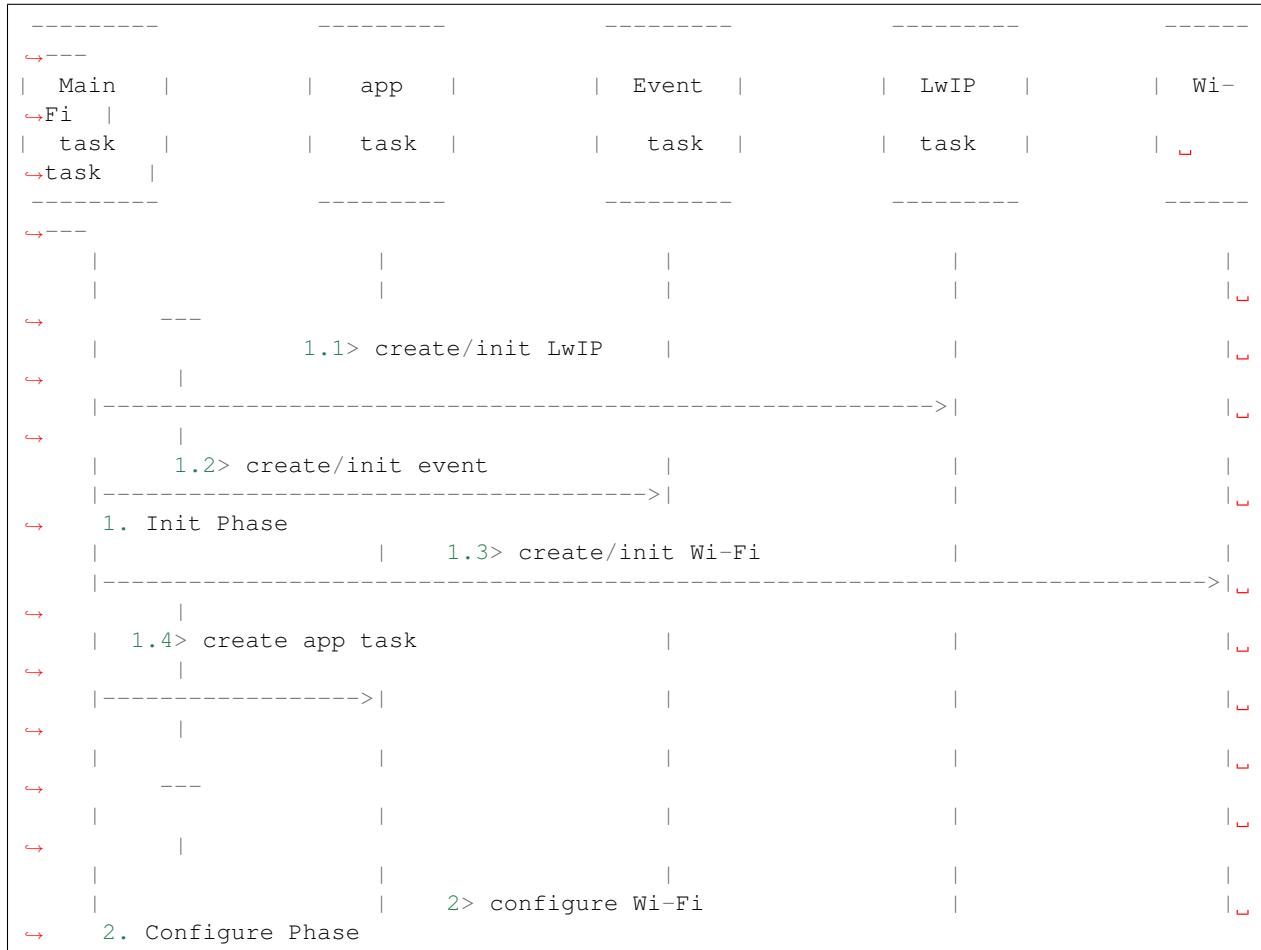
- s7.1: If the IP address is changed, the `<SYSTEM_EVENT_STA_GOT_IP>` will arise.
- s7.2: This event is important to the application. When it occurs, the timing is good for closing all created sockets and recreating them.

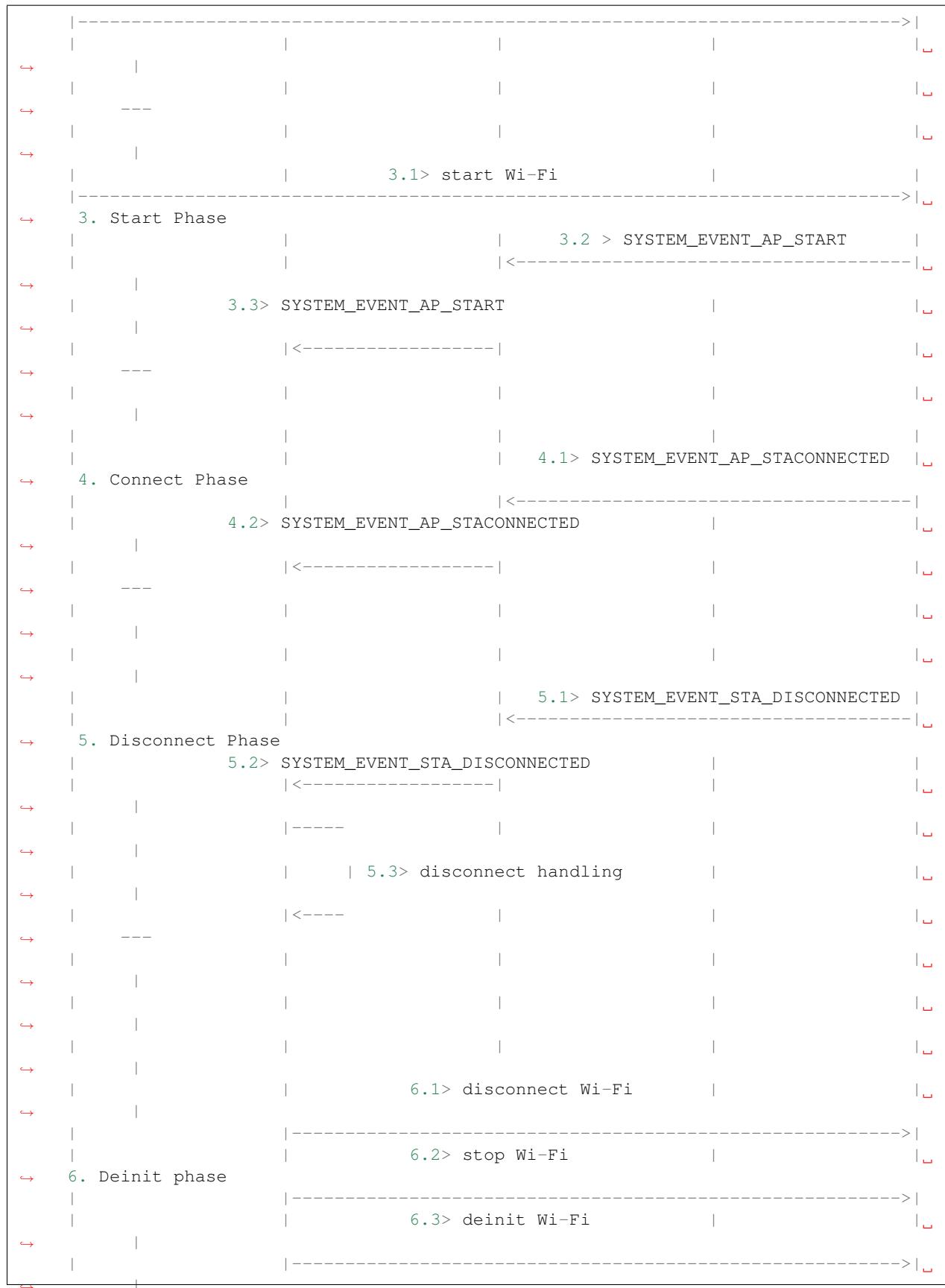
8. Wi-Fi Deinit Phase

- s8.1: Call `esp_wifi_disconnect()` to disconnect the Wi-Fi connectivity.
- s8.2: Call `esp_wifi_stop()` to stop the Wi-Fi driver.
- s8.3: Call `esp_wifi_deinit()` to unload the Wi-Fi driver.

4.14.8 ESP32 Wi-Fi soft-AP General Scenario

Below is a “big scenario” which describes some small scenarios in Soft-AP mode:







4.14.9 ESP32 Wi-Fi Scan

Currently, the `esp_wifi_scan_start()` API is supported only in Station or Station+SoftAP mode.

Scan Type

Mode	Description
Active Scan	Scan by sending a probe request. The default scan is an active scan.
Passive Scan	No probe request is sent out. Just switch to the specific channel and wait for a beacon. Application can enable it via the <code>scan_type</code> field of <code>wifi_scan_config_t</code> .
Foreground Scan	This scan is applicable when there is no Wi-Fi connection in Station mode. Foreground or background scanning is controlled by the Wi-Fi driver and cannot be configured by the application.
Background Scan	This scan is applicable when there is a Wi-Fi connection in Station mode or in Station+SoftAP mode. Whether it is a foreground scan or background scan depends on the Wi-Fi driver and cannot be configured by the application.
All-Channel Scan	It scans all of the channels. If the channel field of <code>wifi_scan_config_t</code> is set to 0, it is an all-channel scan.
Specific Channel Scan	It scans specific channels only. If the channel field of <code>wifi_scan_config_t</code> set to 1, it is a specific-channel scan.

The scan modes in above table can be combined arbitrarily, so we totally have 8 different scans:

- All-Channel Background Active Scan
- All-Channel Background Passive Scan
- All-Channel Foreground Active Scan
- All-Channel Foreground Passive Scan
- Specific-Channel Background Active Scan
- Specific-Channel Background Passive Scan
- Specific-Channel Foreground Active Scan
- Specific-Channel Foreground Passive Scan

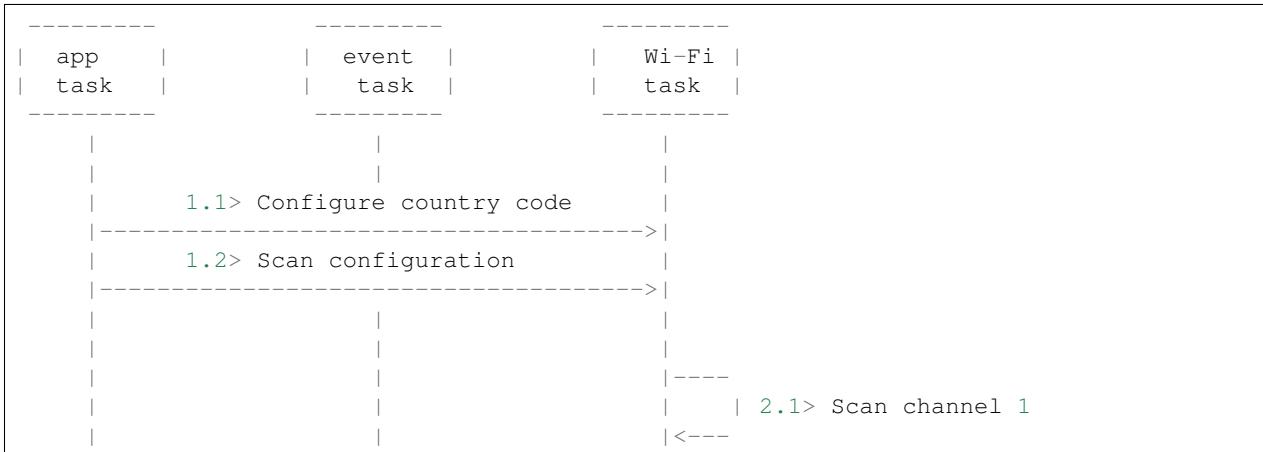
Scan Configuration

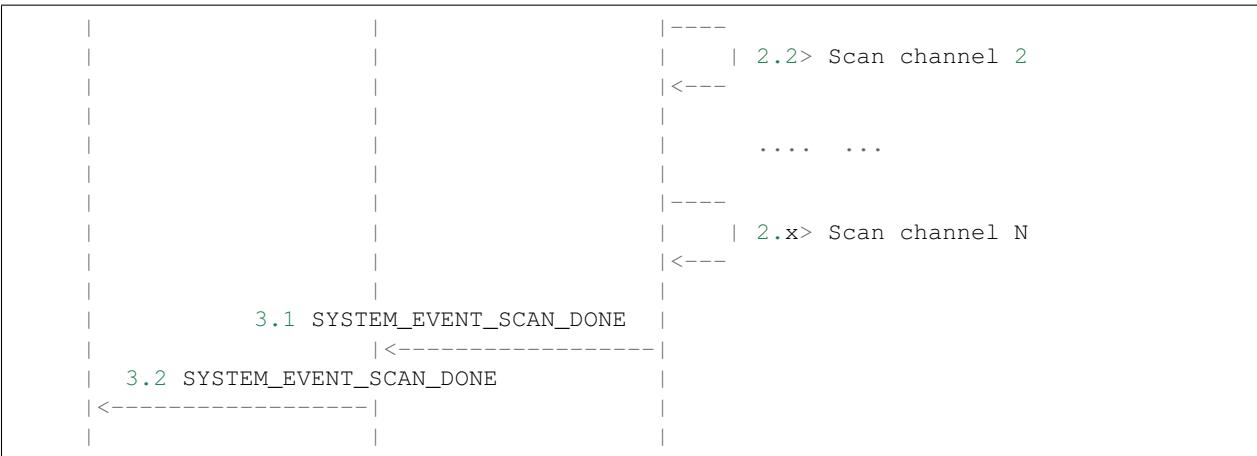
The scan type and other scan attributes are configured by `esp_wifi_scan_start`. The table below provides a detailed description of `wifi_scan_config_t`.

Field	Description
ssid	If the SSID is not NULL, it is only the AP with the same SSID that can be scanned.
bssid	If the BSSID is not NULL, it is only the AP with the same BSSID that can be scanned.
channel	If “channel” is 0, there will be an all-channel scan; otherwise, there will be a specific-channel scan.
show_hidden	If “show_hidden” is 0, the scan ignores the AP with a hidden SSID; otherwise, the scan considers the hidden AP a normal one.
scan_type	If “scan_type” is WIFI_SCAN_TYPE_ACTIVE, the scan is “active”; otherwise, it is a “passive” one.
scan_time	<p>This field is used to control how long the scan dwells on each channel.</p> <p>For passive scans, scan_time.passive designates the dwell time for each channel.</p> <p>For active scans, dwell times for each channel are listed in the table below. Here, min is short for scan_time.active.min and max is short for scan_time.active.max.</p> <ul style="list-style-type: none"> • min=0, max=0: scan dwells on each channel for 120 ms. • min>0, max=0: scan dwells on each channel for 120 ms. • min=0, max>0: scan dwells on each channel for max ms. • min>0, max>0: the minimum time the scan dwells on each channel is min ms. If no AP is found during this time frame, the scan switches to the next channel. Otherwise, the scan dwells on the channel for max ms. <p>If you want to improve the performance of the the scan, you can try to modify these two parameters.</p>

Scan All APs In All Channels (foreground)

Scenario:





The scenario above describes an all-channel, foreground scan. The foreground scan can only occur in Station mode where the station does not connect to any AP. Whether it is a foreground or background scan is totally determined by the Wi-Fi driver, and cannot be configured by the application.

Detailed scenario description:

Scan Configuration Phase

- s1.1: Call `esp_wifi_set_country()` to set the country 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

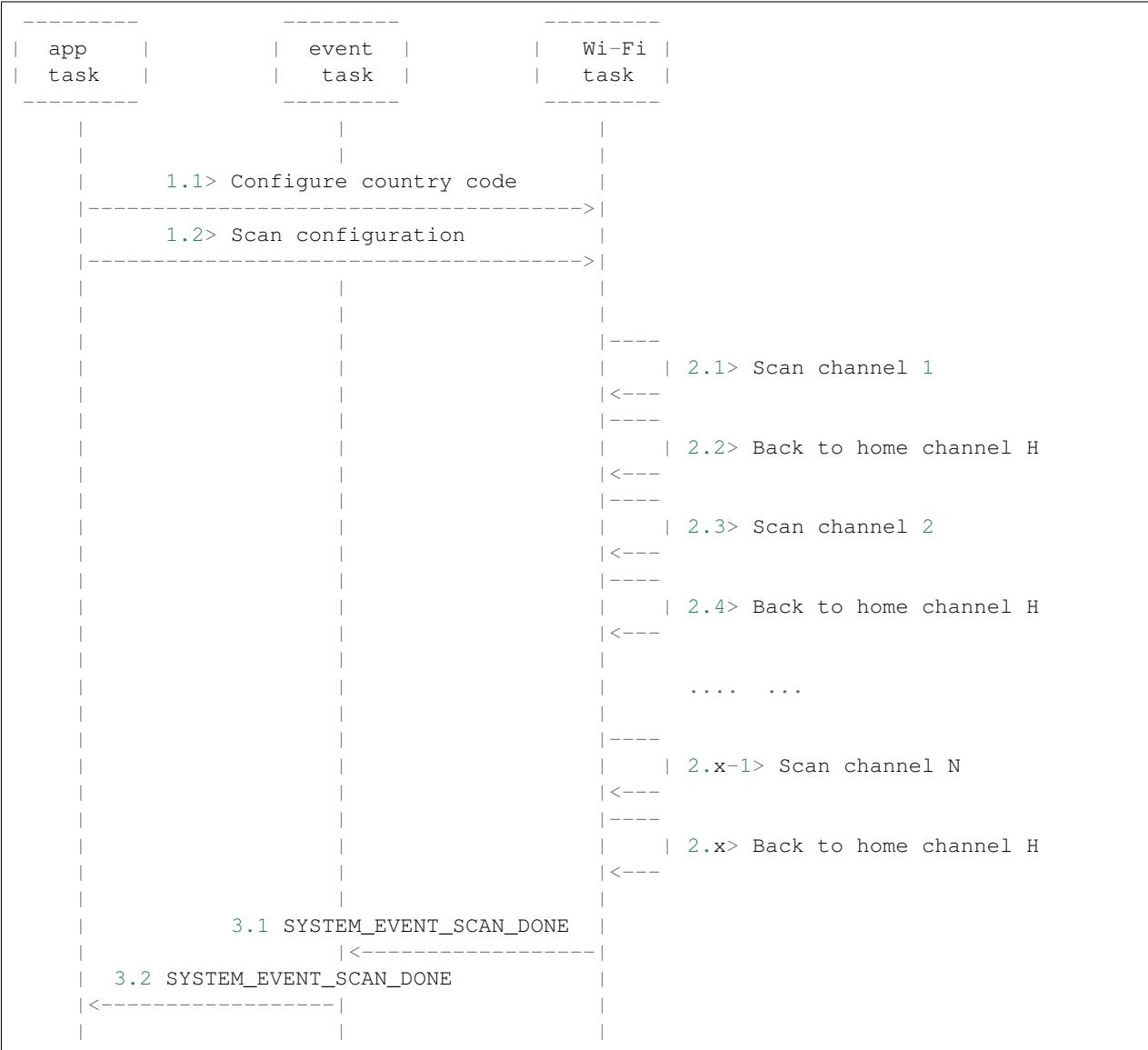
- s2.1: The Wi-Fi driver switches to channel 1, in case the scan type is `WIFI_SCAN_TYPE_ACTIVE`, and broadcasts a probe request. Otherwise, the Wi-Fi will wait for a beacon from the APs. The Wi-Fi driver will stay in channel 1 for some time. The dwell time is configured in min/max time, with default 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)

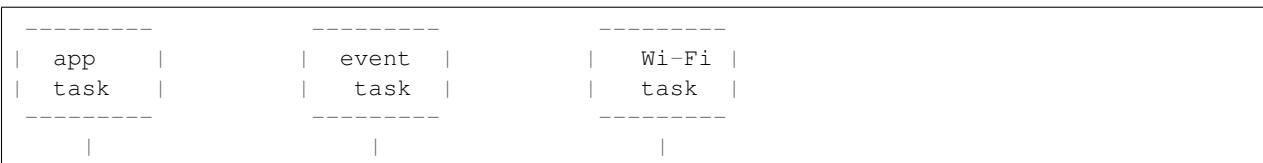
Scenario:

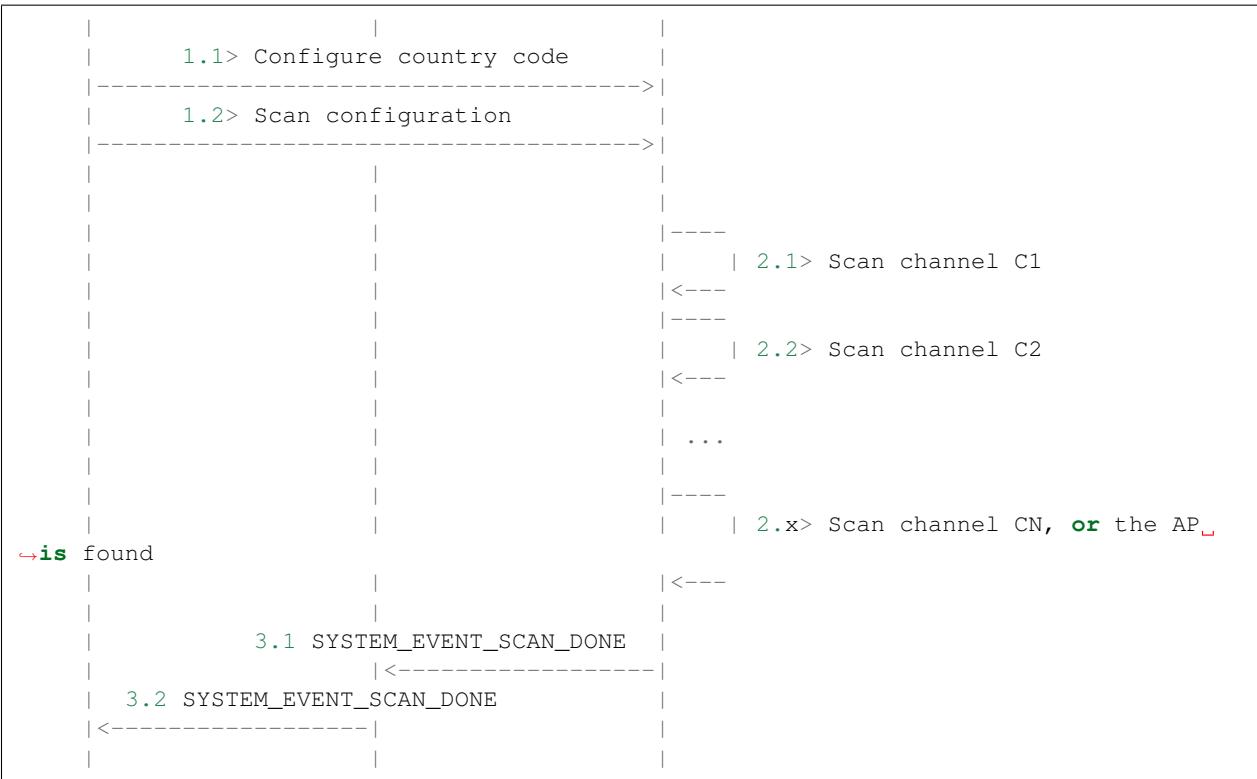


The scenario above is an all-channel background scan. Compared to [*Scan All APs In All Channels\(foreground\)*](#), the difference in the all-channel background scan is that the Wi-Fi driver will scan the back-to-home channel for 30 ms before it switches to the next channel to give the Wi-Fi connection 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.

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

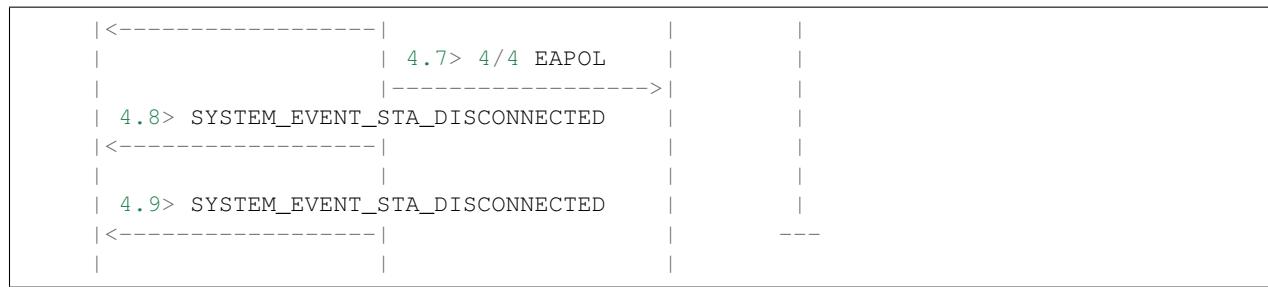
Two application tasks may call `esp_wifi_scan()` at the same time, or the same application task calls `esp_wifi_scan_start()` before it gets a scan-done event. Both scenarios can happen. **However, in IDF2.1, the Wi-Fi driver does not support parallel scans adequately. As a result, a parallel scan should be avoided.** The parallel scan will be enhanced in future releases, as the ESP32's Wi-Fi functionality improves continuously.

4.14.10 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.
- 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	ESP32 value	Mapped To Standard Value	Description
UNSPECIFIED	1	1	Generally, it means an internal failure, e.g., the memory runs out, the internal TX fails, or the reason is received from the remote side, etc.
AUTH_EXPIRE	2	2	<p>The previous authentication is no longer valid. For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> • auth is timed out • the reason is received from the soft-AP. <p>For the ESP32 SoftAP, this reason is reported when:</p> <ul style="list-style-type: none"> • the soft-AP has not received any packets from the station in the past five minutes. • the soft-AP is stopped by calling <code>esp_wifi_stop()</code>. • the station is de-authed by calling <code>esp_wifi_deauth_sta()</code>.
AUTH_LEAVE	3	3	<p>De-authenticated, because the sending STA is leaving (or has left). For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> • it is received from the soft-AP.
ASSOC_EXPIRE	4	4	<p>Disassociated due to inactivity. For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> • it is received from the soft-AP. <p>For the ESP32 Soft-AP, this reason is reported when:</p> <ul style="list-style-type: none"> • the soft-AP has not received any packets from the station in the past five minutes.
626			Chapter 4. API Guides <ul style="list-style-type: none"> • the soft-AP is stopped by calling <code>esp_wifi_stop()</code>. • the station is de-authed by calling <code>esp_wifi_deauth_sta()</code>.

4.14.11 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
<code>WIFI_MODE_NULL</code>	in this mode, the internal data struct is not allocated to the station and the soft-AP, while both the station and soft-AP interfaces are not initialized for RX/TX Wi-Fi data. Generally, this mode is used for Sniffer, or when you only want to stop both the STA and the AP without calling <code>esp_wifi_deinit()</code> to unload the whole Wi-Fi driver.
<code>WIFI_MODE_STA</code>	in this mode, <code>esp_wifi_start()</code> will init the internal station data, while the station's interface is ready for the RX and TX Wi-Fi data. After <code>esp_wifi_connect()</code> is called, the STA will connect to the target AP.
<code>WIFI_MODE_AP</code>	in this mode, <code>esp_wifi_start()</code> will init the internal soft-AP data, while the soft-AP's interface is ready for RX/TX Wi-Fi data. Then, the Wi-Fi driver starts broad-casting beacons, and the soft-AP is ready to get connected to other stations.
<code>WIFI_MODE_APSTA</code>	coexistence mode: in this mode, <code>esp_wifi_start()</code> will simultaneously init both the station and the soft-AP. This is done in station mode and soft-AP mode. Please note that the channel of the external AP, which the ESP32 Station is connected to, has higher priority over the ESP32 Soft-AP channel. Refer to 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
<code>ssid</code>	This is the SSID of the target AP, to which the station wants to connect to.
<code>pass-word</code>	Password of the target AP
<code>bssid</code>	If <code>bssid_set</code> is 0, the station connects to the AP whose SSID is the same as the field “ <code>ssid</code> ”, while the field “ <code>bssid</code> ” is ignored. In all other cases, the station connects to the AP whose SSID is the same as the “ <code>ssid</code> ” field, while its BSSID is the same as the “ <code>bssid</code> ” field .
<code>bssid</code>	This is valid only when <code>bssid_set</code> is 1; see field “ <code>bssid_set</code> ”.
<code>channel</code>	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 “ <code>channel</code> ” 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.
pass-word	Password of soft-AP; if the auth mode is WIFI_AUTH_OPEN, this field will be ignored.
ssid_len	Length of SSID; if ssid_len is 0, check the SSID until there is a termination character. If ssid_len > 32, change it to 32; otherwise, set the SSID length according to ssid_len.
channel	Channel of soft-AP; if the channel is out of range, the Wi-Fi driver defaults the channel to channel 1. So, please make sure the channel is within the required range. For more details, refer to < Channel Range >.
auth-mode	Auth mode of ESP32 soft-AP; currently, ESP32 Wi-Fi does not support AUTH_WEP. If the authmode is an invalid value, soft-AP defaults the value to WIFI_AUTH_OPEN.
ssid_hidden	If ssid_hidden is 1, soft-AP does not broadcast the SSID; otherwise, it does broadcast the SSID.
max_connection	Currently, ESP32 Wi-Fi supports up to 10 Wi-Fi connections. If max_connection > 10, soft-AP defaults the value to 10.
beacon_interval	Beacon interval; the value is 100 ~ 60000 ms, with default value being 100 ms. If the value is out of range, soft-AP defaults it to 100 ms.

Wi-Fi Protocol Mode

Currently, the IDF supports the following protocol modes:

Protocol Mode	Description
802.11 B	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B) to set the station/soft-AP to 802.11B-only mode.
802.11 BG	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G) to set the station/soft-AP to 802.11BG mode.
802.11 BGN	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G WIFI_PROTOCOL_11N) to set the station/ soft-AP to BGN mode.
802.11 BGNLR	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G WIFI_PROTOCOL_11N WIFI_PROTOCOL_11LR) to set the station/soft-AP to BGN and the Espressif-specific mode.
802.11 LR	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.

4.14.12 ESP32 Wi-Fi Power-saving Mode

Currently, ESP32 Wi-Fi supports the Modem-sleep mode which refers to WMM (Wi-Fi Multi Media) power-saving mode in the IEEE 802.11 protocol. If the Modem-sleep mode is enabled and the Wi-Fi enters a sleep state, then, RF, PHY and BB are turned off in order to reduce power consumption. Modem-sleep mode works in Station-only mode and the station must be connected to the AP first.

Call `esp_wifi_set_ps(WIFI_PS_MODEM)` to enable Modem-sleep mode after calling `esp_wifi_init()`. About 10 seconds after the station connects to the AP, Modem-sleep will start. When the station disconnects from the AP, Modem-sleep will stop.

4.14.13 ESP32 Wi-Fi Connect Crypto

Now ESP32 have two group crypto functions can be used when do wifi connect, one is the original functions, the other is optimized by ESP hardware: 1. Original functions which is the source code used in the folder components/wpa_supplicant/src/crypto function; 2. The optimized functions is in the folder components/wpa_supplicant/src/fast_crypto, these function used the hardware crypto to make it faster than origin one, the type of function's name add *fast_* to distinguish with the original one. For example, the API `aes_wrap()` is used to encrypt frame information when do 4 way handshake, the `fast_aes_wrap()` has the same result but can be faster.

Two groups of crypto function can be used when register in the `wpa_crypto_funcs_t`, `wpa2_crypto_funcs_t` and `wps_crypto_funcs_t` structure, also we have given the recommend functions to register in the `fast_crypto_ops.c`, you can register the function as the way you need, however what should make action is that the `crypto_hash_xxx` function and `crypto_cipher_xxx` function need to register with the same function to operation. For example, if you register `crypto_hash_init()` function to initialize the `esp_crypto_hash` structure, you need use the `crypto_hash_update()` and `crypto_hash_finish()` function to finish the operation, rather than `fast_crypto_hash_update()` or `fast_crypto_hash_finish()`.

4.14.14 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 Buffer RX	15	Menuconfig
Dynamic Buffer RX	Unlimited	Menuconfig
Dynamic Buffer TX	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 MBOX RX	12	Release 2.1/2.0 and earlier: DEFAULT_TCP_RECVMBOX_SIZE in lwipopts.h After the 2.1 Release: Menuconfig
RX BA Window	9~16	Release 2.1/2.0 and earlier: not configurable After the 2.1 Release: Menuconfig
TX BA Window	9~16	Release 2.1/2.0 and earlier: not configurable After the 2.1 Release: Menuconfig

Once you adjust the configurations, you can then run your own test code to test the performance. You can also run the iperf example to test the performance. However, the iperf example is not provided in release 2.1 and earlier ones, but will be so in the upcoming release. Those who really care about the performance should seek support from Espressif directly, so that we can provide them with the iperf version bin for their testing.

If you decide to modify some of the configurations in order to gain better throughput for your application, please consider the memory usage very carefully. For a more detailed description, refer to <[Wi-Fi Buffer Usage](#)> and <[Wi-Fi Buffer Configure](#)>.

4.14.15 Wi-Fi 80211 Packet Send

Important notes: The API `esp_wifi_80211_tx` is not available in IDF 2.1, but will be so in the upcoming release.

The `esp_wifi_80211_tx` API can be used to:

- Send the beacon, probe request, probe response, action frame.

- Send the QoS and non-QoS data frame.

It cannot be used for sending cryptographic frames.

Parameters of esp_wifi_80211_tx

Parameter	Description
ifx	Wi-Fi interface ID: if the Wi-Fi mode is Station, the ifx should be WIFI_IF_STA. If the Wi-Fi mode is SoftAP, the ifx should be WIFI_IF_AP. If the Wi-Fi mode is Station+SoftAP, the ifx should be WIFI_IF_STA or WIFI_IF_AP. If the ifx is wrong, the API returns ESP_ERR_WIFI_IF.
buffer	Raw 802.11 buffer. For building the correct buffer, refer to the following sections: If the buffer is wrong, or violates the Wi-Fi driver's restrictions, the API returns ESP_ERR_WIFI_ARG or results in unexpected behavior. Please read the following section carefully to make sure you understand the restrictions on encapsulating the buffer.
len	The length must be <= 1500; otherwise, the API will return ESP_ERR_WIFI_ARG.
en_sys_seq	If <code>en_sys_seq</code> is true, it means that the Wi-Fi driver will rewrite the sequence number in the buffer; otherwise, it will not rewrite the sequence number. Generally, if <code>esp_wifi_80211_tx</code> is called before the Wi-Fi connection has been set up, both <code>en_sys_seq==true</code> and <code>en_sys_seq==false</code> are fine. However, if the API is called after the Wi-Fi connection has been set up, <code>en_sys_seq</code> should be true. For more details, read the following section about the sequence configuration.

Preconditions of Using esp_wifi_80211_tx

- The Wi-Fi mode is Station, or SoftAP, or Station+SoftAP.
- Either `esp_wifi_set_promiscuous(true)`, or `esp_wifi_start()`, or both of these APIs return `ESP_ERR_WIFI_OK`. This is because we need to make sure that Wi-Fi hardware is initialized before `esp_wifi_80211_tx()` is called. In ESP32, both `esp_wifi_set_promiscuous(true)` and `esp_wifi_start()` can trigger the initialization of Wi-Fi hardware.
- The parameters of `esp_wifi_80211_tx` are hereby correctly provided.

Side-Effects to Avoid in Different Scenarios

Theoretically, if we do not consider the side-effects the API imposes on the Wi-Fi driver or other stations/soft-APs, we can send a raw 802.11 packet over the air, with any destination MAC, any source MAC, any BSSID, or any other type of packet. However, robust/useful applications should avoid such side-effects. The table below provides some tips/recommendations on how to avoid the side-effects of `esp_wifi_80211_tx` in different scenarios.

Scenario	Description
NO_CONN_MGMT <ul style="list-style-type: none"> • No Wi-Fi connection • Can send management frame 	<p>In this scenario, no Wi-Fi connection is set up, so there are no side-effects on the Wi-Fi driver. If <code>en_sys_seq==true</code>, the Wi-Fi driver is responsible for the sequence control. If <code>en_sys_seq==false</code>, the application needs to ensure that the buffer has the correct sequence.</p> <p>Theoretically, the MAC address can be any address. However, this may impact other stations/soft-APs with the same MAC/BSSID.</p> <p>Side-effect example#1 The application calls <code>esp_wifi_80211_tx</code> to send a beacon with <code>BSSID == mac_x</code> in SoftAP mode, but the <code>mac_x</code> is not the MAC of the SoftAP interface. Moreover, there is another soft-AP, say “other-AP”, whose <code>bssid</code> is <code>mac_x</code>. If this happens, an “unexpected behavior” may occur, because the stations which connect to the “other-AP” cannot figure out whether the beacon is from the “other-AP” or the <code>esp_wifi_80211_tx</code>.</p> <p>To avoid the above-mentioned side-effects, we recommend that:</p> <ul style="list-style-type: none"> • If <code>esp_wifi_80211_tx</code> is called in Station mode, the first MAC should be a multicast MAC or the exact target-device’s MAC, while the second MAC should be that of the station interface. • If <code>esp_wifi_80211_tx</code> is called in 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. <p>The recommendations above are only for avoiding side-effects and can be ignored when there are good reasons for doing this.</p>
NO_CONN_NON_QOS <ul style="list-style-type: none"> • No Wi-Fi connection • Can send non-QoS frame 	Same as in NO_CONN_MGMT
NO_CONN_QOS <ul style="list-style-type: none"> • No Wi-Fi connection • Can send QoS frame 	Same as in NO_CONN_MGMT
CONN_MGMT <ul style="list-style-type: none"> • Have Wi-Fi connection • Send management frame only 	When the Wi-Fi connection is already set up, and the sequence is controlled by the application, the latter may impact the sequence control of the Wi-Fi connection, as a whole. So, the recommendation is that <code>en_sys_seq</code> be true. The MAC-address recommendations in the NO_CONN_MGMT scenario also apply to the CONN_MGMT scenario.
CONN_NON_QOS <ul style="list-style-type: none"> • Have Wi-Fi connection • Can send non-QoS frame 	<p>Generally, we should use a socket API, instead of this one, in order to send the data frame when the Wi-Fi connection is already set up.</p> <p>However, if you have any special reasons for using this particular API, then <code>en_sys_seq</code> must be true; otherwise, you may impact the internal sequence control of the Wi-Fi connection described in CONN_MGMT. The MAC-address recommendations in the NO_CONN_MGMT scenario also apply to the CONN_NON_QOS scenario.</p>
632	Chapter 1 API Guides
CONN_QOS <ul style="list-style-type: none"> • Have Wi-Fi connection • Can send non-QoS frame 	Same as in CONN_NON_QOS

4.14.16 Wi-Fi Sniffer Mode

The Wi-Fi sniffer mode can be enabled by `esp_wifi_set_promiscuous()`. If the sniffer mode is enabled, the following packets **can** be dumped to the application:

- 802.11 Management frame
- 802.11 Data frame, including MPDU, AMPDU, AMSDU, etc.
- 802.11 MIMO frame, for MIMO frame, the sniffer only dumps the length of the frame.

The following packets will **NOT** be dumped to the application:

- 802.11 Control frame
- 802.11 error frame, such as the frame with a CRC error, etc.

For frames that the sniffer **can** dump, the application can additionally decide which specific type of packets can be filtered to the application by using `esp_wifi_set_promiscuous_filter()`. By default, it will filter all 802.11 data and management frames to the application.

The Wi-Fi sniffer mode can be enabled in the Wi-Fi mode of `WIFI_MODE_NULL`, or `WIFI_MODE_STA`, or `WIFI_MODE_AP`, or `WIFI_MODE_APSTA`. In other words, the sniffer mode is active when the station is connected to the soft-AP, or when the soft-AP has a Wi-Fi connection. Please note that the sniffer has a **great impact** on the throughput of the station or soft-AP Wi-Fi connection. Generally, we should **NOT** enable the sniffer, when the station/soft-AP Wi-Fi connection experiences heavy traffic unless we have special reasons.

Another noteworthy issue about the sniffer is the callback `wifi_promiscuous_cb_t`. The callback will be called directly in the Wi-Fi driver task, so if the application has a lot of work to do for each filtered packet, the recommendation is to post an event to the application task in the callback and defer the real work to the application task.

4.14.17 Wi-Fi Buffer Usage

This section is only about the dynamic buffer configuration.

Why Buffer Configuration Is Important

In order to get a robust, high-performance system, we need to consider the memory usage/configuration very carefully, because

- the available memory in ESP32 is limited.
- currently, the default type of buffer in LwIP and Wi-Fi drivers is “dynamic”, **which means that both the LwIP and Wi-Fi share memory with the application**. Programmers should always keep this in mind; otherwise, they will face a memory issue, such as “running out of heap memory”.
- it is very dangerous to run out of heap memory, as this will cause ESP32 an “undefined behavior”. Thus, enough heap memory should be reserved for the application, so that it never runs out of it.
- the Wi-Fi throughput heavily depends on memory-related configurations, such as the TCP window size, Wi-Fi RX/TX dynamic buffer number, etc. Refer to <[ESP32 Wi-Fi Throughput](#)>.
- the peak heap memory that the ESP32 LwIP/Wi-Fi may consume depends on a number of factors, such as the maximum TCP/UDP connections that the application may have, etc.
- the total memory that the application requires is also an important factor when considering memory configuration.

Due to these reasons, there is not a good-for-all application configuration. Rather, we have to consider memory configurations separately for every different application.

Dynamic vs. Static Buffer

The default type of buffer in LwIP and Wi-Fi drivers is “dynamic”. Most of the time the dynamic buffer can significantly save memory. However, it makes the application programming a little more difficult, because in this case the application needs to consider memory usage in LwIP/Wi-Fi.

Peak LwIP Dynamic Buffer

The default type of LwIP buffer is “dynamic”, and this section considers the dynamic buffer only. The peak heap memory that LwIP consumes is the **theoretically-maximum memory** that the LwIP driver consumes. Generally, the peak heap memory that the LwIP consumes depends on:

- the memory required to create a UDP connection: `lwip_udp_conn`
- the memory required to create a TCP connection: `lwip_tcp_conn`
- the number of UDP connections that the application has: `lwip_udp_con_num`
- the number of TCP connections that the application has: `lwip_tcp_con_num`
- the TCP TX window size: `lwip_tcp_tx_win_size`
- the TCP RX window size: `lwip_tcp_rx_win_size`

So, the peak heap memory that the LwIP consumes can be calculated with the following formula:

$$\text{lwip_dynamic_peak_memory} = (\text{lwip_udp_con_num} * \text{lwip_udp_conn}) + (\text{lwip_tcp_con_num} * (\text{lwip_tcp_tx_win_size} + \text{lwip_tcp_rx_win_size} + \text{lwip_tcp_conn}))$$

Some TCP-based applications need only one TCP connection. However, they may choose to close this TCP connection and create a new one when an error (such as a sending failure) occurs. This may result in multiple TCP connections existing in the system simultaneously, because it may take a long time for a TCP connection to close, according to the TCP state machine (refer to RFC793).

Peak Wi-Fi Dynamic Buffer

The Wi-Fi driver supports several types of buffer (refer to [Wi-Fi Buffer Configure](#)). However, this section is about the usage of the dynamic Wi-Fi buffer only. The peak heap memory that Wi-Fi consumes is the **theoretically-maximum memory** that the Wi-Fi driver consumes. Generally, the peak memory depends on:

- the number of dynamic rx buffers that are configured: `wifi_rx_dynamic_buf_num`
- the number of dynamic tx buffers that are configured: `wifi_tx_dynamic_buf_num`
- the maximum packet size that the Wi-Fi driver can receive: `wifi_rx_pkt_size_max`
- the maximum packet size that the Wi-Fi driver can send: `wifi_tx_pkt_size_max`

So, the peak memory that the Wi-Fi driver consumes can be calculated with the following formula:

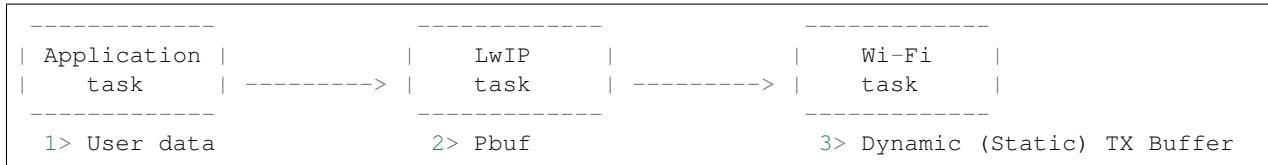
$$\text{wifi_dynamic_peak_memory} = (\text{wifi_rx_dynamic_buf_num} * \text{wifi_rx_pkt_size_max}) + (\text{wifi_tx_dynamic_buf_num} * \text{wifi_tx_pkt_size_max})$$

Generally, we do not need to care about the dynamic tx long buffers and dynamic tx long long buffers, because they are management frames which only have a small impact on the system.

4.14.18 Wi-Fi Menuconfig

Wi-Fi Buffer Configure

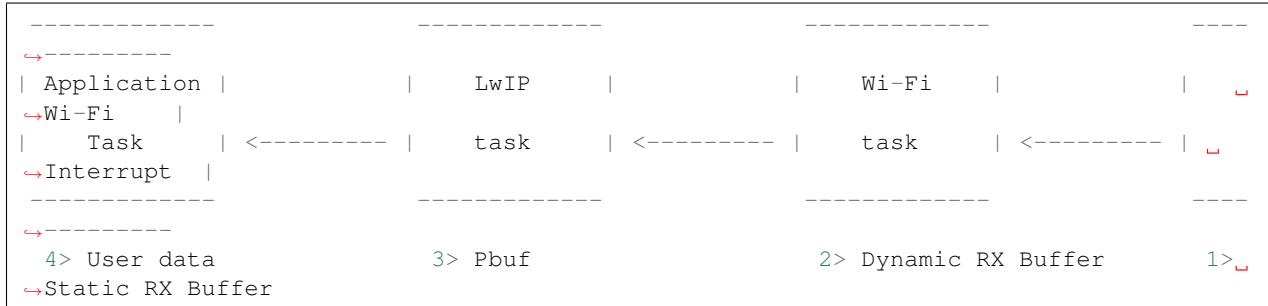
If you are going to modify the default number or type of buffer, it would be helpful to also have an overview of how the buffer is allocated/freed in the data path. The following diagram shows this process in the TX direction:



Description:

- The application allocates the data which needs to be sent out.
- The application calls 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:



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	<p>This is a kind of DMA memory. It is initialized in <code>esp_wifi_init()</code> and freed in <code>esp_wifi_deinit()</code>. The ‘Static Rx Buffer’ forms the hardware receiving list. Upon receiving a frame over the air, hardware writes the frame into the buffer and raises an interrupt to the CPU. Then, the Wi-Fi driver reads the content from the buffer and returns the buffer back to the list.</p>
Dynamic RX Buffer	Dynamic	32	Yes	<p>The buffer length is variable and it depends on the received frames’ length. When the Wi-Fi driver receives a frame from the ‘Hardware Rx Buffer’, the ‘Dynamic Rx Buffer’ needs to be allocated from the heap. The number of the Dynamic Rx Buffer, configured in the menuconfig, is used to limit the total un-freeed Dynamic Rx Buffer number.</p>
Dynamic TX Buffer	Dynamic	32	Yes	<p>This is a kind of DMA memory. It is allocated to the heap. When the upper-layer (LwIP) sends packets to the Wi-Fi driver, it firstly allocates a ‘Dynamic TX Buffer’ and makes a copy of the upper-layer buffer. The Dynamic and</p>
636				<p>Chapter 4. API Guides are mutually exclusive.</p>
Static TX Buffer	Static	32 * 1600Bytes	Yes	<p>This is a kind of DMA</p>

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.

CHAPTER 5

Contributions Guide

We welcome contributions to the esp-idf project!

5.1 How to Contribute

Contributions to esp-idf - fixing bugs, adding features, adding documentation - are welcome. We accept contributions via [Github Pull Requests](#).

5.2 Before Contributing

Before sending us a Pull Request, please consider this list of points:

- Is the contribution entirely your own work, or already licensed under an Apache License 2.0 compatible Open Source License? If not then we unfortunately cannot accept it.
- Does any new code conform to the esp-idf [Style Guide](#)?
- Does the code documentation follow requirements in [Documenting Code](#)?
- Is the code adequately commented for people to understand how it is structured?
- Is there documentation or examples that go with code contributions? There are additional suggestions for writing good examples in [examples](#) readme.
- Are comments and documentation written in clear English, with no spelling or grammar errors?
- If the contribution contains multiple commits, are they grouped together into logical changes (one major change per pull request)? Are any commits with names like “fixed typo” [squashed into previous commits](#)?
- If you’re unsure about any of these points, please open the Pull Request anyhow and then ask us for feedback.

5.3 Pull Request Process

After you open the Pull Request, there will probably be some discussion in the comments field of the request itself.

Once the Pull Request is ready to merge, it will first be merged into our internal git system for in-house automated testing.

If this process passes, it will be merged onto the public github repository.

5.4 Legal Part

Before a contribution can be accepted, you will need to sign our *Contributor Agreement*. You will be prompted for this automatically as part of the Pull Request process.

5.5 Related Documents

5.5.1 Espressif IoT Development Framework Style Guide

About this guide

Purpose of this style guide is to encourage use of common coding practices within the ESP-IDF.

Style guide is a set of rules which are aimed to help create readable, maintainable, and robust code. By writing code which looks the same way across the code base we help others read and comprehend the code. By using same conventions for spaces and newlines we reduce chances that future changes will produce huge unreadable diffs. By following common patterns for module structure and by using language features consistently we help others understand code behavior.

We try to keep rules simple enough, which means that they can not cover all potential cases. In some cases one has to bend these simple rules to achieve readability, maintainability, or robustness.

When doing modifications to third-party code used in ESP-IDF, follow the way that particular project is written. That will help propose useful changes for merging into upstream project.

C code formatting

Indentation

Use 4 spaces for each indentation level. Don't use tabs for indentation. Configure the editor to emit 4 spaces each time you press tab key.

Vertical space

Place one empty line between functions. Don't begin or end a function with an empty line.

```
void function1()
{
    do_one_thing();
    do_another_thing();
    // INCORRECT, don't place empty line here
```

```

}
                                // 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
```

Documenting code

Please see the guide here: [Documenting Code](#).

Structure and naming

Language features

To be written.

5.5.2 Documenting Code

The purpose of this description is to provide quick summary on documentation style used in `espressif/esp-idf` repository and how to add new documentation.

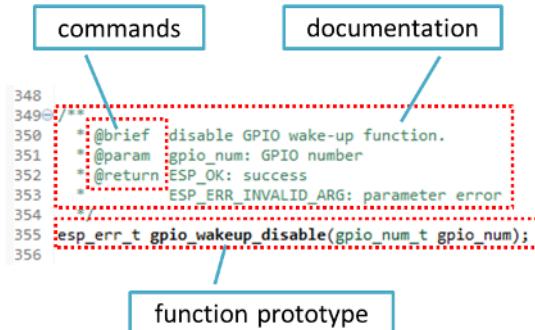
Introduction

When documenting code for this repository, please follow [Doxygen](#) style. You are doing it by inserting special commands, for instance `@param`, into standard comments blocks, for example:

```
/**  
 * @param ratio this is oxygen to air ratio  
 */
```

Doxxygen 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.

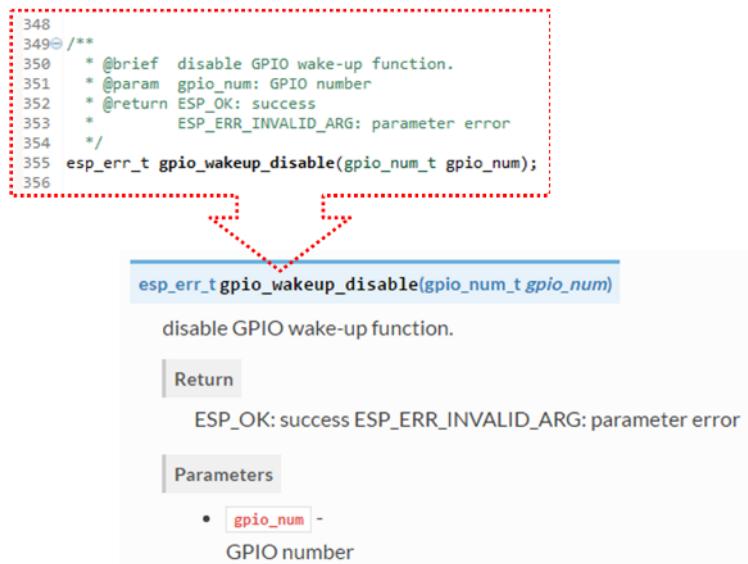


Doxxygen supports couple of formatting styles. It also gives you great flexibility on level of details to include in documentation. To get familiar with available features, please check data reach and very well organized [Doxxygen Manual](#).

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:

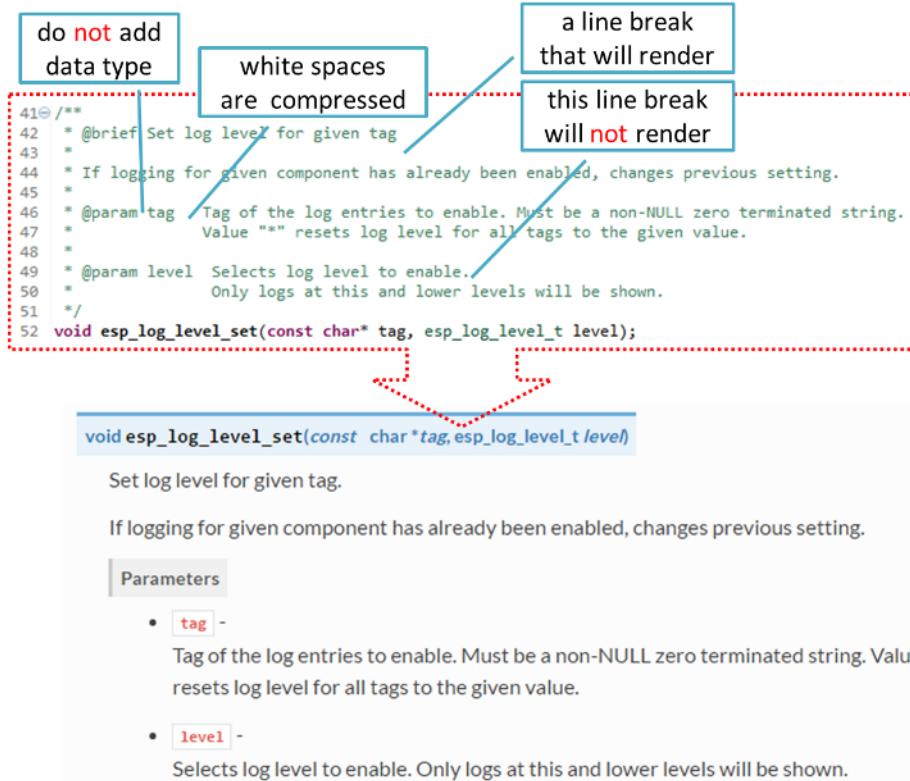


Go for it!

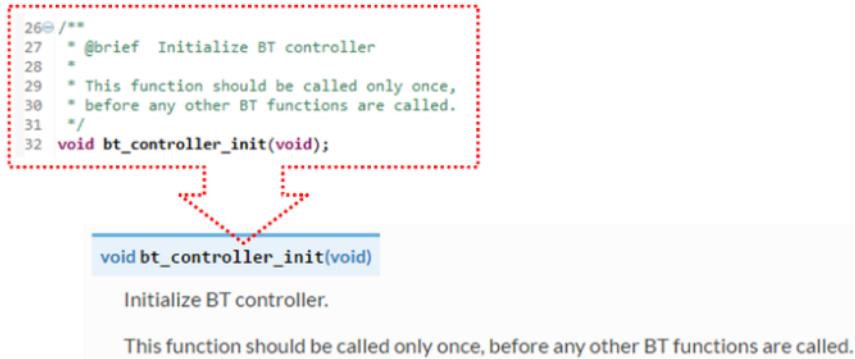
When writing code for this repository, please follow guidelines below.

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.

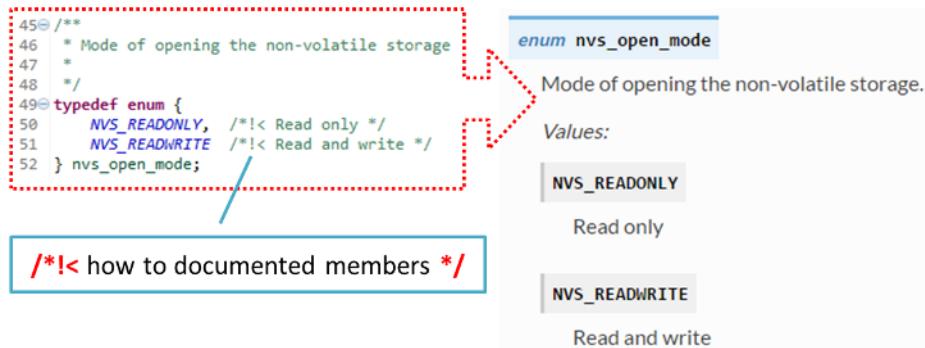
2. Documentation of function should describe what this function does. If it accepts input parameters and returns some value, all of them should be explained.
3. Do not add a data type before parameter or any other characters besides spaces. All spaces and line breaks are compressed into a single space. 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

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`.

Put it all together

Once documentation is ready, follow instruction in [API Documentation Template](#) and create a single file, that will merge all individual pieces of prepared documentation. Finally add a link to this file to respective `.. toctree::` in `index.rst` file located in `/docs` folder or subfolders.

OK, but I am new to Sphinx!

1. No worries. All the software you need is well documented. It is also open source and free. Start by checking Sphinx documentation. If you are not clear how to write using rst markup language, see [reStructuredText Primer](#).
2. Check the source files of this documentation to understand what is behind of what you see now on the screen. Sources are maintained on GitHub in [espressif/esp-idf](#) repository in `docs` folder. You can go directly to the source file of this page by scrolling up and clicking the link in the top right corner. When on GitHub, see what's really inside, open source files by clicking Raw button.
3. You will likely want to see how documentation builds and looks like before posting it on the GitHub. There are two options to do so:
 - Install [Sphinx](#), [Breathe](#) and [Doxygen](#) to build it locally, see chapter below.
 - Set up an account on [Read the Docs](#) and build documentation in the cloud. Read the Docs provides document building and hosting for free and their service works really quick and great.
4. To preview documentation before building use [Sublime Text](#) editor together with [OmniMarkupPreviewer](#) plugin.

Setup for building documentation locally

You can setup environment to build documentation locally on your PC by installing:

1. Doxygen - <https://www.stack.nl/~dimitri/doxygen/>
2. Sphinx - <https://github.com/sphinx-doc/sphinx/#readme-for-sphinx>
3. Document theme “`sphinx_rtd_theme`” - https://github.com/rtfd/sphinx_rtd_theme
4. Breathe - <https://github.com/michaeljones/breathe#breathe>

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.

Installation of Doxygen is OS dependent:

Linux

```
sudo apt-get install doxygen
```

Windows - install in MSYS2 console

```
pacman -S doxygen
```

MacOS

```
brew install doxygen
```

All remaining applications are [Python](#) packages and you can install them in one step as follows:

```
cd ~/esp/esp-idf/docs  
pip install -r requirements.txt
```

Note: Installation steps assume that ESP-IDF is placed in `~/esp/esp-idf` directory, that is default location of ESP-IDF used in documentation.

Now you should be ready to build documentation by invoking:

```
make html
```

This may take couple of minutes. After completion, documentation will be placed in `~/esp/esp-idf/docs/_build/html` folder. To see it, open `index.html` in a web browser.

Wrap up

We love good code that is doing cool things. We love it even better, if it is well documented, so we can quickly make it run and also do the cool things.

Go ahead, contribute your code and documentation!

Related Documents

- [API Documentation Template](#)

5.5.3 API Documentation Template

Note: INSTRUCTIONS

1. Use this file ([docs/api-reference/template.rst](#)) as a template to document API.
 2. Change the file name to the name of the header file that represents documented API.
 3. Include respective files with descriptions from the API folder using `.. include::`:
 - `README.rst`
 - `example.rst`
 - `...`
 4. Optionally provide description right in this file.
 5. Once done, remove all instructions like this one and any superfluous headers.
-

Overview

Note: INSTRUCTIONS

1. Provide overview where and how this API may be used.
2. Where applicable include code snippets to illustrate functionality of particular functions.

- 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:

4. The *.inc files contain formatted reference of API members generated automatically on each documentation build. All *.inc files are placed in Sphinx_build directory. To see directives generated for e.g. esp_wifi.h, run python gen-dxd.py esp32/include/esp_wifi.h.

5. To show contents of *.inc file in documentation, include it as follows:

```
.. include:: ../../../_build/inc/esp_wifi.inc
```

For example see [docs/api-reference/wifi/esp_wifi.rst](#)

6. Optionally, rather than using *.inc files, you may want to describe API in your own way. See [docs/api-reference/system/deep_sleep.rst](#) for example.

Below is the list of common .. doxygen...:: directives:

- Functions - .. doxygenfunction:: name_of_function
- Unions - .. doxygenunion:: name_of_union
- Structures - .. doxygenstruct:: name_of_structure together with :members:
- Macros - .. doxygendefine:: name_of_define
- Type Definitions - .. doxygentypedef:: name_of_type
- Enumerations - .. doxygenenum:: name_of_enumeration

See [Breathe documentation](#) for additional information.

To provide a link to header file, use the [*link custom role*](#) as follows:

```
* :component_file:`path_to/header_file.h`
```

7. In any case, to generate API reference, the file [docs/Doxyfile](#) should be updated with paths to *.h headers that are being documented.
8. When changes are committed and documentation is built, check how this section has been rendered. [*Correct annotations*](#) in respective header files, if required.

5.5.4 Contributor Agreement

Individual Contributor Non-Exclusive License Agreement

including the Traditional Patent License OPTION

Thank you for your interest in contributing to Espressif IoT Development Framework (esp-idf) (“We” or “Us”).

The purpose of this contributor agreement (“Agreement”) is to clarify and document the rights granted by contributors to Us. To make this document effective, please follow the instructions at [CONTRIBUTING.rst](#)

1. DEFINITIONS

“You” means the Individual Copyright owner who submits a Contribution to Us. If You are an employee and submit the Contribution as part of your employment, You have had Your employer approve this Agreement or sign the Entity version of this document.

“Contribution” means any original work of authorship (software and/or documentation) including any modifications or additions to an existing work, Submitted by You to Us, in which You own the Copyright. If You do not own the Copyright in the entire work of authorship, please contact Us at angus@espressif.com.

“Copyright” means all rights protecting works of authorship owned or controlled by You, including copyright, moral and neighboring rights, as appropriate, for the full term of their existence including any extensions by You.

“Material” means the software or documentation made available by Us to third parties. When this Agreement covers more than one software project, the Material means the software or documentation to which the Contribution was Submitted. After You Submit the Contribution, it may be included in the Material.

“Submit” means any form of physical, electronic, or written communication sent to Us, including but not limited to electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, Us, but excluding communication that is conspicuously marked or otherwise designated in writing by You as “Not a Contribution.”

“Submission Date” means the date You Submit a Contribution to Us.

“Documentation” means any non-software portion of a Contribution.

2. LICENSE GRANT

2.1 Copyright License to Us

Subject to the terms and conditions of this Agreement, You hereby grant to Us a worldwide, royalty-free, NON-exclusive, perpetual and irrevocable license, with the right to transfer an unlimited number of non-exclusive licenses or to grant sublicenses to third parties, under the Copyright covering the Contribution to use the Contribution by all means, including, but not limited to:

- to publish the Contribution,
- to modify the Contribution, to prepare derivative works based upon or containing the Contribution and to combine the Contribution with other software code,
- to reproduce the Contribution in original or modified form,
- to distribute, to make the Contribution available to the public, display and publicly perform the Contribution in original or modified form.

2.2 Moral Rights remain unaffected to the extent they are recognized and not waivable by applicable law. Notwithstanding, You may add your name in the header of the source code files of Your Contribution and We will respect this attribution when using Your Contribution.

3. PATENTS

3.1 Patent License

Subject to the terms and conditions of this Agreement You hereby grant to us a worldwide, royalty-free, non-exclusive, perpetual and irrevocable (except as stated in Section 3.2) patent license, with the right to transfer an unlimited number of non-exclusive licenses or to grant sublicenses to third parties, to make, have made, use, sell, offer for sale, import and otherwise transfer the Contribution and the Contribution in combination with the Material (and portions of such combination). This license applies to all patents owned or controlled by You, whether already acquired or hereafter acquired, that would be infringed by making, having made, using, selling, offering for sale, importing or otherwise transferring of Your Contribution(s) alone or by combination of Your Contribution(s) with the Material.

3.2 Revocation of Patent License

You reserve the right to revoke the patent license stated in section 3.1 if we make any infringement claim that is targeted at your Contribution and not asserted for a Defensive Purpose. An assertion of claims of the Patents shall be considered for a “Defensive Purpose” if the claims are asserted against an entity that has filed, maintained, threatened, or voluntarily participated in a patent infringement lawsuit against Us or any of Our licensees.

4. DISCLAIMER

THE CONTRIBUTION IS PROVIDED “AS IS”. MORE PARTICULARLY, ALL EXPRESS OR IMPLIED WARRANTIES INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED BY YOU TO US AND BY US TO YOU. TO THE EXTENT THAT ANY SUCH WARRANTIES CANNOT BE DISCLAIMED, SUCH WARRANTY IS LIMITED IN DURATION TO THE MINIMUM PERIOD PERMITTED BY LAW.

5. Consequential Damage Waiver

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL YOU OR US BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF ANTICIPATED SAVINGS, LOSS OF DATA, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL AND EXEMPLARY DAMAGES ARISING OUT OF THIS AGREEMENT REGARDLESS OF THE LEGAL OR EQUITABLE THEORY (CONTRACT, TORT OR OTHERWISE) UPON WHICH THE CLAIM IS BASED.

6. Approximation of Disclaimer and Damage Waiver

IF THE DISCLAIMER AND DAMAGE WAIVER MENTIONED IN SECTION 4 AND SECTION 5 CANNOT BE GIVEN LEGAL EFFECT UNDER APPLICABLE LOCAL LAW, REVIEWING COURTS SHALL APPLY LOCAL LAW THAT MOST CLOSELY APPROXIMATES AN ABSOLUTE WAIVER OF ALL CIVIL LIABILITY IN CONNECTION WITH THE CONTRIBUTION.

7. Term

7.1 This Agreement shall come into effect upon Your acceptance of the terms and conditions.

7.2 In the event of a termination of this Agreement Sections 4, 5, 6, 7 and 8 shall survive such termination and shall remain in full force thereafter. For the avoidance of doubt, Contributions that are already licensed under a free and open source license at the date of the termination shall remain in full force after the termination of this Agreement.

8. Miscellaneous

8.1 This Agreement and all disputes, claims, actions, suits or other proceedings arising out of this agreement or relating in any way to it shall be governed by the laws of People’s Republic of China excluding its private international law provisions.

8.2 This Agreement sets out the entire agreement between You and Us for Your Contributions to Us and overrides all other agreements or understandings.

8.3 If any provision of this Agreement is found void and unenforceable, such provision will be replaced to the extent possible with a provision that comes closest to the meaning of the original provision and that is enforceable. The terms and conditions set forth in this Agreement shall apply notwithstanding any failure of essential purpose of this Agreement or any limited remedy to the maximum extent possible under law.

8.4 You agree to notify Us of any facts or circumstances of which you become aware that would make this Agreement inaccurate in any respect.

You

Date:	<input type="text"/>
Name:	<input type="text"/>
Title:	<input type="text"/>
Address:	<input type="text"/>

Us

Date:	<input type="text"/>
Name:	<input type="text"/>
Title:	<input type="text"/>
Address:	<input type="text"/>

CHAPTER 6

Resources

- The [esp32.com forum](#) is a place to ask questions and find community resources.
- Check the [Issues](#) section on GitHub if you find a bug or have a feature request. Please check existing [Issues](#) before opening a new one.
- If you're interested in contributing to ESP-IDF, please check the [Contributions Guide](#).
- To develop applications using Arduino platform, refer to [Arduino core for ESP32 WiFi chip](#).
- For additional ESP32 product related information, please refer to [documentation](#) section of [Espressif](#) site.
- Mirror of this documentation is available under: <https://dl.espressif.com/doc/esp-idf/latest/>.

Copyrights and Licenses

7.1 Software Copyrights

All original source code in this repository is Copyright (C) 2015-2016 Espressif Systems. This source code is licensed under the Apache License 2.0 as described in the file LICENSE.

Additional third party copyrighted code is included under the following licenses:

- [Newlib](#) (components/newlib) is licensed under the BSD License and is Copyright of various parties, as described in the file components/newlib/COPYING.NEWLIB.
- Xtensa header files (components/esp32/include/xtensa) are Copyright (C) 2013 Tensilica Inc and are licensed under the MIT License as reproduced in the individual header files.
- [esptool.py](#) (components/esptool_py/esptool) is Copyright (C) 2014-2016 Fredrik Ahlberg, Angus Gratton and is licensed under the GNU General Public License v2, as described in the file components/esptool_py/LICENSE.
- Original parts of [FreeRTOS](#) (components/freertos) are Copyright (C) 2015 Real Time Engineers Ltd and is licensed under the GNU General Public License V2 with the FreeRTOS Linking Exception, as described in the file components/freertos/license.txt.
- Original parts of [LWIP](#) (components/lwip) are Copyright (C) 2001, 2002 Swedish Institute of Computer Science and are licensed under the BSD License as described in the file components/lwip/COPYING.
- KConfig (tools/kconfig) is Copyright (C) 2002 Roman Zippel and others, and is licensed under the GNU General Public License V2.
- [wpa_supplicant](#) Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- [FreeBSD net80211](#) Copyright (c) 2004-2008 Sam Leffler, Erno Consulting and licensed under the BSD license.
- [JSMN](#) JSON Parser (components/jsmn) Copyright (c) 2010 Serge A. Zaitsev and licensed under the MIT license.
- [argtable3](#) argument parsing library Copyright (C) 1998-2001,2003-2011,2013 Stewart Heitmann and licensed under 3-clause BSD license.
- [linenoise](#) line editing library Copyright (c) 2010-2014, Salvatore Sanfilippo <antirez at gmail dot com>, Copyright (c) 2010-2013, Pieter Noordhuis <pnoordhuis at gmail dot com>, licensed under 2-clause BSD license.

Where source code headers specify Copyright & License information, this information takes precedence over the summaries made here.

7.2 ROM Source Code Copyrights

ESP32 mask ROM hardware includes binaries compiled from portions of the following third party software:

- [Newlib](#), as licensed under the BSD License and Copyright of various parties, as described in the file components/newlib/COPYING.NEGLIB.
- 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.

7.3 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.

7.4 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.

7.5 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 8

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.

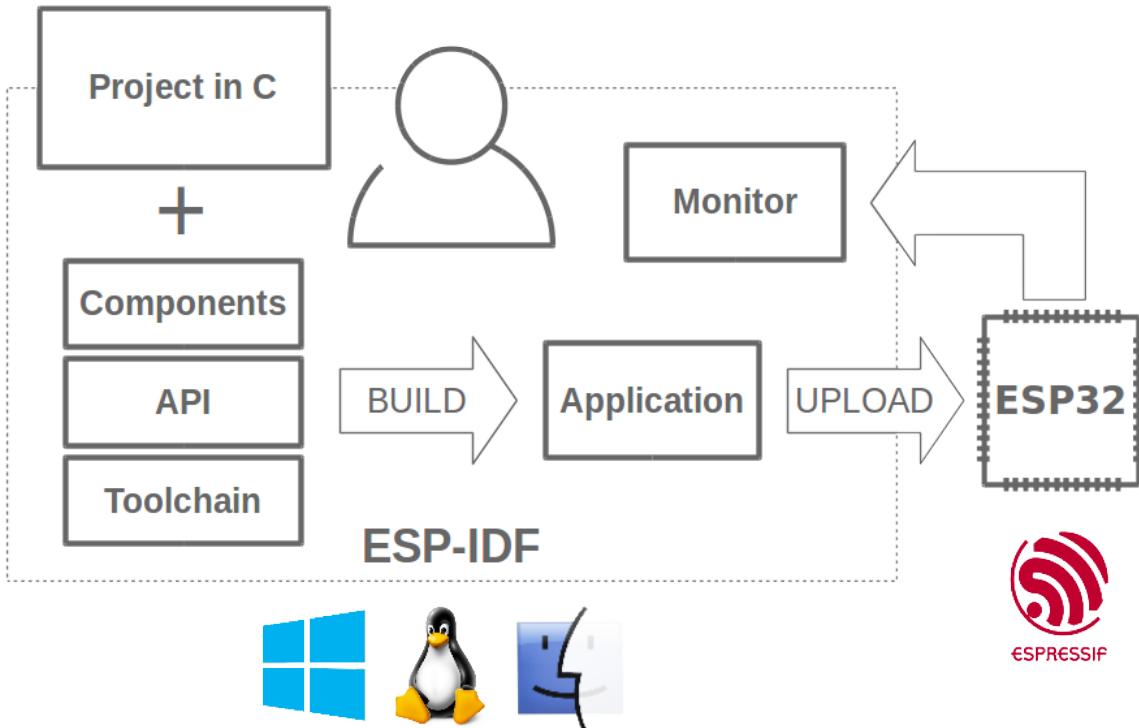


Fig. 8.1: Espressif IoT Integrated Development Framework

The ESP-IDF, Espressif IoT Integrated Development Framework, provides toolchain, API, components and workflows to develop applications for ESP32 using Windows, Linux and Mac OS operating systems.

- genindex

A

ADC1_CHANNEL_0 (C++ class), 166
ADC1_CHANNEL_0_GPIO_NUM (C macro), 167
ADC1_CHANNEL_1 (C++ class), 166
ADC1_CHANNEL_1_GPIO_NUM (C macro), 167
ADC1_CHANNEL_2 (C++ class), 166
ADC1_CHANNEL_2_GPIO_NUM (C macro), 168
ADC1_CHANNEL_3 (C++ class), 166
ADC1_CHANNEL_3_GPIO_NUM (C macro), 168
ADC1_CHANNEL_4 (C++ class), 166
ADC1_CHANNEL_4_GPIO_NUM (C macro), 168
ADC1_CHANNEL_5 (C++ class), 166
ADC1_CHANNEL_5_GPIO_NUM (C macro), 168
ADC1_CHANNEL_6 (C++ class), 166
ADC1_CHANNEL_6_GPIO_NUM (C macro), 168
ADC1_CHANNEL_7 (C++ class), 166
ADC1_CHANNEL_7_GPIO_NUM (C macro), 168
ADC1_CHANNEL_MAX (C++ class), 167
adc1_channel_t (C++ type), 166
adc1_config_channel_atten (C++ function), 164
adc1_config_width (C++ function), 164
adc1_get_raw (C++ function), 165
ADC1_GPIO32_CHANNEL (C macro), 168
ADC1_GPIO33_CHANNEL (C macro), 168
ADC1_GPIO34_CHANNEL (C macro), 168
ADC1_GPIO35_CHANNEL (C macro), 168
ADC1_GPIO36_CHANNEL (C macro), 167
ADC1_GPIO37_CHANNEL (C macro), 167
ADC1_GPIO38_CHANNEL (C macro), 168
ADC1_GPIO39_CHANNEL (C macro), 168
adc1_to_voltage (C++ function), 170
adc1_ulp_enable (C++ function), 165
ADC2_CHANNEL_0 (C++ class), 167
ADC2_CHANNEL_0_GPIO_NUM (C macro), 168
ADC2_CHANNEL_1 (C++ class), 167
ADC2_CHANNEL_1_GPIO_NUM (C macro), 168
ADC2_CHANNEL_2 (C++ class), 167
ADC2_CHANNEL_2_GPIO_NUM (C macro), 168
ADC2_CHANNEL_3 (C++ class), 167
ADC2_CHANNEL_3_GPIO_NUM (C macro), 168
ADC2_CHANNEL_4 (C++ class), 167
ADC2_CHANNEL_4_GPIO_NUM (C macro), 168
ADC2_CHANNEL_5 (C++ class), 167
ADC2_CHANNEL_5_GPIO_NUM (C macro), 168
ADC2_CHANNEL_6 (C++ class), 167
ADC2_CHANNEL_6_GPIO_NUM (C macro), 168
ADC2_CHANNEL_7 (C++ class), 167
ADC2_CHANNEL_7_GPIO_NUM (C macro), 168
ADC2_CHANNEL_8 (C++ class), 167
ADC2_CHANNEL_8_GPIO_NUM (C macro), 168
ADC2_CHANNEL_9 (C++ class), 167
ADC2_CHANNEL_9_GPIO_NUM (C macro), 168
ADC2_CHANNEL_MAX (C++ class), 167
adc2_channel_t (C++ type), 167
ADC2_GPIO0_CHANNEL (C macro), 168
ADC2_GPIO12_CHANNEL (C macro), 168
ADC2_GPIO13_CHANNEL (C macro), 168
ADC2_GPIO14_CHANNEL (C macro), 168
ADC2_GPIO15_CHANNEL (C macro), 168
ADC2_GPIO25_CHANNEL (C macro), 168
ADC2_GPIO26_CHANNEL (C macro), 168
ADC2_GPIO27_CHANNEL (C macro), 168
ADC2_GPIO2_CHANNEL (C macro), 168
ADC2_GPIO4_CHANNEL (C macro), 168
adc2_vref_to_gpio (C++ function), 165
ADC_ATTEN_0db (C++ class), 166
ADC_ATTEN_11db (C++ class), 166
ADC_ATTEN_2_5db (C++ class), 166
ADC_ATTEN_6db (C++ class), 166
adc_atten_t (C++ type), 166
adc_bits_width_t (C++ type), 166
ADC_WIDTH_10Bit (C++ class), 166
ADC_WIDTH_11Bit (C++ class), 166
ADC_WIDTH_12Bit (C++ class), 166
ADC_WIDTH_9Bit (C++ class), 166
ADV_CHNL_37 (C++ class), 96
ADV_CHNL_38 (C++ class), 97
ADV_CHNL_39 (C++ class), 97
ADV_CHNL_ALL (C++ class), 97

ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY
 (C++ class), 97
 ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST
 (C++ class), 97
 ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY
 (C++ class), 97
 ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST
 (C++ class), 97
 ADV_TYPE_DIRECT_IND_HIGH (C++ class), 96
 ADV_TYPE_DIRECT_IND_LOW (C++ class), 96
 ADV_TYPE_IND (C++ class), 96
 ADV_TYPE_NONCONN_IND (C++ class), 96
 ADV_TYPE_SCAN_IND (C++ class), 96

B

BLE_ADDR_TYPE_PUBLIC (C++ class), 74
 BLE_ADDR_TYPE_RANDOM (C++ class), 74
 BLE_ADDR_TYPE_RPA_PUBLIC (C++ class), 74
 BLE_ADDR_TYPE_RPA_RANDOM (C++ class), 74
 BLE_SCAN_FILTER_ALLOW_ALL (C++ class), 97
 BLE_SCAN_FILTER_ALLOW_ONLY_WLST (C++ class), 98
 BLE_SCAN_FILTER_ALLOW_UND_RPA_DIR (C++ class), 98
 BLE_SCAN_FILTER_ALLOW_WLIST_PRA_DIR (C++ class), 98
 BLE_SCAN_TYPE_ACTIVE (C++ class), 97
 BLE_SCAN_TYPE_PASSIVE (C++ class), 97
 BT_CONTROLLER_INIT_CONFIG_DEFAULT
 macro), 70

C

CONFIG_HEAP_TRACING_STACK_DEPTH
 macro), 388

D

DAC_CHANNEL_1 (C++ class), 173
 DAC_CHANNEL_1_GPIO_NUM (C macro), 174
 DAC_CHANNEL_2 (C++ class), 173
 DAC_CHANNEL_2_GPIO_NUM (C macro), 174
 DAC_CHANNEL_MAX (C++ class), 173
 dac_channel_t (C++ type), 173
 DAC_GPIO25_CHANNEL (C macro), 174
 DAC_GPIO26_CHANNEL (C macro), 174
 dac_i2s_disable (C++ function), 173
 dac_i2s_enable (C++ function), 173
 dac_output_disable (C++ function), 173
 dac_output_enable (C++ function), 173
 dac_output_voltage (C++ function), 172
 dmaworkaround_cb_t (C++ type), 272

E

ESP_A2D_AUDIO_CFG_EVT (C++ class), 152

ESP_A2D_AUDIO_STATE_EVT (C++ class), 152
 ESP_A2D_AUDIO_STATE_REMOTE_SUSPEND
 (C++ class), 152
 ESP_A2D_AUDIO_STATE_STARTED (C++ class), 152
 ESP_A2D_AUDIO_STATE_STOPPED (C++ class), 152
 esp_a2d_audio_state_t (C++ type), 152
 esp_a2d_cb_event_t (C++ type), 152
 esp_a2d_cb_param_t (C++ type), 149
 esp_a2d_cb_param_t::a2d_audio_cfg_param (C++
 class), 149
 esp_a2d_cb_param_t::a2d_audio_cfg_param::mcc (C++
 member), 150
 esp_a2d_cb_param_t::a2d_audio_cfg_param::remote_bda
 (C++ member), 150
 esp_a2d_cb_param_t::a2d_audio_stat_param (C++
 class), 150
 esp_a2d_cb_param_t::a2d_audio_stat_param::remote_bda
 (C++ member), 150
 esp_a2d_cb_param_t::a2d_audio_stat_param::state (C++
 member), 150
 esp_a2d_cb_param_t::a2d_conn_stat_param (C++ class),
 150
 esp_a2d_cb_param_t::a2d_conn_stat_param::disc_rsn
 (C++ member), 150
 esp_a2d_cb_param_t::a2d_conn_stat_param::remote_bda
 (C++ member), 150
 esp_a2d_cb_param_t::a2d_conn_stat_param::state (C++
 member), 150
 (C) esp_a2d_cb_param_t::audio_cfg (C++ member), 149
 esp_a2d_cb_param_t::audio_stat (C++ member), 149
 esp_a2d_cb_param_t::conn_stat (C++ member), 149
 esp_a2d_cb_t (C++ type), 151
 ESP_A2D_CIE_LEN_ATRAC (C macro), 151
 ESP_A2D_CIE_LEN_M12 (C macro), 151
 ESP_A2D_CIE_LEN_M24 (C macro), 151
 ESP_A2D_CIE_LEN_SBC (C macro), 151
 ESP_A2D_CONNECTION_STATE_CONNECTED
 (C++ class), 151
 ESP_A2D_CONNECTION_STATE_CONNECTING
 (C++ class), 151
 ESP_A2D_CONNECTION_STATE_DISCONNECTED
 (C++ class), 151
 ESP_A2D_CONNECTION_STATE_DISCONNECTING
 (C++ class), 151
 ESP_A2D_CONNECTION_STATE_EVT (C++ class),
 152
 esp_a2d_connection_state_t (C++ type), 151
 esp_a2d_data_cb_t (C++ type), 151
 ESP_A2D_DISC_RSN_ABNORMAL (C++ class), 152
 ESP_A2D_DISC_RSN_NORMAL (C++ class), 152
 esp_a2d_disc_rsn_t (C++ type), 151
 esp_a2d_mcc_t (C++ class), 150
 esp_a2d_mcc_t::type (C++ member), 150
 ESP_A2D_MCT_ATRAC (C macro), 151

ESP_A2D_MCT_M12 (C macro), 150
 ESP_A2D_MCT_M24 (C macro), 151
 ESP_A2D_MCT_NON_A2DP (C macro), 151
 ESP_A2D_MCT_SBC (C macro), 150
 esp_a2d_mct_t (C++ type), 151
 esp_a2d_register_callback (C++ function), 148
 esp_a2d_register_data_callback (C++ function), 148
 esp_a2d_sink_connect (C++ function), 149
 esp_a2d_sink_deinit (C++ function), 148
 esp_a2d_sink_disconnect (C++ function), 149
 esp_a2d_sink_init (C++ function), 148
 esp_adc_cal_characteristics_t (C++ class), 171
 esp_adc_cal_characteristics_t::bit_width (C++ member), 172
 esp_adc_cal_characteristics_t::gain (C++ member), 171
 esp_adc_cal_characteristics_t::ideal_offset (C++ member), 172
 esp_adc_cal_characteristics_t::offset (C++ member), 171
 esp_adc_cal_characteristics_t::table (C++ member), 172
 esp_adc_cal_characteristics_t::v_ref (C++ member), 171
 esp_adc_cal_get_characteristics (C++ function), 170
 esp_adc_cal_lookup_table_t (C++ class), 171
 esp_adc_cal_lookup_table_t::bit_shift (C++ member), 171
 esp_adc_cal_lookup_table_t::gain_c (C++ member), 171
 esp_adc_cal_lookup_table_t::gain_m (C++ member), 171
 esp_adc_cal_lookup_table_t::offset_c (C++ member), 171
 esp_adc_cal_lookup_table_t::offset_m (C++ member), 171
 esp_adc_cal_lookup_table_t::voltage (C++ member), 171
 esp_adc_raw_to_voltage (C++ function), 170
 ESP_APP_ID_MAX (C macro), 73
 ESP_APP_ID_MIN (C macro), 73
 esp_apptrace_buffer_get (C++ function), 413
 esp_apptrace_buffer_put (C++ function), 413
 esp_apptrace_dest_t (C++ type), 415
 ESP_APPTRACE_DEST_TRAX (C++ class), 415
 ESP_APPTRACE_DEST_UART0 (C++ class), 415
 esp_apptrace_down_buffer_config (C++ function), 413
 esp_apptrace_down_buffer_get (C++ function), 415
 esp_apptrace_down_buffer_put (C++ function), 415
 esp_apptrace_flush (C++ function), 414
 esp_apptrace_flush_nolock (C++ function), 414
 esp_apptrace_init (C++ function), 413
 esp_apptrace_read (C++ function), 414
 esp_apptrace_vprintf (C++ function), 414
 esp_apptrace_vprintf_to (C++ function), 414
 esp_apptrace_write (C++ function), 413
 esp_attr_control_t (C++ class), 100
 esp_attr_control_t::auto_rsp (C++ member), 100
 esp_attr_desc_t (C++ class), 100
 esp_attr_desc_t::length (C++ member), 100
 esp_attr_desc_t::max_length (C++ member), 100
 esp_attr_desc_t::perm (C++ member), 100
 esp_attr_desc_t::uuid_length (C++ member), 100
 esp_attr_desc_t::uuid_p (C++ member), 100
 esp_attr_desc_t::value (C++ member), 100
 esp_attr_value_t (C++ class), 100
 esp_attr_value_t::attr_len (C++ member), 101
 esp_attr_value_t::attr_max_len (C++ member), 101
 esp_attr_value_t::attr_value (C++ member), 101
 esp_avrc_ct_cb_event_t (C++ type), 155
 esp_avrc_ct_cb_param_t (C++ type), 154
 esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param (C++ class), 154
 esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param::connected (C++ member), 154
 esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param::feat_mask (C++ member), 154
 esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param::remote_bda (C++ member), 154
 esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ class), 154
 esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param::key_code (C++ member), 154
 esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param::key_state (C++ member), 154
 esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param::tl (C++ member), 154
 esp_avrc_ct_cb_param_t::conn_stat (C++ member), 154
 esp_avrc_ct_cb_param_t::psth_rsp (C++ member), 154
 esp_avrc_ct_cb_t (C++ type), 154
 ESP_AVRC_CT_CONNECTION_STATE_EVT (C++ class), 155
 esp_avrc_ct_deinit (C++ function), 153
 esp_avrc_ct_init (C++ function), 153
 ESP_AVRC_CT_MAX_EVT (C++ class), 156
 ESP_AVRC_CT_PASSTHROUGH_RSP_EVT (C++ class), 155
 esp_avrc_ct_register_callback (C++ function), 153
 esp_avrc_ct_send_passthrough_cmd (C++ function), 153
 ESP_AVRC_FEAT_ADV_CTRL (C++ class), 155
 ESP_AVRC_FEAT_BROWSE (C++ class), 155
 ESP_AVRC_FEAT_META_DATA (C++ class), 155
 ESP_AVRC_FEAT_RCCT (C++ class), 155
 ESP_AVRC_FEAT_RCTG (C++ class), 155
 ESP_AVRC_FEAT_VENDOR (C++ class), 155
 esp_avrc_features_t (C++ type), 155
 ESP_AVRC_PT_CMD_BACKWARD (C++ class), 155
 ESP_AVRC_PT_CMD_FORWARD (C++ class), 155
 ESP_AVRC_PT_CMD_PAUSE (C++ class), 155
 ESP_AVRC_PT_CMD_PLAY (C++ class), 155
 ESP_AVRC_PT_CMD_STATE_PRESSED (C++ class), 155
 ESP_AVRC_PT_CMD_STATE_RELEASED (C++ class), 155

esp_avrc_pt_cmd_state_t (C++ type), 155
ESP_AVRC_PT_CMD_STOP (C++ class), 155
esp_avrc_pt_cmd_t (C++ type), 155
esp_base_mac_addr_set (C++ function), 412
ESP_BD_ADDR_HEX (C macro), 73
ESP_BD_ADDR_LEN (C macro), 73
ESP_BD_ADDR_STR (C macro), 73
esp_bd_addr_t (C++ type), 73
ESP_BLE_AD_MANUFACTURER_SPECIFIC_TYPE (C++ class), 96
ESP_BLE_AD_TYPE_128SERVICE_DATA (C++ class), 96
ESP_BLE_AD_TYPE_128SOL_SRV_UUID (C++ class), 96
ESP_BLE_AD_TYPE_128SRV_CMPL (C++ class), 95
ESP_BLE_AD_TYPE_128SRV_PART (C++ class), 95
ESP_BLE_AD_TYPE_16SRV_CMPL (C++ class), 95
ESP_BLE_AD_TYPE_16SRV_PART (C++ class), 95
ESP_BLE_AD_TYPE_32SERVICE_DATA (C++ class), 96
ESP_BLE_AD_TYPE_32SOL_SRV_UUID (C++ class), 96
ESP_BLE_AD_TYPE_32SRV_CMPL (C++ class), 95
ESP_BLE_AD_TYPE_32SRV_PART (C++ class), 95
ESP_BLE_AD_TYPE_ADV_INT (C++ class), 96
ESP_BLE_AD_TYPE_APPEARANCE (C++ class), 96
ESP_BLE_AD_TYPE_CHAN_MAP_UPDATE (C++ class), 96
ESP_BLE_AD_TYPE_DEV_CLASS (C++ class), 95
ESP_BLE_AD_TYPE_FLAG (C++ class), 95
ESP_BLE_AD_TYPE_INDOOR_POSITION (C++ class), 96
ESP_BLE_AD_TYPE_INT_RANGE (C++ class), 96
ESP_BLE_AD_TYPE_LE_DEV_ADDR (C++ class), 96
ESP_BLE_AD_TYPE_LE_ROLE (C++ class), 96
ESP_BLE_AD_TYPE_LE_SECURE_CONFIRM (C++ class), 96
ESP_BLE_AD_TYPE_LE_SECURE_RANDOM (C++ class), 96
ESP_BLE_AD_TYPE_LE_SUPPORT_FEATURE (C++ class), 96
ESP_BLE_AD_TYPE_NAME_CMPL (C++ class), 95
ESP_BLE_AD_TYPE_NAME_SHORT (C++ class), 95
ESP_BLE_AD_TYPE_PUBLIC_TARGET (C++ class), 96
ESP_BLE_AD_TYPE_RANDOM_TARGET (C++ class), 96
ESP_BLE_AD_TYPE_SERVICE_DATA (C++ class), 96
ESP_BLE_AD_TYPE_SM_OOB_FLAG (C++ class), 96
ESP_BLE_AD_TYPE_SM_TK (C++ class), 96
ESP_BLE_AD_TYPE_SOL_SRV_UUID (C++ class), 96
ESP_BLE_AD_TYPE_SPAIR_C256 (C++ class), 96
ESP_BLE_AD_TYPE_SPAIR_R256 (C++ class), 96
ESP_BLE_AD_TYPE_TRANS_DISC_DATA (C++ class), 96
ESP_BLE_AD_TYPE_TX_PWR (C++ class), 95
ESP_BLE_AD_TYPE_URI (C++ class), 96
esp_ble_addr_type_t (C++ type), 74
esp_ble_adv_channel_t (C++ type), 96
ESP_BLE_ADV_DATA_LEN_MAX (C macro), 93
esp_ble_adv_data_t (C++ class), 87
esp_ble_adv_data_t::appearance (C++ member), 88
esp_ble_adv_data_t::flag (C++ member), 88
esp_ble_adv_data_t::include_name (C++ member), 88
esp_ble_adv_data_t::include_txpower (C++ member), 88
esp_ble_adv_data_t::manufacturer_len (C++ member), 88
esp_ble_adv_data_t::max_interval (C++ member), 88
esp_ble_adv_data_t::min_interval (C++ member), 88
esp_ble_adv_data_t::p_manufacturer_data (C++ member), 88
esp_ble_adv_data_t::p_service_data (C++ member), 88
esp_ble_adv_data_t::p_service_uuid (C++ member), 88
esp_ble_adv_data_t::service_data_len (C++ member), 88
esp_ble_adv_data_t::service_uuid_len (C++ member), 88
esp_ble_adv_data_t::set_scan_rsp (C++ member), 88
esp_ble_adv_data_type (C++ type), 95
esp_ble_adv_filter_t (C++ type), 97
ESP_BLE_ADV_FLAG_BREDR_NOT_SPT (C macro), 93
ESP_BLE_ADV_FLAG_DMT_CONTROLLER_SPT (C macro), 93
ESP_BLE_ADV_FLAG_DMT_HOST_SPT (C macro), 93
ESP_BLE_ADV_FLAG_GEN_DISC (C macro), 93
ESP_BLE_ADV_FLAG_LIMIT_DISC (C macro), 93
ESP_BLE_ADV_FLAG_NON_LIMIT_DISC (C macro), 93
esp_ble_adv_params_t (C++ class), 87
esp_ble_adv_params_t::adv_filter_policy (C++ member), 87
esp_ble_adv_params_t::adv_int_max (C++ member), 87
esp_ble_adv_params_t::adv_int_min (C++ member), 87
esp_ble_adv_params_t::adv_type (C++ member), 87
esp_ble_adv_params_t::channel_map (C++ member), 87
esp_ble_adv_params_t::own_addr_type (C++ member), 87
esp_ble_adv_params_t::peer_addr (C++ member), 87
esp_ble_adv_params_t::peer_addr_type (C++ member), 87
esp_ble_adv_type_t (C++ type), 96
esp_ble_auth_cmpl_t (C++ class), 92
esp_ble_auth_cmpl_t::addr_type (C++ member), 93
esp_ble_auth_cmpl_t::bd_addr (C++ member), 92
esp_ble_auth_cmpl_t::dev_type (C++ member), 93
esp_ble_auth_cmpl_t::fail_reason (C++ member), 92
esp_ble_auth_cmpl_t::key (C++ member), 92

esp_ble_auth_cmpl_t::key_present (C++ member), 92
 esp_ble_auth_cmpl_t::key_type (C++ member), 92
 esp_ble_auth_cmpl_t::success (C++ member), 92
 esp_ble_auth_req_t (C++ type), 94
 esp_ble_bond_dev_t (C++ class), 91
 esp_ble_bond_dev_t::bd_addr (C++ member), 92
 esp_ble_bond_dev_t::bond_key (C++ member), 92
 esp_ble_bond_key_info_t (C++ class), 91
 esp_ble_bond_key_info_t::key_mask (C++ member), 91
 esp_ble_bond_key_info_t::pcsrk_key (C++ member), 91
 esp_ble_bond_key_info_t::penc_key (C++ member), 91
 esp_ble_bond_key_info_t::pid_key (C++ member), 91
 esp_ble_clear_bond_device_list (C++ function), 81
 esp_ble_confirm_reply (C++ function), 81
ESP_BLE_CONN_PARAM_UNDEF (C macro), 72
 esp_ble_conn_update_params_t (C++ class), 89
 esp_ble_conn_update_params_t::bda (C++ member), 89
 esp_ble_conn_update_params_t::latency (C++ member), 89
 esp_ble_conn_update_params_t::max_int (C++ member), 89
 esp_ble_conn_update_params_t::min_int (C++ member), 89
 esp_ble_conn_update_params_t::timeout (C++ member), 89
ESP_BLE_CSR_KEY_MASK (C macro), 73
ESP_BLE_ENC_KEY_MASK (C macro), 73
ESP_BLE_EVT_CONN_ADV (C++ class), 98
ESP_BLE_EVT_CONN_DIR_ADV (C++ class), 98
ESP_BLE_EVT_DISC_ADV (C++ class), 98
ESP_BLE_EVT_NON_CONN_ADV (C++ class), 98
ESP_BLE_EVT_SCAN_RSP (C++ class), 99
 esp_ble_evt_type_t (C++ type), 98
 esp_ble_gap_cb_param_t (C++ type), 82
 esp_ble_gap_cb_param_t::adv_data_cmpl (C++ member), 82
 esp_ble_gap_cb_param_t::adv_data_raw_cmpl (C++ member), 83
 esp_ble_gap_cb_param_t::adv_start_cmpl (C++ member), 83
 esp_ble_gap_cb_param_t::adv_stop_cmpl (C++ member), 83
 esp_ble_gap_cb_param_t::ble_adv_data_cmpl_evt_param (C++ class), 83
 esp_ble_gap_cb_param_t::ble_adv_data_cmpl_evt_param::status (C++ member), 83
 esp_ble_gap_cb_param_t::ble_adv_data_raw_cmpl_evt_param (C++ class), 83
 esp_ble_gap_cb_param_t::ble_adv_data_raw_cmpl_evt_param::status (C++ member), 83
 esp_ble_gap_cb_param_t::ble_adv_start_cmpl_evt_param (C++ class), 84
 esp_ble_gap_cb_param_t::ble_adv_start_cmpl_evt_param::status (C++ member), 84
 esp_ble_gap_cb_param_t::ble_adv_stop_cmpl_evt_param (C++ class), 84
 esp_ble_gap_cb_param_t::ble_adv_stop_cmpl_evt_param::status (C++ member), 84
 esp_ble_gap_cb_param_t::ble_clear_bond_dev_cmpl_evt_param (C++ class), 84
 esp_ble_gap_cb_param_t::ble_clear_bond_dev_cmpl_evt_param::status (C++ member), 84
 esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param (C++ class), 84
 esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param::bond_dev (C++ member), 84
 esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param::dev_num (C++ member), 84
 esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param::status (C++ member), 84
 esp_ble_gap_cb_param_t::ble_local_privacy_cmpl_evt_param (C++ class), 84
 esp_ble_gap_cb_param_t::ble_local_privacy_cmpl_evt_param::status (C++ member), 84
 esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_evt_param (C++ class), 84
 esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_evt_param::params (C++ member), 85
 esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_evt_param::status (C++ member), 85
 esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param (C++ class), 85
 esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param::bd_addr (C++ member), 85
 esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param::status (C++ member), 85
 esp_ble_gap_cb_param_t::ble_scan_param_cmpl_evt_param (C++ class), 85
 esp_ble_gap_cb_param_t::ble_scan_param_cmpl_evt_param::status (C++ member), 85
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param (C++ class), 85
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::adv_data_len (C++ member), 86
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::bda (C++ member), 85
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::ble_addr_type (C++ member), 85
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::ble_adv (C++ member), 85
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::status (C++ member), 85
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::dev_type (C++ member), 85
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::flag (C++ member), 85
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::num_resps (C++ member), 85

esp_ble_gap_cb_param_t::ble_scan_result_evt_param::rssl esp_ble_gap_cb_param_t::scan_param_cmpl (C++ member), 82
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::scancpblegap_cb_param_t::scan_rsp_data_cmpl (C++ member), 82
 esp_ble_gap_cb_param_t::ble_scan_result_evt_param::searchespblegap_cb_param_t::scan_rsp_data_raw_cmpl (C++ member), 83
 esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::esp_ble_gap_cb_param_t::scan_rst (C++ member), 82
 esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::esp_ble_gap_cb_param_t::scan_start_cmpl (C++ member), 86
 esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::statusber, 83
 esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::esp_ble_gap_cb_param_t::scan_stop_cmpl (C++ member), 86
 esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::esp_ble_gap_cb_param_t::set_rand_addr_cmpl (C++ class), 86
 esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::statusber, 83
 esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param::esp_ble_gap_cb_param_t::update_conn_params (C++ member), 86
 esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_param::espblegap_disconnect (C++ function), 81
 esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_param::espblegap_register_callback (C++ function), 77
 esp_ble_gap_cb_param_t::ble_security (C++ member), 83
 esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param::espblegap_security_rsp (C++ function), 80
 esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param::espblegap_set_device_name (C++ function), 79
 esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param::espblegap_set_pkt_data_len (C++ function), 78
 esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param::espblegap_set_rand_addr (C++ function), 78
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espblegap_start_advertising (C++ function), 77
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espblegap_start_scanning (C++ function), 77
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espblegap_stop_advertising (C++ function), 78
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espblegap_stop_scanning (C++ function), 78
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espblegap_update_conn_params (C++ function), 78
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_app_register (C++ function), 121
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_app_unregister (C++ function), 122
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_cache_refresh (C++ function), 127
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_gattcb_param_t (C++ type), 128
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_cb_param_t::cfg_mtu (C++ member), 128
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_gattcb_param_t::close (C++ member), 128
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_gattcb_param_t::congest (C++ member), 128
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_gattcb_param_t::connect (C++ member), 129
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_gattcb_param_t::disconnect (C++ member), 129
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::timeð
 esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param::espbleGattc_cb_param_t::exec_cmpl (C++ member), 128
 esp_ble_gap_cb_param_t::clear_bond_dev_cmpl (C++ member), 83
 esp_ble_gap_cb_param_t::get_bond_dev_cmpl (C++ member), 83
 esp_ble_gap_cb_param_t::local_privacy_cmpl (C++ member), 83
 esp_ble_gap_cb_param_t::pkt_data_lenth_cmpl (C++ member), 83
 esp_ble_gap_cb_param_t::remove_bond_dev_cmpl (C++ member), 83
 esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param (C++ class), 129
 esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param::conn_id (C++ member), 129
 esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param::mtu (C++ member), 129
 esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param::status (C++ member), 129

esp_ble_gattc_cb_param_t::gattc_close_evt_param (C++ esp_ble_gattc_cb_param_t::gattc_get_descr_evt_param::conn_id
class), 129 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_close_evt_param::conn_id esp_ble_gattc_cb_param_t::gattc_get_descr_evt_param::descr_id
(C++ member), 129 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_close_evt_param::reason esp_ble_gattc_cb_param_t::gattc_get_descr_evt_param::srvc_id
(C++ member), 129 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_close_evt_param::remote esp_ble_gattc_cb_param_t::gattc_get_descr_evt_param::status
(C++ member), 129 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_close_evt_param::status esp_ble_gattc_cb_param_t::gattc_get_incl_srvc_evt_param
(C++ member), 129 (C++ class), 131
esp_ble_gattc_cb_param_t::gattc_congest_evt_param esp_ble_gattc_cb_param_t::gattc_get_incl_srvc_evt_param::conn_id
(C++ class), 129 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_congest_evt_param::congest esp_ble_gattc_cb_param_t::gattc_get_incl_srvc_evt_param::incl_srvc_id
(C++ member), 129 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_congest_evt_param::conn esp_ble_gattc_cb_param_t::gattc_get_incl_srvc_evt_param::srvc_id
(C++ member), 129 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_connect_evt_param esp_ble_gattc_cb_param_t::gattc_get_incl_srvc_evt_param::status
(C++ class), 130 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::conn esp_ble_gattc_cb_param_t::gattc_notify_evt_param
(C++ member), 130 (C++ class), 131
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::remote esp_ble_gattc_cb_param_t::gattc_notify_evt_param::char_id
(C++ member), 130 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_connect_evt_param::status esp_ble_gattc_cb_param_t::gattc_notify_evt_param::conn_id
(C++ member), 130 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param esp_ble_gattc_cb_param_t::gattc_notify_evt_param::descr_id
(C++ class), 130 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::conn esp_ble_gattc_cb_param_t::gattc_notify_evt_param::is_notify
(C++ member), 130 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::remote esp_ble_gattc_cb_param_t::gattc_notify_evt_param::remote_bda
(C++ member), 130 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param::status esp_ble_gattc_cb_param_t::gattc_notify_evt_param::srvc_id
(C++ member), 130 (C++ member), 131
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param esp_ble_gattc_cb_param_t::gattc_notify_evt_param::value
(C++ class), 130 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param::conn esp_ble_gattc_cb_param_t::gattc_notify_evt_param::value_len
(C++ member), 130 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param::status esp_ble_gattc_cb_param_t::gattc_open_evt_param (C++
class), 130
esp_ble_gattc_cb_param_t::gattc_get_char_evt_param esp_ble_gattc_cb_param_t::gattc_open_evt_param::conn_id
(C++ class), 130 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_get_char_evt_param::char esp_ble_gattc_cb_param_t::gattc_open_evt_param::mtu
(C++ member), 130 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_get_char_evt_param::char esp_ble_gattc_cb_param_t::gattc_open_evt_param::remote_bda
(C++ member), 131 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_get_char_evt_param::conn esp_ble_gattc_cb_param_t::gattc_open_evt_param::status
(C++ member), 130 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_get_char_evt_param::srvc esp_ble_gattc_cb_param_t::gattc_read_char_evt_param
(C++ member), 130 (C++ class), 132
esp_ble_gattc_cb_param_t::gattc_get_char_evt_param::status esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::char_id
(C++ member), 130 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_get_descr_evt_param esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::conn_id
(C++ class), 131 (C++ member), 132
esp_ble_gattc_cb_param_t::gattc_get_descr_evt_param::char esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::descr_id
(C++ member), 131 (C++ member), 132

esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::srvc_id
 (C++ member), 132
 esp_ble_gattc_cb_param_t::gattc_write_evt_param::descr_id
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::status
 (C++ member), 132
 esp_ble_gattc_cb_param_t::gattc_write_evt_param::status
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_read_char_evt_param::value_type
 (C++ member), 133
 esp_ble_gattc_cb_param_t::get_char (C++ member), 128
 esp_ble_gattc_cb_param_t::get_descr (C++ member),
 esp_ble_gattc_cb_param_t::get_incl_srvc (C++ member), 129
 esp_ble_gattc_cb_param_t::gattc_reg_evt_param (C++ class), 133
 esp_ble_gattc_cb_param_t::gattc_reg_evt_param::app_id
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_reg_evt_param::status
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param
 (C++ class), 133
 esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::char_id 128
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param::srvc_id 128
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_search_cmpl (C++ member), 128
 esp_ble_gattc_cb_param_t::gattc_search_cmpl (C++ member), 129
 esp_ble_gattc_cb_param_t::gattc_search_cmplEvtParam::stable
 esp_ble_gattc_cb_param_t::unreg_for_notify (C++ member), 129
 esp_ble_gattc_cb_param_t::gattc_search_cmplEvtParam::write (C++ member), 128
 esp_ble_gattc_close (C++ function), 122
 esp_ble_gattc_cb_param_t::gattc_search_cmplEvtParam::esp_ble_gattc_execute_write (C++ function), 127
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_search_cmplEvtParam::esp_ble_gattc_get_characteristic (C++ function), 123
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_search_cmplEvtParam::esp_ble_gattc_get_descriptor (C++ function), 123
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_search_resEvtParam (C++ class), 133
 esp_ble_gattc_cb_param_t::gattc_search_resEvtParam::esp_ble_gattc_open (C++ function), 122
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_search_resEvtParam::esp_ble_gattc_prepare_write (C++ function), 126
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_search_resEvtParam::esp_ble_gattc_prepare_write_char_descr (C++ function),
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_search_resEvtParam::esp_ble_gattc_read_char (C++ function), 124
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_srvc_chgEvtParam (C++ class), 133
 esp_ble_gattc_cb_param_t::gattc_srvc_chgEvtParam::register_callback (C++ function), 121
 (C++ member), 133
 esp_ble_gattc_cb_param_t::gattc_srvc_chgEvtParam::register_for_notify (C++ function), 127
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::remove (C++ function), 123
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::esp_ble_gattc_send_mtu_req (C++ function), 122
 (C++ class), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::esp_ble_gattc_unregister_for_notify (C++ function), 127
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::esp_ble_gattc_write_char (C++ function), 125
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::esp_ble_gattc_write_char_descr (C++ function), 125
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::esp_ble_gatts_add_char (C++ function), 109
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::esp_ble_gattsAddCharDescr (C++ function), 110
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::esp_ble_gattsAddIncludedService (C++ function), 109
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_unregForNotifyEvtParam::esp_ble_gattsAppRegister (C++ function), 108
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_writeEvtParam (C++ class), 134
 esp_ble_gattc_cb_param_t::gattc_writeEvtParam::add_attr_tab (C++ member),
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_writeEvtParam::char_id 113
 (C++ member), 134
 esp_ble_gattc_cb_param_t::gattc_writeEvtParam::esp_ble_gattsCbParamT::add_char (C++ member), 113
 esp_ble_gattc_cb_param_t::gattc_writeEvtParam::conn_id 113
 esp_ble_gattc_cb_param_t::gattc_writeEvtParam::esp_ble_gattsCbParamT::add_char_descr (C++ member), 113
 (C++ member), 134

esp_ble_gatts_cb_param_t::add_incl_srvc (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_cancel_open_evt_param::status (C++ member), 115
esp_ble_gatts_cb_param_t::cancel_open (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_close_evt_param (C++ class), 115
esp_ble_gatts_cb_param_t::close (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_close_evt_param::conn_id (C++ member), 115
esp_ble_gatts_cb_param_t::conf (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_close_evt_param::status (C++ member), 115
esp_ble_gatts_cb_param_t::congest (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_conf_evt_param (C++ class), 115
esp_ble_gatts_cb_param_t::connect (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_conf_evt_param::conn_id (C++ member), 115
esp_ble_gatts_cb_param_t::create (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_connect_evt_param (C++ class), 115
esp_ble_gatts_cb_param_t::del (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_connect_evt_param::is_connected (C++ member), 116
esp_ble_gatts_cb_param_t::disconnect (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_connect_evt_param::remote_bda (C++ member), 116
esp_ble_gatts_cb_param_t::exec_write (C++ member), 113	esp_ble_gatts_cb_param_t::gatts_create_evt_param (C++ class), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param (C++ class), 114	esp_ble_gatts_cb_param_t::gatts_create_evt_param::service_handle (C++ member), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_congest_evt_param::congested (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_create_evt_param::service_id (C++ member), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_congest_evt_param::conn_id (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_create_evt_param::status (C++ member), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_connect_evt_param (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_delete_evt_param (C++ class), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_connect_evt_param::conn_id (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_delete_evt_param::service_handle (C++ member), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_connect_evt_param::is_connected (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_delete_evt_param::service_id (C++ member), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_connect_evt_param::remote_bda (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param (C++ class), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_create_evt_param (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param::status (C++ member), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_create_evt_param::service_handle (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param (C++ class), 116
esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param:: esp_ble_gatts_cb_param_t::gatts_create_evt_param::service_id (C++ member), 114	esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::bda (C++ member), 117
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param (C++ class), 114	esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::status (C++ member), 117
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param::attr (C++ member), 114	
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param::char (C++ member), 114	
esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param::serv (C++ member), 114	
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param (C++ class), 114	
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param:: esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param::conn_id (C++ member), 114	
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param:: esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param::is_connected (C++ member), 115	
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param:: esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param::remote_bda (C++ member), 115	
esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param:: esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param (C++ member), 115	
esp_ble_gatts_cb_param_t::gatts_cancel_open_evt_param (C++ class), 115	

```

esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::conn_id_gatts_cb_param_t::gatts_start_evt_param::service_handle
    (C++ member), 117
    (C++ member), 118
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::exec_write_flag_cb_param_t::gatts_start_evt_param::status
    (C++ member), 117
    (C++ member), 118
esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param::trans_id_gatts_cb_param_t::gatts_stop_evt_param (C++ class),
    (C++ member), 117
    (C++ member), 118
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param (C++ class), 117 esp_ble_gatts_cb_param_t::gatts_stop_evt_param::service_handle
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param::conn_id esp_ble_gatts_cb_param_t::gatts_stop_evt_param::status
    (C++ member), 117
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_mtu_evt_param::mtu esp_ble_gatts_cb_param_t::gatts_write_evt_param (C++ class),
    (C++ member), 117
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_open_evt_param (C++ class), 117 esp_ble_gatts_cb_param_t::gatts_write_evt_param::bda
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_open_evt_param::status esp_ble_gatts_cb_param_t::gatts_write_evt_param::conn_id
    (C++ member), 117
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_read_evt_param (C++ class), 117 esp_ble_gatts_cb_param_t::gatts_write_evt_param::handle
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_read_evt_param::bda esp_ble_gatts_cb_param_t::gatts_write_evt_param::is_prep
    (C++ member), 117
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_read_evt_param::conn_id esp_ble_gatts_cb_param_t::gatts_write_evt_param::len
    (C++ member), 117
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_read_evt_param::handle esp_ble_gatts_cb_param_t::gatts_write_evt_param::need_rsp
    (C++ member), 117
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_read_evt_param::is_long esp_ble_gatts_cb_param_t::gatts_write_evt_param::offset
    (C++ member), 117
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_read_evt_param::need_rsp esp_ble_gatts_cb_param_t::gatts_write_evt_param::trans_id
    (C++ member), 118
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_read_evt_param::offset esp_ble_gatts_cb_param_t::gatts_write_evt_param::value
    (C++ member), 117
    (C++ member), 119
esp_ble_gatts_cb_param_t::gatts_read_evt_param::trans_id esp_ble_gatts_cb_param_t::mtu (C++ member), 113
    (C++ member), 117
    (C++ member), 113
esp_ble_gatts_cb_param_t::gatts_reg_evt_param (C++ class), 118 esp_ble_gatts_cb_param_t::open (C++ member), 113
    (C++ member), 113
esp_ble_gatts_cb_param_t::gatts_reg_evt_param::app_id esp_ble_gatts_cb_param_t::read (C++ member), 113
    (C++ member), 113
esp_ble_gatts_cb_param_t::gatts_reg_evt_param::status esp_ble_gatts_cb_param_t::reg (C++ member), 113
    (C++ member), 113
esp_ble_gatts_cb_param_t::gatts_rsp_evt_param (C++ class), 118 esp_ble_gatts_cb_param_t::rsp (C++ member), 113
    (C++ member), 113
esp_ble_gatts_cb_param_t::gatts_rsp_evt_param::handle esp_ble_gatts_cb_param_t::set_attr_val (C++ member),
    (C++ member), 118
    (C++ member), 114
esp_ble_gatts_cb_param_t::gatts_rsp_evt_param::status esp_ble_gatts_cb_param_t::start (C++ member), 113
    (C++ member), 118
    (C++ member), 113
esp_ble_gatts_cb_param_t::gatts_rsp_evt_param::status esp_ble_gatts_cb_param_t::stop (C++ member), 113
    (C++ member), 118
    (C++ member), 113
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param esp_ble_gatts_cb_param_t::write (C++ member), 113
    (C++ class), 118 esp_ble_gatts_close (C++ function), 112
    (C++ member), 118
    (C++ member), 112
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::attr_hdl_gatts_register_callback (C++ function), 108
    (C++ member), 118
    (C++ member), 111
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::attr_hdl_gatts_send_indicate (C++ function), 111
    (C++ member), 118
    (C++ member), 111
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::attr_hdl_gatts_send_response (C++ function), 111
    (C++ member), 118
    (C++ member), 111
esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param::attr_hdl_gatts_set_attr_value (C++ function), 111
    (C++ member), 118
    (C++ member), 110
esp_ble_gatts_cb_param_t::gatts_start_evt_param (C++ class), 118 esp_ble_gatts_start_service (C++ function), 110
    (C++ member), 118
    (C++ member), 110
esp_ble_gatts_cb_param_t::gatts_start_evt_param::service_handle esp_ble_gatts_stop_service (C++ function), 110
    (C++ member), 118
    (C++ member), 81
esp_ble_get_bond_device_list (C++ function), 81
ESP_BLE_ID_KEY_MASK (C macro), 73

```

esp_ble_io_cap_t (C++ type), 94
ESP_BLE_IS_VALID_PARAM (C macro), 72
esp_ble_key_mask_t (C++ type), 73
esp_ble_key_t (C++ class), 92
esp_ble_key_t::bd_addr (C++ member), 92
esp_ble_key_t::key_type (C++ member), 92
esp_ble_key_t::p_key_value (C++ member), 92
esp_ble_key_type_t (C++ type), 94
esp_ble_key_value_t (C++ type), 82
esp_ble_key_value_t::lcsrk_key (C++ member), 82
esp_ble_key_value_t::lenc_key (C++ member), 82
esp_ble_key_value_t::pcsrk_key (C++ member), 82
esp_ble_key_value_t::penc_key (C++ member), 82
esp_ble_key_value_t::pid_key (C++ member), 82
esp_ble_lcsrk_keys (C++ class), 90
esp_ble_lcsrk_keys::counter (C++ member), 91
esp_ble_lcsrk_keys::csr (C++ member), 91
esp_ble_lcsrk_keys::div (C++ member), 91
esp_ble_lcsrk_keys::sec_level (C++ member), 91
esp_ble_lenc_keys_t (C++ class), 90
esp_ble_lenc_keys_t::div (C++ member), 90
esp_ble_lenc_keys_t::key_size (C++ member), 90
esp_ble_lenc_keys_t::ltk (C++ member), 90
esp_ble_lenc_keys_t::sec_level (C++ member), 90
ESP_BLE_LINK_KEY_MASK (C macro), 73
esp_ble_local_id_keys_t (C++ class), 92
esp_ble_local_id_keys_t::dhk (C++ member), 92
esp_ble_local_id_keys_t::ir (C++ member), 92
esp_ble_local_id_keys_t::irk (C++ member), 92
esp_ble_passkey_reply (C++ function), 80
esp_ble_pcsrk_keys_t (C++ class), 90
esp_ble_pcsrk_keys_t::counter (C++ member), 90
esp_ble_pcsrk_keys_t::csr (C++ member), 90
esp_ble_pcsrk_keys_t::sec_level (C++ member), 90
esp_ble_penc_keys_t (C++ class), 89
esp_ble_penc_keys_t::ediv (C++ member), 89
esp_ble_penc_keys_t::key_size (C++ member), 90
esp_ble_penc_keys_t::ltk (C++ member), 89
esp_ble_penc_keys_t::rand (C++ member), 89
esp_ble_penc_keys_t::sec_level (C++ member), 89
esp_ble_pid_keys_t (C++ class), 90
esp_ble_pid_keys_t::addr_type (C++ member), 90
esp_ble_pid_keys_t::irk (C++ member), 90
esp_ble_pid_keys_t::static_addr (C++ member), 90
esp_ble_pkt_data_length_params_t (C++ class), 89
esp_ble_pkt_data_length_params_t::rx_len (C++ member), 89
esp_ble_pkt_data_length_params_t::tx_len (C++ member), 89
esp_ble_power_type_t (C++ type), 70
ESP_BLE_PWR_TYPE_ADV (C++ class), 71
ESP_BLE_PWR_TYPE_CONN_HDL0 (C++ class), 70
ESP_BLE_PWR_TYPE_CONN_HDL1 (C++ class), 70
ESP_BLE_PWR_TYPE_CONN_HDL2 (C++ class), 70
ESP_BLE_PWR_TYPE_CONN_HDL3 (C++ class), 70
ESP_BLE_PWR_TYPE_CONN_HDL4 (C++ class), 71
ESP_BLE_PWR_TYPE_CONN_HDL5 (C++ class), 71
ESP_BLE_PWR_TYPE_CONN_HDL6 (C++ class), 71
ESP_BLE_PWR_TYPE_CONN_HDL7 (C++ class), 71
ESP_BLE_PWR_TYPE_CONN_HDL8 (C++ class), 71
ESP_BLE_PWR_TYPE_CONN_HDL9 (C++ class), 71
ESP_BLE_PWR_TYPE_DEFAULT (C++ class), 71
ESP_BLE_PWR_TYPE_NUM (C++ class), 71
ESP_BLE_PWR_TYPE_SCAN (C++ class), 71
esp_ble_remove_bond_device (C++ function), 81
esp_ble_resolve_adv_data (C++ function), 79
esp_ble_scan_filter_t (C++ type), 97
esp_ble_scan_params_t (C++ class), 88
esp_ble_scan_params_t::own_addr_type (C++ member), 88
esp_ble_scan_params_t::scan_filter_policy (C++ member), 88
esp_ble_scan_params_t::scan_interval (C++ member), 88
esp_ble_scan_params_t::scan_type (C++ member), 88
esp_ble_scan_params_t::scan_window (C++ member), 89
ESP_BLE_SCAN_RSP_DATA_LEN_MAX (C macro), 94
esp_ble_scan_type_t (C++ type), 97
esp_ble_sec_act_t (C++ type), 97
ESP_BLE_SEC_ENCRYPT (C++ class), 97
ESP_BLE_SEC_ENCRYPT_MITM (C++ class), 97
ESP_BLE_SEC_ENCRYPT_NO_MITM (C++ class), 97
esp_ble_sec_key_notif_t (C++ class), 91
esp_ble_sec_key_notif_t::bd_addr (C++ member), 91
esp_ble_sec_key_notif_t::passkey (C++ member), 91
ESP_BLE_SEC_NONE (C++ class), 97
esp_ble_sec_req_t (C++ class), 91
esp_ble_sec_req_t::bd_addr (C++ member), 91
esp_ble_sec_t (C++ type), 82
esp_ble_sec_t::auth_cmpl (C++ member), 82
esp_ble_sec_t::ble_id_keys (C++ member), 82
esp_ble_sec_t::ble_key (C++ member), 82
esp_ble_sec_t::ble_req (C++ member), 82
esp_ble_sec_t::key_notif (C++ member), 82
esp_ble_set_encryption (C++ function), 80
ESP_BLE_SM_AUTHEN_REQ_MODE (C++ class), 97
ESP_BLE_SM_IOCAP_MODE (C++ class), 97
ESP_BLE_SM_MAX_KEY_SIZE (C++ class), 97
esp_ble_sm_param_t (C++ type), 97
ESP_BLE_SM_PASSKEY (C++ class), 97
ESP_BLE_SM_SET_INIT_KEY (C++ class), 97
ESP_BLE_SM_SET_RSP_KEY (C++ class), 97
esp_ble_tx_power_get (C++ function), 68
esp_ble_tx_power_set (C++ function), 68
esp_bluedroid_deinit (C++ function), 75
esp_bluedroid_disable (C++ function), 75
esp_bluedroid_enable (C++ function), 75

esp_bluedroid_get_status (C++ function), 75
 esp_bluedroid_init (C++ function), 75
 ESP_BLUEBOARD_STATUS_CHECK (C macro), 72
 ESP_BLUEBOARD_STATUS_ENABLED (C++ class), 75
 ESP_BLUEBOARD_STATUS_INITIALIZED (C++ class), 75
 esp_bluedroid_status_t (C++ type), 75
 ESP_BLUEBOARD_STATUS_UNINITIALIZED (C++ class), 75
 esp_bluifi_callbacks_t (C++ class), 144
 esp_bluifi_callbacks_t::checksum_func (C++ member), 144
 esp_bluifi_callbacks_t::decrypt_func (C++ member), 144
 esp_bluifi_callbacks_t::encrypt_func (C++ member), 144
 esp_bluifi_callbacks_t::event_cb (C++ member), 144
 esp_bluifi_callbacks_t::negotiate_data_handler (C++ member), 144
 esp_bluifi_cb_event_t (C++ type), 145
 esp_bluifi_cb_param_t (C++ type), 138
 esp_bluifi_cb_param_t::bluifi_connect_evt_param (C++ class), 139
 esp_bluifi_cb_param_t::bluifi_connect_evt_param::conn_id (C++ member), 139
 esp_bluifi_cb_param_t::bluifi_connect_evt_param::remote_bdap (C++ member), 139
 esp_bluifi_cb_param_t::bluifi_connect_evt_param::server_if (C++ member), 139
 esp_bluifi_cb_param_t::bluifi_deinit_finish_evt_param (C++ class), 139
 esp_bluifi_cb_param_t::bluifi_deinit_finish_evt_param::state (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_disconnect_evt_param (C++ class), 140
 esp_bluifi_cb_param_t::bluifi_disconnect_evt_param::remote_ip (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_init_finish_evt_param (C++ class), 140
 esp_bluifi_cb_param_t::bluifi_init_finish_evt_param::state (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_recv_ca_evt_param (C++ class), 140
 esp_bluifi_cb_param_t::bluifi_recv_ca_evt_param::cert (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_recv_ca_evt_param::cert_len (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_recv_client_cert_evt_param (C++ class), 140
 esp_bluifi_cb_param_t::bluifi_recv_client_cert_evt_param::cert (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_recv_client_cert_evt_param::cert_len (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_recv_client_pkey_evt_param (C++ class), 140
 esp_bluifi_cb_param_t::bluifi_recv_client_pkey_evt_param::pkey (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_recv_client_pkey_evt_param::pkey_len (C++ member), 140
 esp_bluifi_cb_param_t::bluifi_recv_server_cert_evt_param (C++ class), 141
 esp_bluifi_cb_param_t::bluifi_recv_server_cert_evt_param::cert (C++ member), 141
 esp_bluifi_cb_param_t::bluifi_recv_server_cert_evt_param::cert_len (C++ member), 141
 esp_bluifi_cb_param_t::bluifi_recv_server_pkey_evt_param (C++ class), 141
 esp_bluifi_cb_param_t::bluifi_recv_server_pkey_evt_param::pkey (C++ member), 141
 esp_bluifi_cb_param_t::bluifi_recv_server_pkey_evt_param::pkey_len (C++ member), 141
 esp_bluifi_cb_param_t::bluifi_recv_softap_auth_mode_evt_param (C++ class), 141
 esp_bluifi_cb_param_t::bluifi_recv_softap_auth_mode_evt_param::auth_mode (C++ member), 141
 esp_bluifi_cb_param_t::bluifi_recv_softap_channel_evt_param (C++ class), 141
 esp_bluifi_cb_param_t::bluifi_recv_softap_channel_evt_param::channel (C++ member), 141
 esp_bluifi_cb_param_t::bluifi_recv_softap_max_conn_num_evt_param (C++ class), 141
 esp_bluifi_cb_param_t::bluifi_recv_softap_max_conn_num_evt_param::max_conn_num (C++ member), 141
 esp_bluifi_cb_param_t::bluifi_recv_softap_passwd_evt_param (C++ class), 141
 esp_bluifi_cb_param_t::bluifi_recv_softap_passwd_evt_param::passwd (C++ member), 142
 esp_bluifi_cb_param_t::bluifi_recv_softap_passwd_evt_param::passwd_len (C++ member), 142
 esp_bluifi_cb_param_t::bluifi_recv_softap_ssid_evt_param (C++ class), 142
 esp_bluifi_cb_param_t::bluifi_recv_softap_ssid_evt_param::ssid (C++ member), 142
 esp_bluifi_cb_param_t::bluifi_recv_softap_ssid_evt_param::ssid_len (C++ member), 142
 esp_bluifi_cb_param_t::bluifi_recv_sta_bssid_evt_param (C++ class), 142
 esp_bluifi_cb_param_t::bluifi_recv_sta_bssid_evt_param::bssid (C++ member), 142
 esp_bluifi_cb_param_t::bluifi_recv_sta_passwd_evt_param (C++ class), 142
 esp_bluifi_cb_param_t::bluifi_recv_sta_passwd_evt_param::passwd (C++ member), 142
 esp_bluifi_cb_param_t::bluifi_recv_sta_passwd_evt_param::passwd_len (C++ member), 142
 esp_bluifi_cb_param_t::bluifi_recv_sta_ssid_evt_param (C++ class), 142
 esp_bluifi_cb_param_t::bluifi_recv_sta_ssid_evt_param::ssid (C++ member), 142

esp_bluifi_cb_param_t::blufi_recv_sta_ssid_evt_param::ssid [ESP_BLUFI_EVENT_RECV_CA_CERT](#) (C++ class),
 (C++ member), [142](#)
 esp_bluifi_cb_param_t::blufi_recv_username_evt_param [ESP_BLUFI_EVENT_RECV_CLIENT_CERT](#) (C++ class),
 (C++ class), [142](#)
 esp_bluifi_cb_param_t::blufi_recv_username_evt_param::name [ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY](#)
 (C++ class), [143](#)
 esp_bluifi_cb_param_t::blufi_recv_username_evt_param::name [ESP_BLUFI_EVENT_RECV_SERVER_CERT](#) (C++ class),
 (C++ member), [143](#)
 esp_bluifi_cb_param_t::blufi_set_wifi_mode_evt_param [ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY](#)
 (C++ class), [143](#)
 esp_bluifi_cb_param_t::blufi_set_wifi_mode_evt_param::op [ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE](#)
 (C++ class), [146](#)
 esp_bluifi_cb_param_t::ca (C++ member), [139](#)
 esp_bluifi_cb_param_t::client_cert (C++ member), [139](#)
 esp_bluifi_cb_param_t::client_pkey (C++ member), [139](#)
 esp_bluifi_cb_param_t::connect (C++ member), [138](#)
 esp_bluifi_cb_param_t::deinit_finish (C++ member), [138](#)
 esp_bluifi_cb_param_t::disconnect (C++ member), [138](#)
 esp_bluifi_cb_param_t::init_finish (C++ member), [138](#)
 esp_bluifi_cb_param_t::server_cert (C++ member), [139](#)
 esp_bluifi_cb_param_t::server_pkey (C++ member), [139](#)
 esp_bluifi_cb_param_t::softap_auth_mode (C++ member), [139](#)
 esp_bluifi_cb_param_t::softap_channel (C++ member), [139](#)
 esp_bluifi_cb_param_t::softap_max_conn_num (C++ member), [139](#)
 esp_bluifi_cb_param_t::softap_passwd (C++ member), [139](#)
 esp_bluifi_cb_param_t::softap_ssid (C++ member), [139](#)
 esp_bluifi_cb_param_t::sta_bssid (C++ member), [138](#)
 esp_bluifi_cb_param_t::sta_passwd (C++ member), [139](#)
 esp_bluifi_cb_param_t::sta_ssid (C++ member), [139](#)
 esp_bluifi_cb_param_t::username (C++ member), [139](#)
 esp_bluifi_cb_param_t::wifi_mode (C++ member), [138](#)
 esp_bluifi_checksum_func_t (C++ type), [145](#)
 esp_bluifi_close (C++ function), [138](#)
 esp_bluifi_decrypt_func_t (C++ type), [145](#)
[ESP_BLUFI_DEINIT_FAILED](#) (C++ class), [146](#)
[ESP_BLUFI_DEINIT_OK](#) (C++ class), [146](#)
 esp_bluifi_deinit_state_t (C++ type), [146](#)
 esp_bluifi_encrypt_func_t (C++ type), [145](#)
[ESP_BLUFI_EVENT_BLE_CONNECT](#) (C++ class), [145](#)
[ESP_BLUFI_EVENT_BLE_DISCONNECT](#) (C++ class), [145](#)
 esp_bluifi_event_cb_t (C++ type), [144](#)
[ESP_BLUFI_EVENT_DEAUTHENTICATE_STA](#) (C++ class), [145](#)
[ESP_BLUFI_EVENT_DEINIT_FINISH](#) (C++ class), [145](#)
[ESP_BLUFI_EVENT_GET_WIFI_STATUS](#) (C++ class), [145](#)
[ESP_BLUFI_EVENT_INIT_FINISH](#) (C++ class), [145](#)
[ESP_BLUFI_EVENT_RECV_CA_CERT](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_CLIENT_CERT](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_SERVER_CERT](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_SOFTAP_SSID](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_STA_BSSID](#) (C++ class), [145](#)
[ESP_BLUFI_EVENT_RECV_STA_PASSWD](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_STA_SSID](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_RECV_USERNAME](#) (C++ class), [146](#)
[ESP_BLUFI_EVENT_REQ_CONNECT_TO_AP](#) (C++ class), [145](#)
[ESP_BLUFI_EVENT_REQ_DISCONNECT_FROM_AP](#) (C++ class), [145](#)
[ESP_BLUFI_EVENT_SET_WIFI_OPMODE](#) (C++ class), [145](#)
 esp_bluifi_extra_info_t (C++ class), [143](#)
 esp_bluifi_extra_info_t::softap_authmode (C++ member), [143](#)
 esp_bluifi_extra_info_t::softap_authmode_set (C++ member), [143](#)
 esp_bluifi_extra_info_t::softap_channel (C++ member), [144](#)
 esp_bluifi_extra_info_t::softap_channel_set (C++ member), [144](#)
 esp_bluifi_extra_info_t::softap_max_conn_num (C++ member), [144](#)
 esp_bluifi_extra_info_t::softap_max_conn_num_set (C++ member), [144](#)
 esp_bluifi_extra_info_t::softap_passwd (C++ member), [143](#)
 esp_bluifi_extra_info_t::softap_passwd_len (C++ member), [143](#)
 esp_bluifi_extra_info_t::softap_ssid (C++ member), [143](#)

esp_bluifi_extra_info_t::softap_ssid_len (C++ member), 143
 esp_bluifi_extra_info_t::sta_bssid (C++ member), 143
 esp_bluifi_extra_info_t::sta_bssid_set (C++ member), 143
 esp_bluifi_extra_info_t::sta_passwd (C++ member), 143
 esp_bluifi_extra_info_t::sta_passwd_len (C++ member), 143
 esp_bluifi_extra_info_t::sta_ssid (C++ member), 143
 esp_bluifi_extra_info_t::sta_ssid_len (C++ member), 143
 esp_bluifi_get_version (C++ function), 138
 ESP_BLUIFI_INIT_FAILED (C++ class), 146
 ESP_BLUIFI_INIT_OK (C++ class), 146
 esp_bluifi_init_state_t (C++ type), 146
 esp_bluifi_negotiate_data_handler_t (C++ type), 144
 esp_bluifi_profile_deinit (C++ function), 137
 esp_bluifi_profile_init (C++ function), 137
 esp_bluifi_register_callbacks (C++ function), 137
 esp_bluifi_send_wifi_conn_report (C++ function), 137
 ESP_BLUIFI_STA_CONN_FAIL (C++ class), 146
 esp_bluifi_sta_conn_state_t (C++ type), 146
 ESP_BLUIFI_STA_CONN_SUCCESS (C++ class), 146
 esp_bt_controller_config_t (C++ class), 69
 esp_bt_controller_config_t::controller_task_prio (C++ member), 69
 esp_bt_controller_config_t::controller_task_stack_size (C++ member), 69
 esp_bt_controller_config_t::hci_uart_baudrate (C++ member), 69
 esp_bt_controller_config_t::hci_uart_no (C++ member), 69
 esp_bt_controller_deinit (C++ function), 68
 esp_bt_controller_disable (C++ function), 68
 esp_bt_controller_enable (C++ function), 68
 esp_bt_controller_get_status (C++ function), 68
 esp_bt_controller_init (C++ function), 68
 ESP_BT_CONTROLLER_STATUS_ENABLED (C++ class), 70
 ESP_BT_CONTROLLER_STATUS_IDLE (C++ class), 70
 ESP_BT_CONTROLLER_STATUS_INITED (C++ class), 70
 ESP_BT_CONTROLLER_STATUS_NUM (C++ class), 70
 esp_bt_controller_status_t (C++ type), 70
 esp_bt_dev_get_address (C++ function), 76
 esp_bt_dev_set_device_name (C++ function), 76
 esp_bt_dev_type_t (C++ type), 74
 ESP_BT_DEVICE_TYPE_BLE (C++ class), 74
 ESP_BT_DEVICE_TYPE_BREDR (C++ class), 74
 ESP_BT_DEVICE_TYPE_DUMO (C++ class), 74
 esp_bt_gap_set_scan_mode (C++ function), 147
 ESP_BT_MODE_BLE (C++ class), 70
 ESP_BT_MODE_BTDM (C++ class), 70
 ESP_BT_MODE_CLASSIC_BT (C++ class), 70
 ESP_BT_MODE_IDLE (C++ class), 70
 esp_bt_mode_t (C++ type), 70
 ESP_BT_OCTET16_LEN (C macro), 72
 esp_bt_octet16_t (C++ type), 73
 ESP_BT_OCTET8_LEN (C macro), 72
 esp_bt_octet8_t (C++ type), 73
 ESP_BT_SCAN_MODE_CONNECTABLE (C++ class), 147
 ESP_BT_SCAN_MODE_CONNECTABLE_DISCOVERABLE (C++ class), 147
 ESP_BT_SCAN_MODE_NONE (C++ class), 147
 esp_bt_scan_mode_t (C++ type), 147
 ESP_BT_STATUS_AUTH_FAILURE (C++ class), 73
 ESP_BT_STATUS_AUTH_REJECTED (C++ class), 74
 ESP_BT_STATUS_BUSY (C++ class), 73
 ESP_BT_STATUS_CONTROL_LE_DATA_LEN_UNSUPPORTED (C++ class), 74
 ESP_BT_STATUS_DONE (C++ class), 73
 ESP_BT_STATUS_ERR_ILLEGAL_PARAMETER_FMT (C++ class), 74
 ESP_BT_STATUS_FAIL (C++ class), 73
 ESP_BT_STATUS_INVALID_STATIC RAND ADDR (C++ class), 74
 ESP_BT_STATUS_NOMEM (C++ class), 73
 ESP_BT_STATUS_NOT_READY (C++ class), 73
 ESP_BT_STATUS_PARAM_OUT_OF_RANGE (C++ class), 74
 ESP_BT_STATUS_PARM_INVALID (C++ class), 73
 ESP_BT_STATUS_PEER_LE_DATA_LEN_UNSUPPORTED (C++ class), 74
 ESP_BT_STATUS_PENDING (C++ class), 74
 ESP_BT_STATUS_RMT_DEV_DOWN (C++ class), 74
 ESP_BT_STATUS_SUCCESS (C++ class), 73
 esp_bt_status_t (C++ type), 73
 ESP_BT_STATUS_TIMEOUT (C++ class), 74
 ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL (C++ class), 74
 ESP_BT_STATUS_UNHANDLED (C++ class), 73
 ESP_BT_STATUS_UNSUPPORTED (C++ class), 73
 esp_bt_uuid_t (C++ class), 72
 esp_bt_uuid_t::len (C++ member), 72
 esp_deep_sleep_start (C++ function), 405
 ESP_DEFAULT_GATT_IF (C macro), 72
 ESP_EARLY_LOGD (C macro), 409
 ESP_EARLY_LOGE (C macro), 409
 ESP_EARLY_LOGI (C macro), 409
 ESP_EARLY_LOGV (C macro), 409
 ESP_EARLY_LOGW (C macro), 409
 esp_efuse_mac_get_custom (C++ function), 412
 ESP_ERR_FLASH_BASE (C macro), 335
 ESP_ERR_FLASH_OP_FAIL (C macro), 335
 ESP_ERR_FLASH_OP_TIMEOUT (C macro), 335
 ESP_ERR_NVIS_BASE (C macro), 354
 ESP_ERR_NVIS_INVALID_HANDLE (C macro), 354

ESP_ERR_NVS_INVALID_LENGTH (C macro), 354
 ESP_ERR_NVS_INVALID_NAME (C macro), 354
 ESP_ERR_NVS_INVALID_STATE (C macro), 354
 ESP_ERR_NVS_KEY_TOO_LONG (C macro), 354
 ESP_ERR_NVS_NO_FREE_PAGES (C macro), 354
 ESP_ERR_NVS_NOT_ENOUGH_SPACE (C macro), 354
 ESP_ERR_NVS_NOT_FOUND (C macro), 354
 ESP_ERR_NVS_NOT_INITIALIZED (C macro), 354
 ESP_ERR_NVS_PAGE_FULL (C macro), 354
 ESP_ERR_NVS_PART_NOT_FOUND (C macro), 354
 ESP_ERR_NVS_READ_ONLY (C macro), 354
 ESP_ERR_NVS_REMOVE_FAILED (C macro), 354
 ESP_ERR_NVS_TYPE_MISMATCH (C macro), 354
 ESP_ERR_NVS_VALUE_TOO_LONG (C macro), 354
 ESP_ERR_OTA_BASE (C macro), 400
 ESP_ERR_OTA_PARTITION_CONFLICT (C macro), 400
 ESP_ERR_OTA_SELECT_INFO_INVALID (C macro), 400
 ESP_ERR_OTA_VALIDATE FAILED (C macro), 400
 ESP_ERR_ULP_BASE (C macro), 593
 ESP_ERR_ULP_BRANCH_OUT_OF_RANGE (C macro), 593
 ESP_ERR_ULP_DUPLICATE_LABEL (C macro), 593
 ESP_ERR_ULP_INVALID_LOAD_ADDR (C macro), 593
 ESP_ERR_ULP_SIZE_TOO_BIG (C macro), 593
 ESP_ERR_ULP_UNDEFINED_LABEL (C macro), 593
 ESP_ERR_WIFI_ARG (C macro), 63
 ESP_ERR_WIFI_CONN (C macro), 63
 ESP_ERR_WIFI_FAIL (C macro), 63
 ESP_ERR_WIFI_IF (C macro), 63
 ESP_ERR_WIFI_MAC (C macro), 64
 ESP_ERR_WIFI_MODE (C macro), 63
 ESP_ERR_WIFI_NO_MEM (C macro), 63
 ESP_ERR_WIFI_NOT_INIT (C macro), 63
 ESP_ERR_WIFI_NOT_STARTED (C macro), 63
 ESP_ERR_WIFI_NOT_STOPPED (C macro), 63
 ESP_ERR_WIFI_NOT_SUPPORT (C macro), 63
 ESP_ERR_WIFI_NVS (C macro), 63
 ESP_ERR_WIFI_OK (C macro), 63
 ESP_ERR_WIFI_PASSWORD (C macro), 64
 ESP_ERR_WIFI_SSID (C macro), 64
 ESP_ERR_WIFI_STATE (C macro), 63
 ESP_ERR_WIFI_TIMEOUT (C macro), 64
 ESP_ERR_WIFI_WAKE_FAIL (C macro), 64
 esp_esptouch_set_timeout (C++ function), 65
 esp_eth_disable (C++ function), 158
 esp_eth_enable (C++ function), 157
 esp_eth_free_rx_buf (C++ function), 159
 esp_eth_get_mac (C++ function), 158
 esp_eth_init (C++ function), 157
 esp_eth_init_internal (C++ function), 157
 esp_eth_smi_read (C++ function), 158
 esp_eth_smi_wait_set (C++ function), 158
 esp_eth_smi_wait_value (C++ function), 158
 esp_eth_smi_write (C++ function), 158
 esp_eth_tx (C++ function), 157
 ESP_EXT1_WAKEUP_ALL_LOW (C++ class), 403
 ESP_EXT1_WAKEUP_ANY_HIGH (C++ class), 403
 esp_flash_encrypt_check_and_update (C++ function), 342
 esp_flash_encrypt_region (C++ function), 342
 esp_flash_encryption_enabled (C++ function), 342
 ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT (C++ class), 94
 ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT (C++ class), 94
 ESP_GAP_BLE_ADV_START_COMPLETE_EVT (C++ class), 94
 ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT (C++ class), 95
 ESP_GAP_BLE_AUTH_CMPL_EVT (C++ class), 94
 esp_gap_ble_cb_event_t (C++ type), 94
 esp_gap_ble_cb_t (C++ type), 94
 ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT (C++ class), 95
 ESP_GAP_BLE_EVT_MAX (C++ class), 95
 ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT (C++ class), 95
 ESP_GAP_BLE_KEY_EVT (C++ class), 94
 ESP_GAP_BLE_LOCAL_ER_EVT (C++ class), 95
 ESP_GAP_BLE_LOCAL_IR_EVT (C++ class), 95
 ESP_GAP_BLE_NC_REQ_EVT (C++ class), 95
 ESP_GAP_BLE_OOB_REQ_EVT (C++ class), 95
 ESP_GAP_BLE_PASSKEY_NOTIF_EVT (C++ class), 95
 ESP_GAP_BLE_PASSKEY_REQ_EVT (C++ class), 95
 ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT (C++ class), 95
 ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT (C++ class), 94
 ESP_GAP_BLE_SCAN_RESULT_EVT (C++ class), 94
 ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT (C++ class), 94
 ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT (C++ class), 94
 ESP_GAP_BLE_SCAN_START_COMPLETE_EVT (C++ class), 94
 ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT (C++ class), 95
 ESP_GAP_BLE_SEC_REQ_EVT (C++ class), 94
 ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT (C++ class), 95
 ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT (C++ class), 95
 ESP_GAP_BLE_SET_STATIC RAND ADDR EVT

(C++ class), 95
ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT (C++ class), 95
ESP_GAP_SEARCH_DL_DISC_CMPL_EVT (C++ class), 98
ESP_GAP_SEARCH_DISC_BLE_RES_EVT (C++ class), 98
ESP_GAP_SEARCH_DISC_CMPL_EVT (C++ class), 98
ESP_GAP_SEARCH_DISC_RES_EVT (C++ class), 98
esp_gap_search_evt_t (C++ type), 98
ESP_GAP_SEARCH_INQ_CMPL_EVT (C++ class), 98
ESP_GAP_SEARCH_INQ_RES_EVT (C++ class), 98
ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT (C++ class), 98
ESP_GATT_ALREADY_OPEN (C++ class), 106
ESP_GATT_APP_RSP (C++ class), 106
ESP_GATT_ATTR_HANDLE_MAX (C macro), 104
ESP_GATT_AUTH_FAIL (C++ class), 106
ESP_GATT_AUTH_REQ_MITM (C++ class), 107
ESP_GATT_AUTH_REQ_NO_MITM (C++ class), 107
ESP_GATT_AUTH_REQ_NONE (C++ class), 107
ESP_GATT_AUTH_REQ_SIGNED_MITM (C++ class), 107
ESP_GATT_AUTH_REQ_SIGNED_NO_MITM (C++ class), 107
esp_gatt_auth_req_t (C++ type), 107
ESP_GATT_AUTO_RSP (C macro), 105
ESP_GATT_BODY_SENSOR_LOCATION (C macro), 104
ESP_GATT_BUSY (C++ class), 106
ESP_GATT_CANCEL (C++ class), 106
ESP_GATT_CCC_CFG_ERR (C++ class), 106
ESP_GATT_CHAR_PROP_BIT_AUTH (C macro), 105
ESP_GATT_CHAR_PROP_BIT_BROADCAST (C macro), 104
ESP_GATT_CHAR_PROP_BIT_EXT_PROP (C macro), 105
ESP_GATT_CHAR_PROP_BIT_INDICATE (C macro), 105
ESP_GATT_CHAR_PROP_BIT_NOTIFY (C macro), 104
ESP_GATT_CHAR_PROP_BIT_READ (C macro), 104
ESP_GATT_CHAR_PROP_BIT_WRITE (C macro), 104
ESP_GATT_CHAR_PROP_BIT_WRITE_NR (C macro), 104
esp_gatt_char_prop_t (C++ type), 105
ESP_GATT_CMD_STARTED (C++ class), 106
ESP_GATT_CONGESTED (C++ class), 106
ESP_GATT_CONN_CONN_CANCEL (C++ class), 107
ESP_GATT_CONN_FAIL_ESTABLISH (C++ class), 107
ESP_GATT_CONN_L2C_FAILURE (C++ class), 107
ESP_GATT_CONN_LMP_TIMEOUT (C++ class), 107
ESP_GATT_CONN_NONE (C++ class), 107
esp_gatt_conn_reason_t (C++ type), 106
ESP_GATT_CONN_TERMINATE_LOCAL_HOST (C++ class), 107
ESP_GATT_CONN_TERMINATE_PEER_USER (C++ class), 107
ESP_GATT_CONN_TIMEOUT (C++ class), 107
ESP_GATT_CONN_UNKNOWN (C++ class), 107
ESP_GATT_DB_FULL (C++ class), 106
ESP_GATT_DUP_REG (C++ class), 106
ESP_GATT_ENCRYPTED_MITM (C++ class), 106
ESP_GATT_ENCRYPTED_NO_MITM (C++ class), 106
ESP_GATT_ERR_UNLIKELY (C++ class), 106
ESP_GATT_ERROR (C++ class), 106
ESP_GATT_HEART_RATE_CNTL_POINT (C macro), 104
ESP_GATT_HEART_RATE_MEAS (C macro), 104
esp_gatt_id_t (C++ class), 99
esp_gatt_id_t::inst_id (C++ member), 99
esp_gatt_id_t::uuid (C++ member), 99
ESP_GATT_IF_NONE (C macro), 105
esp_gatt_if_t (C++ type), 105
ESP_GATT_ILLEGAL_HANDLE (C macro), 104
ESP_GATT_ILLEGAL_PARAMETER (C++ class), 106
ESP_GATT_ILLEGAL_UUID (C macro), 104
ESP_GATT_INSUF_AUTHENTICATION (C++ class), 105
ESP_GATT_INSUF_AUTHORIZATION (C++ class), 105
ESP_GATT_INSUF_ENCRYPTION (C++ class), 106
ESP_GATT_INSUF_KEY_SIZE (C++ class), 106
ESP_GATT_INSUF_RESOURCE (C++ class), 106
ESP_GATT_INTERNAL_ERROR (C++ class), 106
ESP_GATT_INVALID_ATTR_LEN (C++ class), 106
ESP_GATT_INVALID_CFG (C++ class), 106
ESP_GATT_INVALID_HANDLE (C++ class), 105
ESP_GATT_INVALID_OFFSET (C++ class), 105
ESP_GATT_INVALID_PDU (C++ class), 105
ESP_GATT_MAX_ATTR_LEN (C macro), 105
ESP_GATT_MORE (C++ class), 106
ESP_GATT_NO_RESOURCES (C++ class), 106
ESP_GATT_NOT_ENCRYPTED (C++ class), 106
ESP_GATT_NOT_FOUND (C++ class), 105
ESP_GATT_NOT_LONG (C++ class), 106
ESP_GATT_OK (C++ class), 105
ESP_GATT_OUT_OF_RANGE (C++ class), 106
ESP_GATT_PENDING (C++ class), 106
ESP_GATT_PERM_READ (C macro), 104
ESP_GATT_PERM_READ_ENC_MITM (C macro), 104
ESP_GATT_PERM_READ_ENCRYPTED (C macro), 104
esp_gatt_perm_t (C++ type), 105
ESP_GATT_PERM_WRITE (C macro), 104

ESP_GATT_PERM_WRITE_ENC_MITM (C macro), 104
 ESP_GATT_PERM_WRITE_ENCRYPTED (C macro), 104
 ESP_GATT_PERM_WRITE_SIGNED (C macro), 104
 ESP_GATT_PERM_WRITE_SIGNED_MITM (C macro), 104
 ESP_GATT_PRC_IN_PROGRESS (C++ class), 106
 ESP_GATT_PREP_WRITE_CANCEL (C macro), 119
 ESP_GATT_PREP_WRITE_CANCEL (C++ class), 105
 ESP_GATT_PREP_WRITE_EXEC (C macro), 119
 ESP_GATT_PREP_WRITE_EXEC (C++ class), 105
 esp_gatt_prep_write_type (C++ type), 105
 ESP_GATT_PREPARE_Q_FULL (C++ class), 105
 ESP_GATT_READ_NOT_PERMIT (C++ class), 105
 ESP_GATT_REQ_NOT_SUPPORTED (C++ class), 105
 ESP_GATT_RSP_BY_APP (C macro), 105
 esp_gatt_rsp_t (C++ type), 99
 esp_gatt_rsp_t::attr_value (C++ member), 99
 esp_gatt_rsp_t::handle (C++ member), 99
 ESP_GATT_SERVICE_STARTED (C++ class), 106
 esp_gatt_srvc_id_t (C++ class), 99
 esp_gatt_srvc_id_t::id (C++ member), 100
 esp_gatt_srvc_id_t::is_primary (C++ member), 100
 ESP_GATT_STACK_RSP (C++ class), 106
 esp_gatt_status_t (C++ type), 105
 ESP_GATT_UNKNOWN_ERROR (C++ class), 106
 ESP_GATT_UNSUPORT_GRP_TYPE (C++ class), 106
 ESP_GATT_UUID_ALERT_LEVEL (C macro), 103
 ESP_GATT_UUID_ALERT_NTF_SVC (C macro), 102
 ESP_GATT_UUID_ALERT_STATUS (C macro), 103
 ESP_GATT_UUID_BATTERY_LEVEL (C macro), 104
 ESP_GATT_UUID_BATTERY_SERVICE_SVC (C macro), 102
 ESP_GATT_UUID_BLOOD_PRESSURE_SVC (C macro), 102
 ESP_GATT_UUID_CHAR_AGG_FORMAT (C macro), 102
 ESP_GATT_UUID_CHAR_CLIENT_CONFIG (C macro), 102
 ESP_GATT_UUID_CHAR_DECLARE (C macro), 102
 ESP_GATT_UUID_CHAR_DESCRIPTION (C macro), 102
 ESP_GATT_UUID_CHAR_EXT_PROP (C macro), 102
 ESP_GATT_UUID_CHAR_PRESENT_FORMAT (C macro), 102
 ESP_GATT_UUID_CHAR_SRVR_CONFIG (C macro), 102
 ESP_GATT_UUID_CHAR_VALID_RANGE (C macro), 103
 ESP_GATT_UUID_CSC_FEATURE (C macro), 104
 ESP_GATT_UUID_CSC_MEASUREMENT (C macro), 104
 ESP_GATT_UUID_CURRENT_TIME (C macro), 103
 ESP_GATT_UUID_CURRENT_TIME_SVC (C macro), 102
 ESP_GATT_UUID_CYCLING_POWER_SVC (C macro), 102
 ESP_GATT_UUID_CYCLING_SPEED_CADENCE_SVC (C macro), 102
 ESP_GATT_UUID_DEVICE_INFO_SVC (C macro), 102
 ESP_GATT_UUID_EXT_RPT_REF_DESCR (C macro), 103
 ESP_GATT_UUID_FW_VERSION_STR (C macro), 103
 ESP_GATT_UUID_GAP_CENTRAL_ADDR_RESOL (C macro), 103
 ESP_GATT_UUID_GAP_DEVICE_NAME (C macro), 103
 ESP_GATT_UUID_GAP_ICON (C macro), 103
 ESP_GATT_UUID_GAP_PREF_CONN_PARAM (C macro), 103
 ESP_GATT_UUID_GATT_SRV_CHGD (C macro), 103
 ESP_GATT_UUID_GLUCOSE_SVC (C macro), 102
 ESP_GATT_UUID_GM_CONTEXT (C macro), 103
 ESP_GATT_UUID_GM_CONTROL_POINT (C macro), 103
 ESP_GATT_UUID_GM_FEATURE (C macro), 103
 ESP_GATT_UUID_GM_MEASUREMENT (C macro), 103
 ESP_GATT_UUID_HEALTH_THERMOM_SVC (C macro), 102
 ESP_GATT_UUID_HEART_RATE_SVC (C macro), 102
 ESP_GATT_UUID_HID_BT_KB_INPUT (C macro), 104
 ESP_GATT_UUID_HID_BT_KB_OUTPUT (C macro), 104
 ESP_GATT_UUID_HID_BT_MOUSE_INPUT (C macro), 104
 ESP_GATT_UUID_HID_CONTROL_POINT (C macro), 103
 ESP_GATT_UUID_HID_INFORMATION (C macro), 103
 ESP_GATT_UUID_HID_PROTO_MODE (C macro), 103
 ESP_GATT_UUID_HID_REPORT (C macro), 103
 ESP_GATT_UUID_HID_REPORT_MAP (C macro), 103
 ESP_GATT_UUID_HID_SVC (C macro), 102
 ESP_GATT_UUID_HW_VERSION_STR (C macro), 103
 ESP_GATT_UUID_IEEE_DATA (C macro), 103
 ESP_GATT_UUID_IMMEDIATE_ALERT_SVC (C macro), 102
 ESP_GATT_UUID_INCLUDE_SERVICE (C macro),

102
ESP_GATT_UUID_LINK_LOSS_SVC (C macro), 102
ESP_GATT_UUID_LOCAL_TIME_INFO (C macro), 103
ESP_GATT_UUID_LOCATION_AND_NAVIGATION_SVC (C macro), 102
ESP_GATT_UUID_MANU_NAME (C macro), 103
ESP_GATT_UUID_MODEL_NUMBER_STR (C macro), 103
ESP_GATT_UUID_NEXT_DST_CHANGE_SVC (C macro), 102
ESP_GATT_UUID_NW_STATUS (C macro), 103
ESP_GATT_UUID_NW_TRIGGER (C macro), 103
ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC (C macro), 102
ESP_GATT_UUID_PNP_ID (C macro), 103
ESP_GATT_UUID_PRI_SERVICE (C macro), 102
ESP_GATT_UUID_REF_TIME_INFO (C macro), 103
ESP_GATT_UUID_REF_TIME_UPDATE_SVC (C macro), 102
ESP_GATT_UUID_RINGER_CP (C macro), 103
ESP_GATT_UUID_RINGER_SETTING (C macro), 103
ESP_GATT_UUID_RPT_REF_DESCR (C macro), 103
ESP_GATT_UUID_RSC_FEATURE (C macro), 104
ESP_GATT_UUID_RSC_MEASUREMENT (C macro), 104
ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC (C macro), 102
ESP_GATT_UUID_SC_CONTROL_POINT (C macro), 104
ESP_GATT_UUID_SCAN_INT_WINDOW (C macro), 104
ESP_GATT_UUID_SCAN_PARAMETERS_SVC (C macro), 102
ESP_GATT_UUID_SCAN_REFRESH (C macro), 104
ESP_GATT_UUID_SEC_SERVICE (C macro), 102
ESP_GATT_UUID_SENSOR_LOCATION (C macro), 104
ESP_GATT_UUID_SERIAL_NUMBER_STR (C macro), 103
ESP_GATT_UUID_SW_VERSION_STR (C macro), 103
ESP_GATT_UUID_SYSTEM_ID (C macro), 103
ESP_GATT_UUID_TX_POWER_LEVEL (C macro), 103
ESP_GATT_UUID_TX_POWER_SVC (C macro), 102
ESP_GATT_UUID_USER_DATA_SVC (C macro), 102
ESP_GATT_UUID_WEIGHT_SCALE_SVC (C macro), 102
esp_gatt_value_t (C++ class), 101
esp_gatt_value_t::auth_req (C++ member), 101
esp_gatt_value_t::handle (C++ member), 101
esp_gatt_value_t::len (C++ member), 101
esp_gatt_value_t::offset (C++ member), 101
esp_gatt_value_t::value (C++ member), 101
ESP_GATT_WRITE_NOT_PERMIT (C++ class), 105
ESP_GATT_WRITE_TYPE_NO_RSP (C++ class), 107
ESP_GATT_WRITE_TYPE_RSP (C++ class), 107
esp_gatt_write_type_t (C++ type), 107
ESP_GATT_WRONG_STATE (C++ class), 106
ESP_GATTC_ACL_EVT (C++ class), 135
ESP_GATTC_ADV_DATA_EVT (C++ class), 135
ESP_GATTC_ADV_VSC_EVT (C++ class), 136
ESP_GATTC_BTH_SCAN_CFG_EVT (C++ class), 136
ESP_GATTC_BTH_SCAN_DIS_EVT (C++ class), 136
ESP_GATTC_BTH_SCAN_ENB_EVT (C++ class), 136
ESP_GATTC_BTH_SCAN_PARAM_EVT (C++ class), 136
ESP_GATTC_BTH_SCAN_RD_EVT (C++ class), 136
ESP_GATTC_BTH_SCAN_THR_EVT (C++ class), 136
ESP_GATTC_CANCEL_OPEN_EVT (C++ class), 135
esp_gattc_cb_event_t (C++ type), 135
esp_gattc_cb_t (C++ type), 134
ESP_GATTC_CFG_MTU_EVT (C++ class), 135
ESP_GATTC_CLOSE_EVT (C++ class), 135
ESP_GATTC_CONGEST_EVT (C++ class), 136
ESP_GATTC_CONNECT_EVT (C++ class), 136
ESP_GATTC_DISCONNECT_EVT (C++ class), 136
ESP_GATTC_ENC_CMPL_CB_EVT (C++ class), 135
ESP_GATTC_EXEC_EVT (C++ class), 135
ESP_GATTC_GET_CHAR_EVT (C++ class), 136
ESP_GATTC_GET_DESCR_EVT (C++ class), 136
ESP_GATTC_GET_INCL_SRVC_EVT (C++ class), 136
ESP_GATTC_MULT_ADV_DATA_EVT (C++ class), 136
ESP_GATTC_MULT_ADV_DIS_EVT (C++ class), 136
ESP_GATTC_MULT_ADV_ENB_EVT (C++ class), 135
ESP_GATTC_MULT_ADV_UPD_EVT (C++ class), 136
ESP_GATTC_NOTIFY_EVT (C++ class), 135
ESP_GATTC_OPEN_EVT (C++ class), 135
ESP_GATTC_PREP_WRITE_EVT (C++ class), 135
ESP_GATTC_READ_CHAR_EVT (C++ class), 135
ESP_GATTC_READ_DESCR_EVT (C++ class), 135
ESP_GATTC_REG_EVT (C++ class), 135
ESP_GATTC_REG_FOR_NOTIFY_EVT (C++ class), 136
ESP_GATTC_SCAN_FLT_CFG_EVT (C++ class), 136
ESP_GATTC_SCAN_FLT_PARAM_EVT (C++ class), 136
ESP_GATTC_SCAN_FLT_STATUS_EVT (C++ class), 136
ESP_GATTC_SEARCH_CMPL_EVT (C++ class), 135
ESP_GATTC_SEARCH_RES_EVT (C++ class), 135
ESP_GATTC_SRVC_CHG_EVT (C++ class), 135
ESP_GATTC_UNREG_EVT (C++ class), 135

ESP_GATT_C_UNREG_FOR_NOTIFY_EVT class), 136	(C++)	ESP_INTR_FLAG_LEVEL1 (C macro), 393
ESP_GATT_WRITE_CHAR_EVT (C++ class), 135		ESP_INTR_FLAG_LEVEL2 (C macro), 393
ESP_GATT_WRITE_DESCR_EVT (C++ class), 135		ESP_INTR_FLAG_LEVEL3 (C macro), 393
ESP_GATTS_ADD_CHAR_DESCR_EVT (C++ class), 120		ESP_INTR_FLAG_LEVEL4 (C macro), 393
ESP_GATTS_ADD_CHAR_EVT (C++ class), 120		ESP_INTR_FLAG_LEVEL5 (C macro), 393
ESP_GATTS_ADD_INCL_SRVC_EVT (C++ class), 120		ESP_INTR_FLAG_LEVEL6 (C macro), 393
esp_gatts_attr_db_t (C++ class), 100		ESP_INTR_FLAG_LEVELMASK (C macro), 394
esp_gatts_attr_db_t::att_desc (C++ member), 100		ESP_INTR_FLAG_LOWMED (C macro), 394
esp_gatts_attr_db_t::attr_control (C++ member), 100		ESP_INTR_FLAG_NMI (C macro), 394
ESP_GATTS_CANCEL_OPEN_EVT (C++ class), 120		ESP_INTR_FLAG_SHARED (C macro), 394
esp_gatts_cb_event_t (C++ type), 120		esp_intr_free (C++ function), 392
esp_gatts_cb_t (C++ type), 119		esp_intr_get_cpu (C++ function), 392
ESP_GATTS_CLOSE_EVT (C++ class), 121		esp_intr_get_intno (C++ function), 392
ESP_GATTS_CONF_EVT (C++ class), 120		esp_intr_mark_shared (C++ function), 390
ESP_GATTS_CONGEST_EVT (C++ class), 121		esp_intr_noniram_disable (C++ function), 393
ESP_GATTS_CONNECT_EVT (C++ class), 120		esp_intr_noniram_enable (C++ function), 393
ESP_GATTS_CREAT_ATTR_TAB_EVT (C++ class), 121		esp_intr_reserve (C++ function), 391
ESP_GATTS_CREATE_EVT (C++ class), 120		ESP_IO_CAP_IN (C macro), 93
ESP_GATTS_DELETE_EVT (C++ class), 120		ESP_IO_CAP_IO (C macro), 93
ESP_GATTS_DISCONNECT_EVT (C++ class), 120		ESP_IO_CAP_KBDISP (C macro), 93
ESP_GATTS_EXEC_WRITE_EVT (C++ class), 120		ESP_IO_CAP_NONE (C macro), 93
esp_gatts_incl128_svc_desc_t (C++ class), 101		ESP_IO_CAP_OUT (C macro), 93
esp_gatts_incl128_svc_desc_t::end_hdl (C++ member), 101		ESP_LE_AUTH_BOND (C macro), 93
esp_gatts_incl128_svc_desc_t::start_hdl (C++ member), 101		ESP_LE_AUTH_NO_BOND (C macro), 93
esp_gatts_incl_svc_desc_t (C++ class), 101		ESP_LE_AUTH_REQ_MITM (C macro), 93
esp_gatts_incl_svc_desc_t::end_hdl (C++ member), 101		ESP_LE_AUTH_REQ_SC_BOND (C macro), 93
esp_gatts_incl_svc_desc_t::start_hdl (C++ member), 101		ESP_LE_AUTH_REQ_SC_MITM (C macro), 93
esp_gatts_incl_svc_desc_t::uuid (C++ member), 101		ESP_LE_AUTH_REQ_SC_MITM_BOND (C macro), 93
ESP_GATTS_LISTEN_EVT (C++ class), 121		ESP_LE_AUTH_REQ_SC_ONLY (C macro), 93
ESP_GATTS_MTU_EVT (C++ class), 120		ESP_LE_KEY_LCSRK (C macro), 93
ESP_GATTS_OPEN_EVT (C++ class), 120		ESP_LE_KEY_LENC (C macro), 93
ESP_GATTS_READ_EVT (C++ class), 120		ESP_LE_KEY_LID (C macro), 93
ESP_GATTS_REG_EVT (C++ class), 120		ESP_LE_KEY_LLK (C macro), 93
ESP_GATTS_RESPONSE_EVT (C++ class), 121		ESP_LE_KEY_NONE (C macro), 93
ESP_GATTS_SET_ATTR_VAL_EVT (C++ class), 121		ESP_LE_KEY_PCSRK (C macro), 93
ESP_GATTS_START_EVT (C++ class), 120		ESP_LE_KEY_PENC (C macro), 93
ESP_GATTS_STOP_EVT (C++ class), 120		ESP_LE_KEY_PID (C macro), 93
ESP_GATTS_UNREG_EVT (C++ class), 120		ESP_LE_KEY_PLK (C macro), 93
ESP_GATTS_WRITE_EVT (C++ class), 120		esp_light_sleep_start (C++ function), 405
esp_int_wdt_init (C++ function), 396		ESP_LINE_ENDINGS_CR (C++ class), 361
esp_intr_alloc (C++ function), 391		ESP_LINE_ENDINGS_CRLF (C++ class), 361
esp_intr_alloc_intrstatus (C++ function), 391		ESP_LINE_ENDINGS_LF (C++ class), 361
esp_intr_disable (C++ function), 393		esp_line_endings_t (C++ type), 361
esp_intr_enable (C++ function), 393		esp_link_key (C++ type), 73
ESP_INTR_FLAG_EDGE (C macro), 394		esp_log_buffer_char (C++ function), 409
ESP_INTR_FLAG_HIGH (C macro), 394		esp_log_buffer_hex (C++ function), 408
ESP_INTR_FLAG_INTRDISABLED (C macro), 394		ESP_LOG_DEBUG (C++ class), 410
ESP_INTR_FLAG_IRAM (C macro), 394		esp_log_early_timestamp (C++ function), 408
		ESP_LOG_ERROR (C++ class), 410
		ESP_LOG_INFO (C++ class), 410
		esp_log_level_set (C++ function), 408
		esp_log_level_t (C++ type), 410
		ESP_LOG_NONE (C++ class), 410

esp_log_set_vprintf (C++ function), 408	ESP_PARTITION_SUBTYPE_APP_OTA_6 (C++ class), 340
esp_log_timestamp (C++ function), 408	ESP_PARTITION_SUBTYPE_APP_OTA_7 (C++ class), 340
ESP_LOG_VERBOSE (C++ class), 410	ESP_PARTITION_SUBTYPE_APP_OTA_8 (C++ class), 340
ESP_LOG_WARN (C++ class), 410	ESP_PARTITION_SUBTYPE_APP_OTA_9 (C++ class), 340
esp_log_write (C++ function), 408	ESP_PARTITION_SUBTYPE_APP_OTA_MAX (C++ class), 341
ESP_LOGD (C macro), 410	ESP_PARTITION_SUBTYPE_APP_OTA_MIN (C++ class), 340
ESP_LOGE (C macro), 410	ESP_PARTITION_SUBTYPE_APP_TEST (C++ class), 341
ESP_LOGI (C macro), 410	ESP_PARTITION_SUBTYPE_DATA_COREDUMP (C++ class), 341
ESP_LOGV (C macro), 410	ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD (C++ class), 341
ESP_LOGW (C macro), 410	ESP_PARTITION_SUBTYPE_DATA_FAT (C++ class), 341
esp_ota_begin (C++ function), 397	ESP_PARTITION_SUBTYPE_DATA_NVS (C++ class), 341
esp_ota_end (C++ function), 398	ESP_PARTITION_SUBTYPE_DATA_OTA (C++ class), 341
esp_ota_get_boot_partition (C++ function), 399	ESP_PARTITION_SUBTYPE_DATA_PHY (C++ class), 341
esp_ota_get_next_update_partition (C++ function), 399	ESP_PARTITION_SUBTYPE_DATA_SPIFFS (C++ class), 341
esp_ota_get_running_partition (C++ function), 399	ESP_PARTITION_SUBTYPE_OTA (C macro), 339
esp_ota_handle_t (C++ type), 400	esp_partition_subtype_t (C++ type), 340
esp_ota_set_boot_partition (C++ function), 398	esp_partition_t (C++ class), 339
esp_ota_write (C++ function), 397	esp_partition_t::address (C++ member), 339
esp_partition_erase_range (C++ function), 338	esp_partition_t::encrypted (C++ member), 339
esp_partition_find (C++ function), 336	esp_partition_t::label (C++ member), 339
esp_partition_find_first (C++ function), 336	esp_partition_t::size (C++ member), 339
esp_partition_get (C++ function), 336	esp_partition_t::subtype (C++ member), 339
esp_partition_iterator_release (C++ function), 337	esp_partition_t::type (C++ member), 339
esp_partition_iterator_t (C++ type), 340	ESP_PARTITION_TYPE_APP (C++ class), 340
esp_partition_mmap (C++ function), 338	ESP_PARTITION_TYPE_DATA (C++ class), 340
esp_partition_next (C++ function), 337	esp_partition_type_t (C++ type), 340
esp_partition_read (C++ function), 337	esp_partition_verify (C++ function), 337
ESP_PARTITION_SUBTYPE_ANY (C++ class), 341	esp_partition_write (C++ function), 338
ESP_PARTITION_SUBTYPE_APP_FACTORY (C++ class), 340	ESP_PD_DOMAIN_MAX (C++ class), 404
ESP_PARTITION_SUBTYPE_APP_OTA_0 (C++ class), 340	ESP_PD_DOMAIN_RTC_FAST_MEM (C++ class), 404
ESP_PARTITION_SUBTYPE_APP_OTA_1 (C++ class), 340	ESP_PD_DOMAIN_RTC_PERIPH (C++ class), 404
ESP_PARTITION_SUBTYPE_APP_OTA_10 (C++ class), 341	ESP_PD_DOMAIN_RTC_SLOW_MEM (C++ class), 404
ESP_PARTITION_SUBTYPE_APP_OTA_11 (C++ class), 341	ESP_PD_OPTION_AUTO (C++ class), 405
ESP_PARTITION_SUBTYPE_APP_OTA_12 (C++ class), 341	ESP_PD_OPTION_OFF (C++ class), 405
ESP_PARTITION_SUBTYPE_APP_OTA_13 (C++ class), 341	ESP_PD_OPTION_ON (C++ class), 405
ESP_PARTITION_SUBTYPE_APP_OTA_14 (C++ class), 341	esp_power_level_t (C++ type), 71
ESP_PARTITION_SUBTYPE_APP_OTA_15 (C++ class), 341	ESP_PWR_LVL_N11 (C++ class), 71
ESP_PARTITION_SUBTYPE_APP_OTA_2 (C++ class), 340	ESP_PWR_LVL_N14 (C++ class), 71
ESP_PARTITION_SUBTYPE_APP_OTA_3 (C++ class), 340	
ESP_PARTITION_SUBTYPE_APP_OTA_4 (C++ class), 340	
ESP_PARTITION_SUBTYPE_APP_OTA_5 (C++ class), 340	

ESP_PWR_LVL_N2 (C++ class), 71
 ESP_PWR_LVL_N5 (C++ class), 71
 ESP_PWR_LVL_N8 (C++ class), 71
 ESP_PWR_LVL_P1 (C++ class), 71
 ESP_PWR_LVL_P4 (C++ class), 71
 ESP_PWR_LVL_P7 (C++ class), 71
 esp_sleep_enable_ext0_wakeup (C++ function), 402
 esp_sleep_enable_ext1_wakeup (C++ function), 403
 esp_sleep_enable_timer_wakeup (C++ function), 401
 esp_sleep_enable_touchpad_wakeup (C++ function), 401
 esp_sleep_enable_ulp_wakeup (C++ function), 404
 esp_sleep_ext1_wakeup_mode_t (C++ type), 403
 esp_sleep_get_ext1_wakeup_status (C++ function), 406
 esp_sleep_get_touchpad_wakeup_status (C++ function), 406
 esp_sleep_get_wakeup_cause (C++ function), 405
 esp_sleep_pd_config (C++ function), 404
 esp_sleep_pd_domain_t (C++ type), 404
 esp_sleep_pd_option_t (C++ type), 404
 esp_sleep_wakeup_cause_t (C++ type), 405
 ESP_SLEEP_WAKEUP_EXT0 (C++ class), 405
 ESP_SLEEP_WAKEUP_EXT1 (C++ class), 406
 ESP_SLEEP_WAKEUP_TIMER (C++ class), 406
 ESP_SLEEP_WAKEUP_TOUCHPAD (C++ class), 406
 ESP_SLEEP_WAKEUP_ULP (C++ class), 406
 ESP_SLEEP_WAKEUP_UNDEFINED (C++ class), 405
 esp_smartconfig_fast_mode (C++ function), 66
 esp_smartconfig_get_version (C++ function), 65
 esp_smartconfig_set_type (C++ function), 66
 esp_smartconfig_start (C++ function), 65
 esp_smartconfig_stop (C++ function), 65
 esp_spiffs_format (C++ function), 372
 esp_spiffs_info (C++ function), 372
 esp_spiffs_mounted (C++ function), 371
 esp_task_wdt_delete (C++ function), 396
 esp_task_wdt_feed (C++ function), 396
 esp_task_wdt_init (C++ function), 396
 ESP_UUID_LEN_128 (C macro), 72
 ESP_UUID_LEN_16 (C macro), 72
 ESP_UUID_LEN_32 (C macro), 72
 esp_vendor_ie_cb_t (C++ type), 64
 esp_vfs_close (C++ function), 359
 esp_vfs_dev_uart_register (C++ function), 360
 esp_vfs_dev_uart_set_rx_line_endings (C++ function), 360
 esp_vfs_dev_uart_set_tx_line_endings (C++ function), 360
 esp_vfs_dev_uart_use_driver (C++ function), 361
 esp_vfs_dev_uart_use_nonblocking (C++ function), 361
 esp_vfs_fat_mount_config_t (C++ class), 364, 367
 esp_vfs_fat_mount_config_t::format_if_mount_failed (C++ member), 364, 367
 esp_vfs_fat_mount_config_t::max_files (C++ member), 364, 367
 esp_vfs_fat_register (C++ function), 362
 esp_vfs_fat_sdmmc_mount (C++ function), 363
 esp_vfs_fat_sdmmc_unmount (C++ function), 364
 esp_vfs_fat_spiflash_mount (C++ function), 366
 esp_vfs_fat_spiflash_unmount (C++ function), 367
 esp_vfs_fat_unregister_path (C++ function), 362
 ESP_VFS_FLAG_CONTEXT_PTR (C macro), 360
 ESP_VFS_FLAG_DEFAULT (C macro), 360
 esp_vfs_fstat (C++ function), 359
 esp_vfs_link (C++ function), 359
 esp_vfs_lseek (C++ function), 359
 esp_vfs_open (C++ function), 359
 ESP_VFS_PATH_MAX (C macro), 360
 esp_vfs_read (C++ function), 359
 esp_vfs_register (C++ function), 359
 esp_vfs_rename (C++ function), 359
 esp_vfs_spiffs_conf_t (C++ class), 372
 esp_vfs_spiffs_conf_t::base_path (C++ member), 372
 esp_vfs_spiffs_conf_t::format_if_mount_failed (C++ member), 372
 esp_vfs_spiffs_conf_t::max_files (C++ member), 372
 esp_vfs_spiffs_conf_t::partition_label (C++ member), 372
 esp_vfs_spiffs_register (C++ function), 371
 esp_vfs_spiffs_unregister (C++ function), 371
 esp_vfs_stat (C++ function), 359
 esp_vfs_t (C++ class), 359
 esp_vfs_t::fd_offset (C++ member), 360
 esp_vfs_t::flags (C++ member), 360
 esp_vfs_unlink (C++ function), 359
 esp_vfs_unregister (C++ function), 359
 esp_vfs_write (C++ function), 359
 esp_vhci_host_callback (C++ class), 69
 esp_vhci_host_callback::notify_host_recv (C++ member), 69
 esp_vhci_host_callback::notify_host_send_available (C++ member), 69
 esp_vhci_host_callback_t (C++ type), 70
 esp_vhci_host_check_send_available (C++ function), 69
 esp_vhci_host_register_callback (C++ function), 69
 esp_vhci_host_send_packet (C++ function), 69
 esp_wifi_ap_get_sta_list (C++ function), 59
 esp_wifi_clear_fast_connect (C++ function), 52
 esp_wifi_connect (C++ function), 51
 esp_wifi_deauth_sta (C++ function), 52
 esp_wifi_deinit (C++ function), 50
 esp_wifi_disconnect (C++ function), 51
 esp_wifi_get_auto_connect (C++ function), 60
 esp_wifi_get_bandwidth (C++ function), 55
 esp_wifi_get_channel (C++ function), 56
 esp_wifi_get_config (C++ function), 59
 esp_wifi_get_country (C++ function), 56
 esp_wifi_get_mac (C++ function), 57
 esp_wifi_get_max_tx_power (C++ function), 61

esp_wifi_get_mode (C++ function), 50
 esp_wifi_get_promiscuous (C++ function), 58
 esp_wifi_get_promiscuous_filter (C++ function), 58
 esp_wifi_get_protocol (C++ function), 54
 esp_wifi_get_ps (C++ function), 54
 esp_wifi_init (C++ function), 49
 esp_wifi_restore (C++ function), 51
 esp_wifi_scan_get_ap_num (C++ function), 53
 esp_wifi_scan_get_ap_records (C++ function), 53
 esp_wifi_scan_start (C++ function), 52
 esp_wifi_scan_stop (C++ function), 53
 esp_wifi_set_auto_connect (C++ function), 60
 esp_wifi_set_bandwidth (C++ function), 55
 esp_wifi_set_channel (C++ function), 55
 esp_wifi_set_config (C++ function), 58
 esp_wifi_set_country (C++ function), 56
 esp_wifi_set_mac (C++ function), 56
 esp_wifi_set_max_tx_power (C++ function), 61
 esp_wifi_set_mode (C++ function), 50
 esp_wifi_set_promiscuous (C++ function), 57
 esp_wifi_set_promiscuous_filter (C++ function), 58
 esp_wifi_set_promiscuous_rx_cb (C++ function), 57
 esp_wifi_set_protocol (C++ function), 54
 esp_wifi_set_ps (C++ function), 54
 esp_wifi_set_storage (C++ function), 59
 esp_wifi_set_vendor_ie (C++ function), 60
 esp_wifi_set_vendor_ie_cb (C++ function), 61
 esp_wifi_sta_get_ap_info (C++ function), 53
 esp_wifi_start (C++ function), 50
 esp_wifi_stop (C++ function), 51
 eth_config_t (C++ class), 159
 eth_config_t::flow_ctrl_enable (C++ member), 159
 eth_config_t::gpio_config (C++ member), 159
 eth_config_t::mac_mode (C++ member), 159
 eth_config_t::phy_addr (C++ member), 159
 eth_config_t::phy_check_init (C++ member), 159
 eth_config_t::phy_check_link (C++ member), 159
 eth_config_t::phy_get_duplex_mode (C++ member), 159
 eth_config_t::phy_get_partner_pause_enable (C++ member), 159
 eth_config_t::phy_get_speed_mode (C++ member), 159
 eth_config_t::phy_init (C++ member), 159
 eth_config_t::phy_power_enable (C++ member), 159
 eth_config_t::tcpip_input (C++ member), 159
 eth_duplex_mode_t (C++ type), 160
 eth_gpio_config_func (C++ type), 160
 ETH_MODE_FULLDUPLEX (C++ class), 160
 ETH_MODE_HALFDUPLEX (C++ class), 160
 ETH_MODE_MII (C++ class), 160
 ETH_MODE_RMII (C++ class), 160
 eth_mode_t (C++ type), 160
 eth_phy_base_t (C++ type), 160
 eth_phy_check_init_func (C++ type), 160
 eth_phy_check_link_func (C++ type), 160

eth_phy_func (C++ type), 160
 eth_phy_get_duplex_mode_func (C++ type), 160
 eth_phy_get_partner_pause_enable_func (C++ type), 160
 eth_phy_get_speed_mode_func (C++ type), 160
 eth_phy_power_enable_func (C++ type), 160
 ETH_SPEED_MODE_100M (C++ class), 160
 ETH_SPEED_MODE_10M (C++ class), 160
 eth_speed_mode_t (C++ type), 160
 eth_tcpip_input_func (C++ type), 160
 ETS_INTERNAL_INTR_SOURCE_OFF (C macro), 394
 ETS_INTERNAL_PROFILING_INTR_SOURCE (C macro), 394
 ETS_INTERNAL_SW0_INTR_SOURCE (C macro), 394
 ETS_INTERNAL_SW1_INTR_SOURCE (C macro), 394
 ETS_INTERNAL_TIMER0_INTR_SOURCE (C macro), 394
 ETS_INTERNAL_TIMER1_INTR_SOURCE (C macro), 394
 ETS_INTERNAL_TIMER2_INTR_SOURCE (C macro), 394

F

ff_diskio_impl_t (C++ class), 364
 ff_diskio_impl_t::init (C++ member), 365
 ff_diskio_impl_t::ioctl (C++ member), 365
 ff_diskio_impl_t::read (C++ member), 365
 ff_diskio_impl_t::status (C++ member), 365
 ff_diskio_impl_t::write (C++ member), 365
 ff_diskio_register (C++ function), 364
 ff_diskio_register_sdmmc (C++ function), 365

G

gpio_config (C++ function), 174
 gpio_config_t (C++ class), 180
 gpio_config_t::intr_type (C++ member), 180
 gpio_config_t::mode (C++ member), 180
 gpio_config_t::pin_bit_mask (C++ member), 180
 gpio_config_t::pull_down_en (C++ member), 180
 gpio_config_t::pull_up_en (C++ member), 180
 GPIO_DRIVE_CAP_0 (C++ class), 182
 GPIO_DRIVE_CAP_1 (C++ class), 182
 GPIO_DRIVE_CAP_2 (C++ class), 182
 GPIO_DRIVE_CAP_3 (C++ class), 182
 GPIO_DRIVE_CAP_DEFAULT (C++ class), 182
 GPIO_DRIVE_CAP_MAX (C++ class), 182
 gpio_drive_cap_t (C++ type), 182
 GPIO_FLOATING (C++ class), 182
 gpio_get_drive_capability (C++ function), 179
 gpio_get_level (C++ function), 175
 gpio_install_isr_service (C++ function), 178
 gpio_int_type_t (C++ type), 181

GPIO_INTR_ANYEDGE (C++ class), 181
GPIO_INTR_DISABLE (C++ class), 181
gpio_intr_disable (C++ function), 175
gpio_intr_enable (C++ function), 175
GPIO_INTR_HIGH_LEVEL (C++ class), 181
GPIO_INTR_LOW_LEVEL (C++ class), 181
GPIO_INTR_MAX (C++ class), 181
GPIO_INTR_NEGEDGE (C++ class), 181
GPIO_INTR_POSEDGE (C++ class), 181
GPIO_IS_VALID_GPIO (C macro), 180
GPIO_IS_VALID_OUTPUT_GPIO (C macro), 180
gpio_isr_handle_t (C++ type), 180
gpio_isr_handler_add (C++ function), 178
gpio_isr_handler_remove (C++ function), 179
gpio_isr_register (C++ function), 177
gpio_isr_t (C++ type), 180
GPIO_MODE_INPUT (C++ class), 181
GPIO_MODE_INPUT_OUTPUT (C++ class), 181
GPIO_MODE_INPUT_OUTPUT_OD (C++ class), 181
GPIO_MODE_OUTPUT (C++ class), 181
GPIO_MODE_OUTPUT_OD (C++ class), 181
gpio_mode_t (C++ type), 181
GPIO_NUM_0 (C++ class), 180
GPIO_NUM_1 (C++ class), 180
GPIO_NUM_2 (C++ class), 181
gpio_num_t (C++ type), 180
gpio_pull_mode_t (C++ type), 182
gpio_pulldown_dis (C++ function), 178
GPIO_PULLDOWN_DISABLE (C++ class), 181
gpio_pulldown_en (C++ function), 178
GPIO_PULLDOWN_ENABLE (C++ class), 182
GPIO_PULLDOWN_ONLY (C++ class), 182
gpio_pulldown_t (C++ type), 181
gpio_pullup_dis (C++ function), 177
GPIO_PULLUP_DISABLE (C++ class), 181
gpio_pullup_en (C++ function), 177
GPIO_PULLUP_ENABLE (C++ class), 181
GPIO_PULLUP_ONLY (C++ class), 182
GPIO_PULLUP_PULLDOWN (C++ class), 182
gpio_pullup_t (C++ type), 181
GPIO_SEL_0 (C macro), 180
GPIO_SEL_1 (C macro), 180
GPIO_SEL_2 (C macro), 180
gpio_set_direction (C++ function), 176
gpio_set_drive_capability (C++ function), 179
gpio_set_intr_type (C++ function), 175
gpio_set_level (C++ function), 175
gpio_set_pull_mode (C++ function), 176
gpio_uninstall_isr_service (C++ function), 178
gpio_wakeup_disable (C++ function), 177
gpio_wakeup_enable (C++ function), 176

H

hall_sensor_read (C++ function), 165

heap_caps_add_region (C++ function), 377
heap_caps_add_region_with_caps (C++ function), 377
heap_caps_check_integrity (C++ function), 375
heap_caps_enable_nonos_stack_heaps (C++ function), 377
heap_caps_free (C++ function), 374
heap_caps_get_free_size (C++ function), 374
heap_caps_get_info (C++ function), 375
heap_caps_get_largest_free_block (C++ function), 375
heap_caps_get_minimum_free_size (C++ function), 375
heap_caps_init (C++ function), 377
heap_caps_malloc (C++ function), 374
heap_caps_print_heap_info (C++ function), 375
heap_caps_realloc (C++ function), 374
HEAP_TRACE_ALL (C++ class), 388
heap_trace_dump (C++ function), 387
heap_trace_get (C++ function), 387
heap_trace_get_count (C++ function), 387
heap_trace_init_standalone (C++ function), 386
HEAP_TRACE_LEAKS (C++ class), 388
heap_trace_mode_t (C++ type), 388
heap_trace_record_t (C++ class), 388
heap_trace_record_t::address (C++ member), 388
heap_trace_record_t::alloced_by (C++ member), 388
heap_trace_record_t::ccount (C++ member), 388
heap_trace_record_t::freed_by (C++ member), 388
heap_trace_record_t::size (C++ member), 388
heap_trace_resume (C++ function), 387
heap_trace_start (C++ function), 386
heap_trace_stop (C++ function), 387
HSPI_HOST (C++ class), 272

|

I2C_ADDR_BIT_10 (C++ class), 197
I2C_ADDR_BIT_7 (C++ class), 197
I2C_ADDR_BIT_MAX (C++ class), 197
i2c_addr_mode_t (C++ type), 197
I2C_APB_CLK_FREQ (C macro), 196
I2C_CMD_END (C++ class), 197
i2c_cmd_handle_t (C++ type), 196
i2c_cmd_link_create (C++ function), 190
i2c_cmd_link_delete (C++ function), 190
I2C_CMD_READ (C++ class), 197
I2C_CMD_RESTART (C++ class), 197
I2C_CMD_STOP (C++ class), 197
I2C_CMD_WRITE (C++ class), 197
i2c_config_t (C++ class), 195
i2c_config_t::addr_10bit_en (C++ member), 196
i2c_config_t::clk_speed (C++ member), 196
i2c_config_t::mode (C++ member), 196
i2c_config_t::scl_io_num (C++ member), 196
i2c_config_t::scl_pullup_en (C++ member), 196
i2c_config_t::sda_io_num (C++ member), 196
i2c_config_t::sda_pullup_en (C++ member), 196

i2c_config_t::slave_addr (C++ member), 196
 I2C_DATA_MODE_LSB_FIRST (C++ class), 197
 I2C_DATA_MODE_MAX (C++ class), 197
 I2C_DATA_MODE_MSB_FIRST (C++ class), 197
 i2c_driver_delete (C++ function), 188
 i2c_driver_install (C++ function), 187
 I2C_FIFO_LEN (C macro), 196
 i2c_get_data_mode (C++ function), 195
 i2c_get_data_timing (C++ function), 195
 i2c_get_period (C++ function), 193
 i2c_get_start_timing (C++ function), 194
 i2c_get_stop_timing (C++ function), 194
 i2c_isr_free (C++ function), 189
 i2c_isr_register (C++ function), 189
 i2c_master_cmd_begin (C++ function), 192
 I2C_MASTER_READ (C++ class), 196
 i2c_master_read (C++ function), 191
 i2c_master_read_byte (C++ function), 191
 i2c_master_start (C++ function), 190
 i2c_master_stop (C++ function), 191
 I2C_MASTER_WRITE (C++ class), 196
 i2c_master_write (C++ function), 190
 i2c_master_write_byte (C++ function), 190
 I2C_MODE_MASTER (C++ class), 196
 I2C_MODE_MAX (C++ class), 196
 I2C_MODE_SLAVE (C++ class), 196
 i2c_mode_t (C++ type), 196
 I2C_NUM_0 (C++ class), 197
 I2C_NUM_1 (C++ class), 197
 I2C_NUM_MAX (C++ class), 197
 i2c_opmode_t (C++ type), 197
 i2c_param_config (C++ function), 188
 i2c_port_t (C++ type), 197
 i2c_reset_rx_fifo (C++ function), 188
 i2c_reset_tx_fifo (C++ function), 188
 i2c_rw_t (C++ type), 196
 i2c_set_data_mode (C++ function), 195
 i2c_set_data_timing (C++ function), 194
 i2c_set_period (C++ function), 193
 i2c_set_pin (C++ function), 189
 i2c_set_start_timing (C++ function), 193
 i2c_set_stop_timing (C++ function), 194
 i2c_slave_read_buffer (C++ function), 192
 i2c_slave_write_buffer (C++ function), 192
 i2c_trans_mode_t (C++ type), 197
 I2S_BITS_PER_SAMPLE_16BIT (C++ class), 204
 I2S_BITS_PER_SAMPLE_24BIT (C++ class), 204
 I2S_BITS_PER_SAMPLE_32BIT (C++ class), 204
 I2S_BITS_PER_SAMPLE_8BIT (C++ class), 204
 i2s_bits_per_sample_t (C++ type), 204
 I2S_CHANNEL_FMT_ALL_LEFT (C++ class), 205
 I2S_CHANNEL_FMT_ALL_RIGHT (C++ class), 205
 I2S_CHANNEL_FMT_ONLY_LEFT (C++ class), 205
 I2S_CHANNEL_FMT_ONLY_RIGHT (C++ class), 205
 I2S_CHANNEL_FMT_RIGHT_LEFT (C++ class), 205
 i2s_channel_fmt_t (C++ type), 205
 I2S_CHANNEL_MONO (C++ class), 205
 I2S_CHANNEL_STEREO (C++ class), 205
 i2s_channel_t (C++ type), 204
 I2S_COMM_FORMAT_I2S (C++ class), 205
 I2S_COMM_FORMAT_I2S_LSB (C++ class), 205
 I2S_COMM_FORMAT_I2S_MSB (C++ class), 205
 I2S_COMM_FORMAT_PCM (C++ class), 205
 I2S_COMM_FORMAT_PCM_LONG (C++ class), 205
 I2S_COMM_FORMAT_PCM_SHORT (C++ class), 205
 i2s_comm_format_t (C++ type), 205
 i2s_config_t (C++ class), 203
 i2s_config_t::bits_per_sample (C++ member), 203
 i2s_config_t::channel_format (C++ member), 203
 i2s_config_t::communication_format (C++ member), 203
 i2s_config_t::dma_buf_count (C++ member), 203
 i2s_config_t::dma_buf_len (C++ member), 203
 i2s_config_t::intr_alloc_flags (C++ member), 203
 i2s_config_t::mode (C++ member), 203
 i2s_config_t::sample_rate (C++ member), 203
 I2S_DAC_CHANNEL_BOTH_EN (C++ class), 207
 I2S_DAC_CHANNEL_DISABLE (C++ class), 206
 I2S_DAC_CHANNEL_LEFT_EN (C++ class), 206
 I2S_DAC_CHANNEL_MAX (C++ class), 207
 I2S_DAC_CHANNEL_RIGHT_EN (C++ class), 206
 i2s_dac_mode_t (C++ type), 206
 i2s_driver_install (C++ function), 200
 i2s_driver_uninstall (C++ function), 200
 I2S_EVENT_DMA_ERROR (C++ class), 206
 I2S_EVENT_MAX (C++ class), 206
 I2S_EVENT_RX_DONE (C++ class), 206
 i2s_event_t (C++ class), 203
 i2s_event_t::size (C++ member), 204
 i2s_event_t::type (C++ member), 204
 I2S_EVENT_TX_DONE (C++ class), 206
 i2s_event_type_t (C++ type), 206
 i2s_isr_handle_t (C++ type), 204
 I2S_MODE_DAC_BUILT_IN (C++ class), 206
 I2S_MODE_MASTER (C++ class), 206
 I2S_MODE_PDM (C++ class), 206
 I2S_MODE_RX (C++ class), 206
 I2S_MODE_SLAVE (C++ class), 206
 i2s_mode_t (C++ type), 206
 I2S_MODE_TX (C++ class), 206
 I2S_NUM_0 (C++ class), 206
 I2S_NUM_1 (C++ class), 206
 I2S_NUM_MAX (C++ class), 206
 i2s_pin_config_t (C++ class), 204
 i2s_pin_config_t::bck_io_num (C++ member), 204
 i2s_pin_config_t::data_in_num (C++ member), 204
 i2s_pin_config_t::data_out_num (C++ member), 204
 i2s_pin_config_t::ws_io_num (C++ member), 204
 I2S_PIN_NO_CHANGE (C macro), 204

i2s_pop_sample (C++ function), 201
 i2s_port_t (C++ type), 205
 i2s_push_sample (C++ function), 201
 i2s_read_bytes (C++ function), 201
 i2s_set_clk (C++ function), 203
 i2s_set_dac_mode (C++ function), 199
 i2s_set_pin (C++ function), 199
 i2s_set_sample_rates (C++ function), 201
 i2s_start (C++ function), 202
 i2s_stop (C++ function), 202
 i2s_write_bytes (C++ function), 200
 i2s_zero_dma_buffer (C++ function), 202
 I_ADDI (C macro), 595
 I_ADDR (C macro), 595
 I_ANDI (C macro), 595
 I_ANDR (C macro), 595
 I_BGE (C macro), 594
 I_BL (C macro), 594
 I_BXFI (C macro), 594
 I_BXFR (C macro), 594
 I_BXI (C macro), 594
 I_BXR (C macro), 594
 I_BXZI (C macro), 594
 I_BXZR (C macro), 594
 I_DELAY (C macro), 593
 I_END (C macro), 593
 I_HALT (C macro), 593
 I_LD (C macro), 594
 I_LSHI (C macro), 595
 I_LSHR (C macro), 595
 I_MOVI (C macro), 595
 I_MOVR (C macro), 595
 I_ORI (C macro), 595
 I_ORR (C macro), 595
 I_RD_REG (C macro), 594
 I_RSHI (C macro), 595
 I_RSHR (C macro), 595
 I_ST (C macro), 594
 I_SUBI (C macro), 595
 I_SUBR (C macro), 595
 I_WR_REG (C macro), 594
 intr_handle_data_t (C++ type), 394
 intr_handle_t (C++ type), 394
 intr_handler_t (C++ type), 394

L

LEDC_APB_CLK (C++ class), 214
 LEDC_APB_CLK_HZ (C macro), 213
 ledc_bind_channel_timer (C++ function), 211
 LEDC_CHANNEL_0 (C++ class), 214
 LEDC_CHANNEL_1 (C++ class), 214
 LEDC_CHANNEL_2 (C++ class), 214
 LEDC_CHANNEL_3 (C++ class), 215
 LEDC_CHANNEL_4 (C++ class), 215
 LEDC_CHANNEL_5 (C++ class), 215
 LEDC_CHANNEL_6 (C++ class), 215
 LEDC_CHANNEL_7 (C++ class), 215
 ledc_channel_config (C++ function), 207
 ledc_channel_config_t (C++ class), 213
 ledc_channel_config_t::channel (C++ member), 213
 ledc_channel_config_t::duty (C++ member), 213
 ledc_channel_config_t::gpio_num (C++ member), 213
 ledc_channel_config_t::intr_type (C++ member), 213
 ledc_channel_config_t::speed_mode (C++ member), 213
 ledc_channel_config_t::timer_sel (C++ member), 213
 LEDC_CHANNEL_MAX (C++ class), 215
 ledc_channel_t (C++ type), 214
 ledc_clk_src_t (C++ type), 214
 LEDC_DUTY_DIR_DECREASE (C++ class), 214
 LEDC_DUTY_DIR_INCREASE (C++ class), 214
 ledc_duty_direction_t (C++ type), 214
 ledc_fade_func_install (C++ function), 212
 ledc_fade_func_uninstall (C++ function), 212
 LEDC_FADE_MAX (C++ class), 215
 ledc_fade_mode_t (C++ type), 215
 LEDC_FADE_NO_WAIT (C++ class), 215
 ledc_fade_start (C++ function), 212
 LEDC_FADE_WAIT_DONE (C++ class), 215
 ledc_get_duty (C++ function), 209
 ledc_get_freq (C++ function), 208
 LEDC_HIGH_SPEED_MODE (C++ class), 214
 LEDC_INTR_DISABLE (C++ class), 214
 LEDC_INTR_FADE_END (C++ class), 214
 ledc_intr_type_t (C++ type), 214
 ledc_isr_handle_t (C++ type), 213
 ledc_isr_register (C++ function), 210
 LEDC_LOW_SPEED_MODE (C++ class), 214
 ledc_mode_t (C++ type), 214
 LEDC_REF_CLK_HZ (C macro), 213
 LEDC_REF_TICK (C++ class), 214
 ledc_set_duty (C++ function), 209
 ledc_set_fade (C++ function), 209
 ledc_set_fade_with_step (C++ function), 211
 ledc_set_fade_with_time (C++ function), 212
 ledc_set_freq (C++ function), 208
 LEDC_SPEED_MODE_MAX (C++ class), 214
 ledc_stop (C++ function), 208
 LEDC_TIMER_0 (C++ class), 214
 LEDC_TIMER_1 (C++ class), 214
 LEDC_TIMER_10_BIT (C++ class), 215
 LEDC_TIMER_11_BIT (C++ class), 215
 LEDC_TIMER_12_BIT (C++ class), 215
 LEDC_TIMER_13_BIT (C++ class), 215
 LEDC_TIMER_14_BIT (C++ class), 215
 LEDC_TIMER_15_BIT (C++ class), 215
 LEDC_TIMER_2 (C++ class), 214
 LEDC_TIMER_3 (C++ class), 214
 ledc_timer_bit_t (C++ type), 215

ledc_timer_config (C++ function), 207
 ledc_timer_config_t (C++ class), 213
 ledc_timer_config_t::bit_num (C++ member), 213
 ledc_timer_config_t::freq_hz (C++ member), 213
 ledc_timer_config_t::speed_mode (C++ member), 213
 ledc_timer_config_t::timer_num (C++ member), 213
 ledc_timer_pause (C++ function), 210
 ledc_timer_resume (C++ function), 211
 ledc_timer_rst (C++ function), 210
 ledc_timer_set (C++ function), 210
 ledc_timer_t (C++ type), 214
 ledc_update_duty (C++ function), 208
 LOG_COLOR_D (C macro), 409
 LOG_COLOR_E (C macro), 409
 LOG_COLOR_I (C macro), 409
 LOG_COLOR_V (C macro), 409
 LOG_COLOR_W (C macro), 409
 LOG_FORMAT (C macro), 409
 LOG_LOCAL_LEVEL (C macro), 409
 LOG_RESET_COLOR (C macro), 409

M

M_BGE (C macro), 595
 M_BL (C macro), 595
 M_BX (C macro), 596
 M_BXF (C macro), 596
 M_BXZ (C macro), 596
 M_LABEL (C macro), 595
 MALLOC_CAP_32BIT (C macro), 376
 MALLOC_CAP_8BIT (C macro), 376
 MALLOC_CAP_DMA (C macro), 376
 MALLOC_CAP_EXEC (C macro), 376
 MALLOC_CAP_INVALID (C macro), 376
 MALLOC_CAP_PID2 (C macro), 376
 MALLOC_CAP_PID3 (C macro), 376
 MALLOC_CAP_PID4 (C macro), 376
 MALLOC_CAP_PID5 (C macro), 376
 MALLOC_CAP_PID6 (C macro), 376
 MALLOC_CAP_PID7 (C macro), 376
 MALLOC_CAP_SPISRAM (C macro), 376
 MCPWM0A (C++ class), 228
 MCPWM0B (C++ class), 228
 MCPWM1A (C++ class), 228
 MCPWM1B (C++ class), 228
 MCPWM2A (C++ class), 228
 MCPWM2B (C++ class), 228
 mcpwm_action_on_pwmxa_t (C++ type), 231
 mcpwm_action_on_pwmxb_t (C++ type), 231
 MCPWM_ACTIVE_HIGH_COMPLIMENT_MODE (C++ class), 232
 MCPWM_ACTIVE_HIGH_MODE (C++ class), 232
 MCPWM_ACTIVE_LOW_COMPLIMENT_MODE (C++ class), 232
 MCPWM_ACTIVE_LOW_MODE (C++ class), 232

MCPWM_ACTIVE_RED_FED_FROM_PWMXA (C++ class), 232
 MCPWM_ACTIVE_RED_FED_FROM_PWMXB (C++ class), 232
 MCPWM_BYPASS_FED (C++ class), 232
 MCPWM_BYPASS_RED (C++ class), 232
 MCPWM_CAP_0 (C++ class), 229
 MCPWM_CAP_1 (C++ class), 229
 MCPWM_CAP_2 (C++ class), 229
 mcpwm_capture_disable (C++ function), 225
 mcpwm_capture_enable (C++ function), 225
 mcpwm_capture_on_edge_t (C++ type), 232
 mcpwm_capture_signal_get_edge (C++ function), 225
 mcpwm_capture_signal_get_value (C++ function), 225
 mcpwm_capture_signal_t (C++ type), 232
 mcpwm_carrier_config_t (C++ class), 228
 mcpwm_carrier_config_t::carrier_duty (C++ member), 228
 mcpwm_carrier_config_t::carrier_ivt_mode (C++ member), 228
 mcpwm_carrier_config_t::carrier_os_mode (C++ member), 228
 mcpwm_carrier_config_t::carrier_period (C++ member), 228
 mcpwm_carrier_config_t::pulse_width_in_os (C++ member), 228
 mcpwm_carrier_disable (C++ function), 221
 mcpwm_carrier_enable (C++ function), 221
 mcpwm_carrier_init (C++ function), 220
 mcpwm_carrier_oneshot_mode_disable (C++ function), 222
 mcpwm_carrier_oneshot_mode_enable (C++ function), 222
 mcpwm_carrier_os_t (C++ type), 230
 MCPWM_CARRIER_OUT_IVT_DIS (C++ class), 230
 MCPWM_CARRIER_OUT_IVT_EN (C++ class), 230
 mcpwm_carrier_out_ivt_t (C++ type), 230
 mcpwm_carrier_output_invert (C++ function), 222
 mcpwm_carrier_set_duty_cycle (C++ function), 221
 mcpwm_carrier_set_period (C++ function), 221
 mcpwm_config_t (C++ class), 227
 mcpwm_config_t::cmpr_a (C++ member), 227
 mcpwm_config_t::cmpr_b (C++ member), 228
 mcpwm_config_t::counter_mode (C++ member), 228
 mcpwm_config_t::duty_mode (C++ member), 228
 mcpwm_config_t::frequency (C++ member), 227
 MCPWM_COUNTER_MAX (C++ class), 230
 mcpwm_counter_type_t (C++ type), 230
 mcpwm_deadtime_disable (C++ function), 223
 mcpwm_deadtime_enable (C++ function), 223
 MCPWM_DEADTIME_TYPE_MAX (C++ class), 232
 mcpwm_deadtime_type_t (C++ type), 232
 MCPWM_DOWN_COUNTER (C++ class), 230
 MCPWM_DUTY_MODE_0 (C++ class), 230

MCPWM_DUTY_MODE_1 (C++ class), 230	mcpwm_pin_config_t::mcpwm_cap1_in_num (C++ member), 227
MCPWM_DUTY_MODE_MAX (C++ class), 230	mcpwm_pin_config_t::mcpwm_cap2_in_num (C++ member), 227
mcpwm_duty_type_t (C++ type), 230	mcpwm_pin_config_t::mcpwm_fault0_in_num (C++ member), 227
MCPWMFAULT_0 (C++ class), 229	mcpwm_pin_config_t::mcpwm_fault1_in_num (C++ member), 227
MCPWMFAULT_1 (C++ class), 229	mcpwm_pin_config_t::mcpwm_fault2_in_num (C++ member), 227
MCPWMFAULT_2 (C++ class), 229	mcpwm_pin_config_t::mcpwm_sync0_in_num (C++ member), 227
mcpwm_fault_deinit (C++ function), 224	mcpwm_pin_config_t::mcpwm_sync1_in_num (C++ member), 227
mcpwm_fault_init (C++ function), 223	mcpwm_pin_config_t::mcpwm_sync2_in_num (C++ member), 227
mcpwm_fault_input_level_t (C++ type), 231	MCPWM_POS_EDGE (C++ class), 232
mcpwm_fault_set_cyc_mode (C++ function), 224	MCPWM_SELECT_CAP0 (C++ class), 232
mcpwm_fault_set_oneshot_mode (C++ function), 223	MCPWM_SELECT_CAP1 (C++ class), 232
mcpwm_fault_signal_t (C++ type), 231	MCPWM_SELECT_CAP2 (C++ class), 232
MCPWM_FORCE_MCPWMXA_HIGH (C++ class), 231	MCPWM_SELECT_F0 (C++ class), 231
MCPWM_FORCE_MCPWMXA_LOW (C++ class), 231	MCPWM_SELECT_F1 (C++ class), 231
MCPWM_FORCE_MCPWMXB_HIGH (C++ class), 231	MCPWM_SELECT_F2 (C++ class), 231
MCPWM_FORCE_MCPWMXB_LOW (C++ class), 231	MCPWM_SELECT_SYNC0 (C++ class), 230
mcpwm_get_duty (C++ function), 219	MCPWM_SELECT_SYNC1 (C++ class), 231
mcpwm_get_frequency (C++ function), 219	MCPWM_SELECT_SYNC2 (C++ class), 231
mcpwm_gpio_init (C++ function), 217	mcpwm_set_duty (C++ function), 218
MCPWM_HIGH_LEVEL_TGR (C++ class), 231	mcpwm_set_duty_in_us (C++ function), 218
mcpwm_init (C++ function), 217	mcpwm_set_duty_type (C++ function), 219
mcpwm_io_signals_t (C++ type), 228	mcpwm_set_frequency (C++ function), 218
mcpwm_isr_register (C++ function), 226	mcpwm_set_pin (C++ function), 217
MCPWM_LOW_LEVEL_TGR (C++ class), 231	mcpwm_set_signal_high (C++ function), 219
MCPWM_NEG_EDGE (C++ class), 232	mcpwm_set_signal_low (C++ function), 220
MCPWM_NO_CHANGE_IN_MCPWMXA (C++ class), 231	mcpwm_start (C++ function), 220
MCPWM_NO_CHANGE_IN_MCPWMXB (C++ class), 231	mcpwm_stop (C++ function), 220
MCPWM_ONESHOT_MODE_DIS (C++ class), 230	MCPWM_SYNC_0 (C++ class), 228
MCPWM_ONESHOT_MODE_EN (C++ class), 230	MCPWM_SYNC_1 (C++ class), 229
mcpwm_operator_t (C++ type), 229	MCPWM_SYNC_2 (C++ class), 229
MCPWM_OPR_A (C++ class), 229	mcpwm_sync_disable (C++ function), 226
MCPWM_OPR_B (C++ class), 229	mcpwm_sync_enable (C++ function), 226
MCPWM_OPR_MAX (C++ class), 230	mcpwm_sync_signal_t (C++ type), 230
mcpwm_pin_config_t (C++ class), 227	MCPWM_TIMER_0 (C++ class), 229
mcpwm_pin_config_t::mcpwm0a_out_num (C++ member), 227	MCPWM_TIMER_1 (C++ class), 229
mcpwm_pin_config_t::mcpwm0b_out_num (C++ member), 227	MCPWM_TIMER_2 (C++ class), 229
mcpwm_pin_config_t::mcpwm1a_out_num (C++ member), 227	MCPWM_TIMER_MAX (C++ class), 229
mcpwm_pin_config_t::mcpwm1b_out_num (C++ member), 227	mcpwm_timer_t (C++ type), 229
mcpwm_pin_config_t::mcpwm2a_out_num (C++ member), 227	MCPWM_TOG_MCPWMXA (C++ class), 231
mcpwm_pin_config_t::mcpwm2b_out_num (C++ member), 227	MCPWM_TOG_MCPWMXB (C++ class), 231
mcpwm_pin_config_t::mcpwm_cap0_in_num (C++ member), 227	MCPWM_UNIT_0 (C++ class), 229
	MCPWM_UNIT_1 (C++ class), 229
	MCPWM_UNIT_MAX (C++ class), 229
	mcpwm_unit_t (C++ type), 229
	MCPWM_UP_COUNTER (C++ class), 230
	MCPWM_UP_DOWN_COUNTER (C++ class), 230

mdns_free (C++ function), 323
 mdns_init (C++ function), 323
 mdns_query (C++ function), 326
 mdns_query_end (C++ function), 326
 mdns_result_free (C++ function), 326
 mdns_result_get (C++ function), 326
 mdns_result_get_count (C++ function), 326
 mdns_result_s (C++ class), 327
 mdns_result_s::addr (C++ member), 327
 mdns_result_s::addrv6 (C++ member), 327
 mdns_result_s::host (C++ member), 327
 mdns_result_s::instance (C++ member), 327
 mdns_result_s::next (C++ member), 327
 mdns_result_s::port (C++ member), 327
 mdns_result_s::priority (C++ member), 327
 mdns_result_s::txt (C++ member), 327
 mdns_result_s::weight (C++ member), 327
 mdns_result_t (C++ type), 327
 mdns_server_t (C++ type), 327
 mdns_service_add (C++ function), 324
 mdns_service_instance_set (C++ function), 324
 mdns_service_port_set (C++ function), 325
 mdns_service_remove (C++ function), 324
 mdns_service_remove_all (C++ function), 325
 mdns_service_txt_set (C++ function), 325
 mdns_set_hostname (C++ function), 323
 mdns_set_instance (C++ function), 323
 multi_heap_check (C++ function), 380
 multi_heap_dump (C++ function), 379
 multi_heap_free (C++ function), 378
 multi_heap_free_size (C++ function), 380
 multi_heap_get_allocated_size (C++ function), 379
 multi_heap_get_info (C++ function), 380
 multi_heap_handle_t (C++ type), 381
 multi_heap_info_t (C++ class), 381
 multi_heap_info_t::allocated_blocks (C++ member), 381
 multi_heap_info_t::free_blocks (C++ member), 381
 multi_heap_info_t::largest_free_block (C++ member), 381
 multi_heap_info_t::minimum_free_bytes (C++ member), 381
 multi_heap_info_t::total_allocated_bytes (C++ member), 381
 multi_heap_info_t::total_blocks (C++ member), 381
 multi_heap_info_t::total_free_bytes (C++ member), 381
 multi_heap_malloc (C++ function), 378
 multi_heap_minimum_free_size (C++ function), 380
 multi_heap_realloc (C++ function), 378
 multi_heap_register (C++ function), 379
 multi_heap_set_lock (C++ function), 379

N

nvs_close (C++ function), 353
 nvs_commit (C++ function), 353

NVS_DEFAULT_PART_NAME (C macro), 354
 nvs_erase_all (C++ function), 353
 nvs_erase_key (C++ function), 352
 nvs_flash_erase (C++ function), 348
 nvs_flash_erase_partition (C++ function), 348
 nvs_flash_init (C++ function), 347
 nvs_flash_init_partition (C++ function), 347
 nvs_get_blob (C++ function), 351
 nvs_get_i16 (C++ function), 350
 nvs_get_i32 (C++ function), 350
 nvs_get_i64 (C++ function), 350
 nvs_get_i8 (C++ function), 349
 nvs_get_str (C++ function), 350
 nvs_get_u16 (C++ function), 350
 nvs_get_u32 (C++ function), 350
 nvs_get_u64 (C++ function), 350
 nvs_get_u8 (C++ function), 350
 nvs_handle (C++ type), 355
 nvs_open (C++ function), 351
 nvs_open_from_partition (C++ function), 351
 nvs_open_mode (C++ type), 355
 NVS_READONLY (C++ class), 355
 NVS_READWRITE (C++ class), 355
 nvs_set_blob (C++ function), 352
 nvs_set_i16 (C++ function), 349
 nvs_set_i32 (C++ function), 349
 nvs_set_i64 (C++ function), 349
 nvs_set_i8 (C++ function), 348
 nvs_set_str (C++ function), 349
 nvs_set_u16 (C++ function), 349
 nvs_set_u32 (C++ function), 349
 nvs_set_u64 (C++ function), 349
 nvs_set_u8 (C++ function), 349

O

OTA_SIZE_UNKNOWN (C macro), 400

P

PCNT_CHANNEL_0 (C++ class), 239
 PCNT_CHANNEL_1 (C++ class), 239
 PCNT_CHANNEL_MAX (C++ class), 239
 pcnt_channel_t (C++ type), 239
 pcnt_config_t (C++ class), 238
 pcnt_config_t::channel (C++ member), 238
 pcnt_config_t::counter_h_lim (C++ member), 238
 pcnt_config_t::counter_l_lim (C++ member), 238
 pcnt_config_t::ctrl_gpio_num (C++ member), 238
 pcnt_config_t::hcrtl_mode (C++ member), 238
 pcnt_config_t::lctrl_mode (C++ member), 238
 pcnt_config_t::neg_mode (C++ member), 238
 pcnt_config_t::pos_mode (C++ member), 238
 pcnt_config_t::pulse_gpio_num (C++ member), 238
 pcnt_config_t::unit (C++ member), 238
 PCNT_COUNT_DEC (C++ class), 239

PCNT_COUNT_DIS (C++ class), 239	PHY10 (C++ class), 161
PCNT_COUNT_INC (C++ class), 239	PHY11 (C++ class), 161
PCNT_COUNT_MAX (C++ class), 239	PHY12 (C++ class), 161
pcnt_count_mode_t (C++ type), 239	PHY13 (C++ class), 161
pcnt_counter_clear (C++ function), 234	PHY14 (C++ class), 161
pcnt_counter_pause (C++ function), 233	PHY15 (C++ class), 161
pcnt_counter_resume (C++ function), 234	PHY16 (C++ class), 161
pcnt_ctrl_mode_t (C++ type), 238	PHY17 (C++ class), 161
pcnt_event_disable (C++ function), 235	PHY18 (C++ class), 161
pcnt_event_enable (C++ function), 234	PHY19 (C++ class), 161
PCNT_EVT_H_LIM (C++ class), 240	PHY2 (C++ class), 160
PCNT_EVT_L_LIM (C++ class), 239	PHY20 (C++ class), 161
PCNT_EVT_MAX (C++ class), 240	PHY21 (C++ class), 161
PCNT_EVT_THRES_0 (C++ class), 240	PHY22 (C++ class), 161
PCNT_EVT_THRES_1 (C++ class), 240	PHY23 (C++ class), 161
pcnt_evt_type_t (C++ type), 239	PHY24 (C++ class), 161
PCNT_EVT_ZERO (C++ class), 240	PHY25 (C++ class), 161
pcnt_filter_disable (C++ function), 236	PHY26 (C++ class), 161
pcnt_filter_enable (C++ function), 236	PHY27 (C++ class), 161
pcnt_get_counter_value (C++ function), 233	PHY28 (C++ class), 161
pcnt_get_event_value (C++ function), 235	PHY29 (C++ class), 161
pcnt_get_filter_value (C++ function), 237	PHY3 (C++ class), 160
pcnt_intr_disable (C++ function), 234	PHY30 (C++ class), 161
pcnt_intr_enable (C++ function), 234	PHY31 (C++ class), 161
pcnt_isr_handle_t (C++ type), 238	PHY4 (C++ class), 160
pcnt_isr_register (C++ function), 235	PHY5 (C++ class), 160
PCNT_MODE_DISABLE (C++ class), 239	PHY6 (C++ class), 160
PCNT_MODE_KEEP (C++ class), 238	PHY7 (C++ class), 160
PCNT_MODE_MAX (C++ class), 239	PHY8 (C++ class), 160
PCNT_MODE_REVERSE (C++ class), 238	PHY9 (C++ class), 161
PCNT_PIN_NOT_USED (C macro), 238	phy_lan8720_check_phy_init (C++ function), 163
pcnt_set_event_value (C++ function), 235	phy_lan8720_default_ethernet_config (C++ member), 156
pcnt_set_filter_value (C++ function), 237	phy_lan8720_dump_registers (C++ function), 163
pcnt_set_mode (C++ function), 237	phy_lan8720_get_duplex_mode (C++ function), 163
pcnt_set_pin (C++ function), 236	phy_lan8720_get_speed_mode (C++ function), 163
PCNT_UNIT_0 (C++ class), 239	phy_lan8720_init (C++ function), 163
PCNT_UNIT_1 (C++ class), 239	phy_lan8720_power_enable (C++ function), 163
PCNT_UNIT_2 (C++ class), 239	phy_mii_check_link_status (C++ function), 162
PCNT_UNIT_3 (C++ class), 239	phy_mii_enable_flow_ctrl (C++ function), 162
PCNT_UNIT_4 (C++ class), 239	phy_mii_get_partner_pause_enable (C++ function), 162
PCNT_UNIT_5 (C++ class), 239	phy_rmii_configure_data_interface_pins (C++ function), 161
PCNT_UNIT_6 (C++ class), 239	phy_rmii_smi_configure_pins (C++ function), 162
PCNT_UNIT_7 (C++ class), 239	phy_tlk110_check_phy_init (C++ function), 162
pcnt_unit_config (C++ function), 233	phy_tlk110_default_ethernet_config (C++ member), 156
PCNT_UNIT_MAX (C++ class), 239	phy_tlk110_dump_registers (C++ function), 162
pcnt_unit_t (C++ type), 239	phy_tlk110_get_duplex_mode (C++ function), 162
PDM_PCM_CONV_DISABLE (C++ class), 205	phy_tlk110_get_speed_mode (C++ function), 162
PDM_PCM_CONV_ENABLE (C++ class), 205	phy_tlk110_init (C++ function), 162
pdm_pcm_conv_t (C++ type), 205	phy_tlk110_power_enable (C++ function), 162
PDM_SAMPLE_RATE_RATIO_128 (C++ class), 205	
PDM_SAMPLE_RATE_RATIO_64 (C++ class), 205	
pdm_sample_rate_ratio_t (C++ type), 205	
PHY0 (C++ class), 160	
PHY1 (C++ class), 160	

R1 (C macro), 593
 R2 (C macro), 593
 R3 (C macro), 593
 RMT_BASECLK_APB (C++ class), 253
 RMT_BASECLK_MAX (C++ class), 253
 RMT_BASECLK_REF (C++ class), 253
 RMT_CARRIER_LEVEL_HIGH (C++ class), 253
 RMT_CARRIER_LEVEL_LOW (C++ class), 253
 RMT_CARRIER_LEVEL_MAX (C++ class), 253
 rmt_carrier_level_t (C++ type), 253
 RMT_CHANNEL_0 (C++ class), 252
 RMT_CHANNEL_1 (C++ class), 252
 RMT_CHANNEL_2 (C++ class), 252
 RMT_CHANNEL_3 (C++ class), 252
 RMT_CHANNEL_4 (C++ class), 252
 RMT_CHANNEL_5 (C++ class), 252
 RMT_CHANNEL_6 (C++ class), 252
 RMT_CHANNEL_7 (C++ class), 252
 RMT_CHANNEL_MAX (C++ class), 252
 rmt_channel_t (C++ type), 252
 rmt_clr_intr_enable_mask (C++ function), 246
 rmt_config (C++ function), 248
 rmt_config_t (C++ class), 251
 rmt_config_t::channel (C++ member), 251
 rmt_config_t::clk_div (C++ member), 251
 rmt_config_t::gpio_num (C++ member), 252
 rmt_config_t::mem_block_num (C++ member), 252
 rmt_config_t::rmt_mode (C++ member), 251
 rmt_config_t::rx_config (C++ member), 252
 rmt_config_t::tx_config (C++ member), 252
 RMT_DATA_MODE_FIFO (C++ class), 253
 RMT_DATA_MODE_MAX (C++ class), 253
 RMT_DATA_MODE_MEM (C++ class), 253
 rmt_data_mode_t (C++ type), 253
 rmt_driver_install (C++ function), 249
 rmt_driver_uninstall (C++ function), 249
 rmt_fill_tx_items (C++ function), 249
 rmt_get_clk_div (C++ function), 240
 rmt_get_mem_block_num (C++ function), 242
 rmt_get_mem_pd (C++ function), 243
 rmt_get_memory_owner (C++ function), 244
 rmt_get_ringbuf_handle (C++ function), 250
 rmt_get_rx_idle_thresh (C++ function), 241
 rmt_get_source_clk (C++ function), 246
 rmt_get_status (C++ function), 246
 rmt_get_tx_loop_mode (C++ function), 245
 RMT_IDLE_LEVEL_HIGH (C++ class), 253
 RMT_IDLE_LEVEL_LOW (C++ class), 253
 RMT_IDLE_LEVEL_MAX (C++ class), 253
 rmt_idle_level_t (C++ type), 253
 rmt_isr_deregister (C++ function), 248
 rmt_isr_handle_t (C++ type), 252
 rmt_isr_register (C++ function), 248
 RMT_MEM_BLOCK_BYTE_NUM (C macro), 252
 RMT_MEM_ITEM_NUM (C macro), 252
 RMT_MEM_OWNER_MAX (C++ class), 253
 RMT_MEM_OWNER_RX (C++ class), 253
 rmt_mem_owner_t (C++ type), 252
 RMT_MEM_OWNER_TX (C++ class), 252
 rmt_memory_rw_rst (C++ function), 244
 RMT_MODE_MAX (C++ class), 253
 RMT_MODE_RX (C++ class), 253
 rmt_mode_t (C++ type), 253
 RMT_MODE_TX (C++ class), 253
 rmt_rx_config_t (C++ class), 251
 rmt_rx_config_t::filter_en (C++ member), 251
 rmt_rx_config_t::filter_ticks_thresh (C++ member), 251
 rmt_rx_config_t::idle_threshold (C++ member), 251
 rmt_rx_start (C++ function), 243
 rmt_rx_stop (C++ function), 243
 rmt_set_clk_div (C++ function), 240
 rmt_set_err_intr_en (C++ function), 247
 rmt_set_idle_level (C++ function), 246
 rmt_set_intr_enable_mask (C++ function), 246
 rmt_set_mem_block_num (C++ function), 241
 rmt_set_mem_pd (C++ function), 242
 rmt_set_memory_owner (C++ function), 244
 rmt_set_pin (C++ function), 248
 rmt_set_rx_filter (C++ function), 245
 rmt_set_rx_idle_thresh (C++ function), 241
 rmt_set_rx_intr_en (C++ function), 247
 rmt_set_source_clk (C++ function), 245
 rmt_set_tx_carrier (C++ function), 242
 rmt_set_tx_intr_en (C++ function), 247
 rmt_set_tx_loop_mode (C++ function), 244
 rmt_set_tx_thr_intr_en (C++ function), 247
 rmt_source_clk_t (C++ type), 253
 rmt_tx_config_t (C++ class), 251
 rmt_tx_config_t::carrier_duty_percent (C++ member),
 251
 rmt_tx_config_t::carrier_en (C++ member), 251
 rmt_tx_config_t::carrier_freq_hz (C++ member), 251
 rmt_tx_config_t::carrier_level (C++ member), 251
 rmt_tx_config_t::idle_level (C++ member), 251
 rmt_tx_config_t::idle_output_en (C++ member), 251
 rmt_tx_config_t::loop_en (C++ member), 251
 rmt_tx_start (C++ function), 243
 rmt_tx_stop (C++ function), 243
 rmt_wait_tx_done (C++ function), 250
 rmt_write_items (C++ function), 249
 rtc_gpio_deinit (C++ function), 183
 rtc_gpio_desc_t (C++ class), 186
 rtc_gpio_desc_t::drv_s (C++ member), 186
 rtc_gpio_desc_t::drv_v (C++ member), 186
 rtc_gpio_desc_t::func (C++ member), 186
 rtc_gpio_desc_t::hold (C++ member), 186
 rtc_gpio_desc_t::hold_force (C++ member), 186
 rtc_gpio_desc_t::ie (C++ member), 186

rtc_gpio_desc_t::mux (C++ member), 186
 rtc_gpio_desc_t::pulldown (C++ member), 186
 rtc_gpio_desc_t::pullup (C++ member), 186
 rtc_gpio_desc_t::reg (C++ member), 186
 rtc_gpio_desc_t::rtc_num (C++ member), 186
 rtc_gpio_desc_t::slpie (C++ member), 186
 rtc_gpio_desc_t::slpsel (C++ member), 186
 rtc_gpio_force_hold_dis_all (C++ function), 185
 rtc_gpio_get_drive_capability (C++ function), 185
 rtc_gpio_get_level (C++ function), 183
 rtc_gpio_hold_dis (C++ function), 185
 rtc_gpio_hold_en (C++ function), 185
 rtc_gpio_init (C++ function), 182
 RTC_GPIO_IS_VALID_GPIO (C macro), 187
 rtc_gpio_is_valid_gpio (C++ function), 182
 RTC_GPIO_MODE_DISABLED (C++ class), 187
 RTC_GPIO_MODE_INPUT_ONLY (C++ class), 187
 RTC_GPIO_MODE_INPUT_OUTUT (C++ class), 187
 RTC_GPIO_MODE_OUTPUT_ONLY (C++ class), 187
 rtc_gpio_mode_t (C++ type), 187
 rtc_gpio_pulldown_dis (C++ function), 184
 rtc_gpio_pulldown_en (C++ function), 184
 rtc_gpio_pullup_dis (C++ function), 184
 rtc_gpio_pullup_en (C++ function), 184
 rtc_gpio_set_direction (C++ function), 183
 rtc_gpio_set_drive_capability (C++ function), 185
 rtc_gpio_set_level (C++ function), 183
 RTC_SLOW_MEM (C macro), 596

S

sc_callback_t (C++ type), 66
 SC_STATUS_FIND_CHANNEL (C++ class), 67
 SC_STATUS_GETTING_SSID_PSWD (C++ class), 67
 SC_STATUS_LINK (C++ class), 67
 SC_STATUS_LINK_OVER (C++ class), 67
 SC_STATUS_WAIT (C++ class), 67
 SC_TYPE_AIRKISS (C++ class), 67
 SC_TYPE_ESPTOUCH (C++ class), 67
 SC_TYPE_ESPTOUCH_AIRKISS (C++ class), 67
 sdmmc_card_init (C++ function), 257
 sdmmc_card_t (C++ class), 256
 sdmmc_card_t::cid (C++ member), 256
 sdmmc_card_t::csd (C++ member), 256
 sdmmc_card_t::host (C++ member), 256
 sdmmc_card_t::ocr (C++ member), 256
 sdmmc_card_t::rca (C++ member), 256
 sdmmc_card_t::scr (C++ member), 256
 sdmmc_cid_t (C++ class), 257
 sdmmc_cid_t::date (C++ member), 257
 sdmmc_cid_t::mfg_id (C++ member), 257
 sdmmc_cid_t::name (C++ member), 257
 sdmmc_cid_t::oem_id (C++ member), 257
 sdmmc_cid_t::revision (C++ member), 257
 sdmmc_cid_t::serial (C++ member), 257

sdmmc_command_t (C++ class), 255
 sdmmc_command_t::arg (C++ member), 255
 sdmmc_command_t::blklen (C++ member), 256
 sdmmc_command_t::data (C++ member), 256
 sdmmc_command_t::datalen (C++ member), 256
 sdmmc_command_t::error (C++ member), 256
 sdmmc_command_t::flags (C++ member), 256
 sdmmc_command_t::opcode (C++ member), 255
 sdmmc_command_t::response (C++ member), 255
 sdmmc_csd_t (C++ class), 256
 sdmmc_csd_t::capacity (C++ member), 256
 sdmmc_csd_t::card_command_class (C++ member), 256
 sdmmc_csd_t::csd_ver (C++ member), 256
 sdmmc_csd_t::mmc_ver (C++ member), 256
 sdmmc_csd_t::read_block_len (C++ member), 256
 sdmmc_csd_t::sector_size (C++ member), 256
 sdmmc_csd_t::tr_speed (C++ member), 257
 SDMMC_FREQ_DEFAULT (C macro), 255
 SDMMC_FREQ_HIGHSPEED (C macro), 255
 SDMMC_FREQ_PROBING (C macro), 255
 SDMMC_HOST_DEFAULT (C macro), 259
 sdmmc_host_deinit (C++ function), 261
 sdmmc_host_do_transaction (C++ function), 260
 SDMMC_HOST_FLAG_1BIT (C macro), 255
 SDMMC_HOST_FLAG_4BIT (C macro), 255
 SDMMC_HOST_FLAG_8BIT (C macro), 255
 SDMMC_HOST_FLAG_SPI (C macro), 255
 sdmmc_host_init (C++ function), 258
 sdmmc_host_init_slot (C++ function), 259
 sdmmc_host_set_bus_width (C++ function), 260
 sdmmc_host_set_card_clk (C++ function), 260
 SDMMC_HOST_SLOT_0 (C macro), 259
 SDMMC_HOST_SLOT_1 (C macro), 259
 sdmmc_host_t (C++ class), 254
 sdmmc_host_t::deinit (C++ member), 255
 sdmmc_host_t::do_transaction (C++ member), 255
 sdmmc_host_t::flags (C++ member), 255
 sdmmc_host_t::init (C++ member), 255
 sdmmc_host_t::io_voltage (C++ member), 255
 sdmmc_host_t::max_freq_khz (C++ member), 255
 sdmmc_host_t::set_bus_width (C++ member), 255
 sdmmc_host_t::set_card_clk (C++ member), 255
 sdmmc_host_t::slot (C++ member), 255
 sdmmc_read_sectors (C++ function), 258
 sdmmc_scr_t (C++ class), 257
 sdmmc_scr_t::bus_width (C++ member), 257
 sdmmc_scr_t::sd_spec (C++ member), 257
 SDMMC_SLOT_CONFIG_DEFAULT (C macro), 260
 sdmmc_slot_config_t (C++ class), 259
 sdmmc_slot_config_t::gpio_cd (C++ member), 259
 sdmmc_slot_config_t::gpio_wp (C++ member), 259
 sdmmc_slot_config_t::width (C++ member), 259
 SDMMC_SLOT_NO_CD (C macro), 259
 SDMMC_SLOT_NO_WP (C macro), 259

SDMMC_SLOT_WIDTH_DEFAULT (C macro), 259
 sdm mmc_write_sectors (C++ function), 257
 SDSPI_HOST_DEFAULT (C macro), 261
 sdspi_host_deinit (C++ function), 263
 sdspi_host_do_transaction (C++ function), 263
 sdspi_host_init (C++ function), 261
 sdspi_host_init_slot (C++ function), 261
 sdspi_host_set_card_clk (C++ function), 262
 SDSPI_SLOT_CONFIG_DEFAULT (C macro), 262
 sdspi_slot_config_t (C++ class), 262
 sdspi_slot_config_t::dma_channel (C++ member), 262
 sdspi_slot_config_t::gpio_cd (C++ member), 262
 sdspi_slot_config_t::gpio_cs (C++ member), 262
 sdspi_slot_config_t::gpio_miso (C++ member), 262
 sdspi_slot_config_t::gpio_mosi (C++ member), 262
 sdspi_slot_config_t::gpio_sck (C++ member), 262
 sdspi_slot_config_t::gpio_wp (C++ member), 262
 SDSPI_SLOT_NO_CD (C macro), 262
 SDSPI_SLOT_NO_WP (C macro), 262
 SIGMADELTA_CHANNEL_0 (C++ class), 265
 SIGMADELTA_CHANNEL_1 (C++ class), 265
 SIGMADELTA_CHANNEL_2 (C++ class), 265
 SIGMADELTA_CHANNEL_3 (C++ class), 265
 SIGMADELTA_CHANNEL_4 (C++ class), 265
 SIGMADELTA_CHANNEL_5 (C++ class), 265
 SIGMADELTA_CHANNEL_6 (C++ class), 265
 SIGMADELTA_CHANNEL_7 (C++ class), 265
 SIGMADELTA_CHANNEL_MAX (C++ class), 265
 sigmadelta_channel_t (C++ type), 265
 sigmadelta_config (C++ function), 264
 sigmadelta_config_t (C++ class), 265
 sigmadelta_config_t::channel (C++ member), 265
 sigmadelta_config_t::sigmadelta_duty (C++ member), 265
 sigmadelta_config_t::sigmadelta_gpio (C++ member), 265
 sigmadelta_config_t::sigmadelta_prescale (C++ member), 265
 sigmadelta_set_duty (C++ function), 264
 sigmadelta_set_pin (C++ function), 264
 sigmadelta_set_prescale (C++ function), 264
 slave_transaction_cb_t (C++ type), 282
 smartconfig_status_t (C++ type), 67
 smartconfig_type_t (C++ type), 67
 spi_bus_add_device (C++ function), 273
 spi_bus_config_t (C++ class), 271
 spi_bus_config_t::max_transfer_sz (C++ member), 271
 spi_bus_config_t::miso_io_num (C++ member), 271
 spi_bus_config_t::mosi_io_num (C++ member), 271
 spi_bus_config_t::quadhd_io_num (C++ member), 271
 spi_bus_config_t::quadwp_io_num (C++ member), 271
 spi_bus_config_t::sclk_io_num (C++ member), 271
 spi_bus_free (C++ function), 273
 spi_bus_initialize (C++ function), 272
 spi_bus_remove_device (C++ function), 273
 SPI_DEVICE_3WIRE (C macro), 276
 SPI_DEVICE_BIT_LSBFIRST (C macro), 276
 SPI_DEVICE_CLK_AS_CS (C macro), 276
 spi_device_get_trans_result (C++ function), 274
 SPI_DEVICE_HALFDUPLEX (C macro), 276
 spi_device_handle_t (C++ type), 277
 spi_device_interface_config_t (C++ class), 275
 spi_device_interface_config_t::address_bits (C++ member), 275
 spi_device_interface_config_t::clock_speed_hz (C++ member), 275
 spi_device_interface_config_t::command_bits (C++ member), 275
 spi_device_interface_config_t::cs_ena_posttrans (C++ member), 275
 spi_device_interface_config_t::cs_ena_pretrans (C++ member), 275
 spi_device_interface_config_t::dummy_bits (C++ member), 275
 spi_device_interface_config_t::duty_cycle_pos (C++ member), 275
 spi_device_interface_config_t::flags (C++ member), 275
 spi_device_interface_config_t::mode (C++ member), 275
 spi_device_interface_config_t::post_cb (C++ member), 275
 spi_device_interface_config_t::pre_cb (C++ member), 275
 spi_device_interface_config_t::queue_size (C++ member), 275
 spi_device_interface_config_t::spics_io_num (C++ member), 275
 SPI_DEVICE_POSITIVE_CS (C macro), 276
 spi_device_queue_trans (C++ function), 274
 SPI_DEVICE_RXBIT_LSBFIRST (C macro), 276
 spi_device_transmit (C++ function), 274
 SPI_DEVICE_TXBIT_LSBFIRST (C macro), 276
 spi_flash_cache2phys (C++ function), 333
 SPI_FLASH_CACHE2PHYS_FAIL (C macro), 335
 spi_flash_cache_enabled (C++ function), 334
 spi_flash_erase_range (C++ function), 331
 spi_flash_erase_sector (C++ function), 331
 spi_flash_get_chip_size (C++ function), 331
 spi_flash_guard_end_func_t (C++ type), 335
 spi_flash_guard_funcs_t (C++ class), 334
 spi_flash_guard_funcs_t::end (C++ member), 335
 spi_flash_guard_funcs_t::op_lock (C++ member), 335
 spi_flash_guard_funcs_t::op_unlock (C++ member), 335
 spi_flash_guard_funcs_t::start (C++ member), 335
 spi_flash_guard_set (C++ function), 334
 spi_flash_guard_start_func_t (C++ type), 335
 spi_flash_init (C++ function), 331
 spi_flash mmap (C++ function), 332
 SPI_FLASH_MMAP_DATA (C++ class), 336

spi_flash_mmap_dump (C++ function), 333
 spi_flash_mmap_handle_t (C++ type), 335
 SPI_FLASH_MMAP_INST (C++ class), 336
 spi_flash_mmap_memory_t (C++ type), 336
 spi_flash_mmap_pages (C++ function), 333
 SPI_FLASH_MMU_PAGE_SIZE (C macro), 335
 spi_flash_munmap (C++ function), 333
 spi_flash_op_lock_func_t (C++ type), 335
 spi_flash_op_unlock_func_t (C++ type), 335
 spi_flash_phys2cache (C++ function), 333
 spi_flash_read (C++ function), 332
 spi_flash_read_encrypted (C++ function), 332
 SPI_FLASH_SEC_SIZE (C macro), 335
 spi_flash_write (C++ function), 331
 spi_flash_write_encrypted (C++ function), 331
 SPI_HOST (C++ class), 272
 spi_host_device_t (C++ type), 272
 SPI_MAX_DMA_LEN (C macro), 271
 SPI_SLAVE_BIT_LSBFIRST (C macro), 281
 spi_slave_free (C++ function), 279
 spi_slave_get_trans_result (C++ function), 280
 spi_slave_initialize (C++ function), 279
 spi_slave_interface_config_t (C++ class), 281
 spi_slave_interface_config_t::flags (C++ member), 281
 spi_slave_interface_config_t::mode (C++ member), 281
 spi_slave_interface_config_t::post_setup_cb (C++ member), 281
 spi_slave_interface_config_t::post_trans_cb (C++ member), 281
 spi_slave_interface_config_t::queue_size (C++ member), 281
 spi_slave_interface_config_t::spics_io_num (C++ member), 281
 spi_slave_queue_trans (C++ function), 279
 SPI_SLAVE_RXBIT_LSBFIRST (C macro), 281
 spi_slave_transaction_t (C++ class), 281
 spi_slave_transaction_t (C++ type), 282
 spi_slave_transaction_t::length (C++ member), 281
 spi_slave_transaction_t::rx_buffer (C++ member), 281
 spi_slave_transaction_t::tx_buffer (C++ member), 281
 spi_slave_transaction_t::user (C++ member), 281
 spi_slave_transmit (C++ function), 280
 SPI_SLAVE_TXBIT_LSBFIRST (C macro), 281
 SPI_TRANS_MODE_DIO (C macro), 276
 SPI_TRANS_MODE_DIOQIO_ADDR (C macro), 277
 SPI_TRANS_MODE_QIO (C macro), 277
 SPI_TRANS_USE_RXDATA (C macro), 277
 SPI_TRANS_USE_TXDATA (C macro), 277
 spi_transaction_t (C++ class), 275
 spi_transaction_t (C++ type), 277
 spi_transaction_t::addr (C++ member), 276
 spi_transaction_t::cmd (C++ member), 275
 spi_transaction_t::flags (C++ member), 275
 spi_transaction_t::length (C++ member), 276

spi_transaction_t::rx_buffer (C++ member), 276
 spi_transaction_t::rx_data (C++ member), 276
 spi_transaction_t::rxlength (C++ member), 276
 spi_transaction_t::tx_buffer (C++ member), 276
 spi_transaction_t::tx_data (C++ member), 276
 spi_transaction_t::user (C++ member), 276
 spicommon_bus_free_io (C++ function), 269
 spicommon_bus_initialize_io (C++ function), 269
 SPICOMMON_BUSFLAG_MASTER (C macro), 271
 SPICOMMON_BUSFLAG_QUAD (C macro), 272
 SPICOMMON_BUSFLAG_SLAVE (C macro), 271
 spicommon_cs_free (C++ function), 269
 spicommon_cs_initialize (C++ function), 269
 spicommon_dma_chan_claim (C++ function), 268
 spicommon_dma_chan_free (C++ function), 268
 spicommon_dmaworkaround_idle (C++ function), 271
 spicommon_dmaworkaround_req_reset (C++ function), 270
 spicommon_dmaworkaround_reset_in_progress (C++ function), 270
 spicommon_dmaworkaround_transfer_active (C++ function), 271
 spicommon_hw_for_host (C++ function), 270
 spicommon_irqsource_for_host (C++ function), 270
 spicommon_periph_claim (C++ function), 268
 spicommon_periph_free (C++ function), 268
 spicommon_setup_dma_desc_links (C++ function), 270

T

TIMER_0 (C++ class), 288
 TIMER_1 (C++ class), 288
 TIMER_ALARM_DIS (C++ class), 288
 TIMER_ALARM_EN (C++ class), 289
 TIMER_ALARM_MAX (C++ class), 289
 timer_alarm_t (C++ type), 288
 TIMER_AUTORELOAD_DIS (C++ class), 289
 TIMER_AUTORELOAD_EN (C++ class), 289
 TIMER_AUTORELOAD_MAX (C++ class), 289
 timer_autoreload_t (C++ type), 289
 TIMER_BASE_CLK (C macro), 287
 timer_config_t (C++ class), 287
 timer_config_t::alarm_en (C++ member), 287
 timer_config_t::auto_reload (C++ member), 287
 timer_config_t::counter_dir (C++ member), 287
 timer_config_t::counter_en (C++ member), 287
 timer_config_t::divider (C++ member), 287
 timer_config_t::intr_type (C++ member), 287
 timer_count_dir_t (C++ type), 288
 TIMER_COUNT_DOWN (C++ class), 288
 TIMER_COUNT_MAX (C++ class), 288
 TIMER_COUNT_UP (C++ class), 288
 timer_disable_intr (C++ function), 287
 timer_enable_intr (C++ function), 287
 timer_get_alarm_value (C++ function), 284

timer_get_config (C++ function), 286
 timer_get_counter_time_sec (C++ function), 282
 timer_get_counter_value (C++ function), 282
 TIMER_GROUP_0 (C++ class), 288
 TIMER_GROUP_1 (C++ class), 288
 timer_group_intr_disable (C++ function), 286
 timer_group_intr_enable (C++ function), 286
 TIMER_GROUP_MAX (C++ class), 288
 timer_group_t (C++ type), 288
 timer_idx_t (C++ type), 288
 timer_init (C++ function), 285
 TIMER_INTR_LEVEL (C++ class), 289
 TIMER_INTR_MAX (C++ class), 289
 timer_intr_mode_t (C++ type), 289
 timer_isr_handle_t (C++ type), 288
 timer_isr_register (C++ function), 285
 TIMER_MAX (C++ class), 288
 TIMER_PAUSE (C++ class), 288
 timer_pause (C++ function), 283
 timer_set_alarm (C++ function), 285
 timer_set_alarm_value (C++ function), 284
 timer_set_auto_reload (C++ function), 284
 timer_set_counter_mode (C++ function), 283
 timer_set_counter_value (C++ function), 283
 timer_set_divider (C++ function), 284
 TIMER_START (C++ class), 288
 timer_start (C++ function), 283
 timer_start_t (C++ type), 288
 touch_cnt_slope_t (C++ type), 303
 TOUCH_FSM_MODE_DEFAULT (C macro), 301
 TOUCH_FSM_MODE_MAX (C++ class), 304
 TOUCH_FSM_MODE_SW (C++ class), 304
 touch_fsm_mode_t (C++ type), 303
 TOUCH_FSM_MODE_TIMER (C++ class), 304
 touch_high_volt_t (C++ type), 302
 TOUCH_HVOLT_2V4 (C++ class), 302
 TOUCH_HVOLT_2V5 (C++ class), 302
 TOUCH_HVOLT_2V6 (C++ class), 302
 TOUCH_HVOLT_2V7 (C++ class), 302
 TOUCH_HVOLT_ATTEN_0V (C++ class), 302
 TOUCH_HVOLT_ATTEN_0V5 (C++ class), 302
 TOUCH_HVOLT_ATTEN_1V (C++ class), 302
 TOUCH_HVOLT_ATTEN_1V5 (C++ class), 302
 TOUCH_HVOLT_ATTEN_KEEP (C++ class), 302
 TOUCH_HVOLT_ATTEN_MAX (C++ class), 302
 TOUCH_HVOLT_KEEP (C++ class), 302
 TOUCH_HVOLT_MAX (C++ class), 302
 touch_isr_handle_t (C++ type), 301
 touch_low_volt_t (C++ type), 302
 TOUCH_LVOLT_0V5 (C++ class), 302
 TOUCH_LVOLT_0V6 (C++ class), 302
 TOUCH_LVOLT_0V7 (C++ class), 302
 TOUCH_LVOLT_0V8 (C++ class), 302
 TOUCH_LVOLT_KEEP (C++ class), 302

TOUCH_LVOLT_MAX (C++ class), 302
 TOUCH_PAD_BIT_MASK_MAX (C macro), 301
 touch_pad_clear_group_mask (C++ function), 299
 touch_pad_clear_status (C++ function), 299
 touch_pad_config (C++ function), 293
 touch_pad_deinit (C++ function), 293
 touch_pad_filter_delete (C++ function), 300
 touch_pad_filter_start (C++ function), 300
 touch_pad_filter_stop (C++ function), 300
 touch_pad_get_cnt_mode (C++ function), 296
 touch_pad_get_filter_period (C++ function), 300
 touch_pad_get_fsm_mode (C++ function), 297
 touch_pad_get_group_mask (C++ function), 298
 touch_pad_get_meas_time (C++ function), 295
 touch_pad_get_status (C++ function), 299
 touch_pad_get_thresh (C++ function), 297
 touch_pad_get_trigger_mode (C++ function), 298
 touch_pad_get_trigger_source (C++ function), 298
 touch_pad_get_voltage (C++ function), 295
 TOUCH_PAD_GPIO0_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO12_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO13_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO14_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO15_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO27_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO2_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO32_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO33_CHANNEL (C macro), 304
 TOUCH_PAD_GPIO4_CHANNEL (C macro), 304
 touch_pad_init (C++ function), 293
 touch_pad_intr_disable (C++ function), 299
 touch_pad_intr_enable (C++ function), 299
 touch_pad_io_init (C++ function), 296
 touch_pad_isr_deregister (C++ function), 294
 touch_pad_isr_handler_register (C++ function), 294
 touch_pad_isr_register (C++ function), 294
 TOUCH_PAD_MAX (C++ class), 302
 TOUCH_PAD_MEASURE_CYCLE_DEFAULT (C macro), 301
 TOUCH_PAD_NUM0 (C++ class), 301
 TOUCH_PAD_NUM0_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM1 (C++ class), 301
 TOUCH_PAD_NUM1_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM2 (C++ class), 301
 TOUCH_PAD_NUM2_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM3 (C++ class), 301
 TOUCH_PAD_NUM3_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM4 (C++ class), 301
 TOUCH_PAD_NUM4_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM5 (C++ class), 301
 TOUCH_PAD_NUM5_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM6 (C++ class), 301
 TOUCH_PAD_NUM6_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM7 (C++ class), 301

TOUCH_PAD_NUM7_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM8 (C++ class), 301
 TOUCH_PAD_NUM8_GPIO_NUM (C macro), 304
 TOUCH_PAD_NUM9 (C++ class), 302
 TOUCH_PAD_NUM9_GPIO_NUM (C macro), 304
 touch_pad_read (C++ function), 293
 touch_pad_read_filtered (C++ function), 293
 touch_pad_set_cnt_mode (C++ function), 296
 touch_pad_set_filter_period (C++ function), 299
 touch_pad_set_fsm_mode (C++ function), 296
 touch_pad_set_group_mask (C++ function), 298
 touch_pad_set_meas_time (C++ function), 294
 touch_pad_set_thresh (C++ function), 297
 touch_pad_set_trigger_mode (C++ function), 297
 touch_pad_set_trigger_source (C++ function), 298
 touch_pad_set_voltage (C++ function), 295
 TOUCH_PAD_SLEEP_CYCLE_DEFAULT (C macro), 301
 TOUCH_PAD_SLOPE_0 (C++ class), 303
 TOUCH_PAD_SLOPE_1 (C++ class), 303
 TOUCH_PAD_SLOPE_2 (C++ class), 303
 TOUCH_PAD_SLOPE_3 (C++ class), 303
 TOUCH_PAD_SLOPE_4 (C++ class), 303
 TOUCH_PAD_SLOPE_5 (C++ class), 303
 TOUCH_PAD_SLOPE_6 (C++ class), 303
 TOUCH_PAD_SLOPE_7 (C++ class), 303
 TOUCH_PAD_SLOPE_MAX (C++ class), 303
 touch_pad_sw_start (C++ function), 297
 touch_pad_t (C++ type), 301
 TOUCH_PAD_TIE_OPT_HIGH (C++ class), 303
 TOUCH_PAD_TIE_OPT_LOW (C++ class), 303
 TOUCH_PAD_TIE_OPT_MAX (C++ class), 303
 touch.tie_opt_t (C++ type), 303
 TOUCH_TRIGGER ABOVE (C++ class), 303
 TOUCH_TRIGGER BELOW (C++ class), 303
 TOUCH_TRIGGER_MAX (C++ class), 303
 TOUCH_TRIGGER_MODE_DEFAULT (C macro), 301
 touch_trigger_mode_t (C++ type), 303
 TOUCH_TRIGGER_SOURCE_BOTH (C++ class), 303
 TOUCH_TRIGGER_SOURCE_DEFAULT (C macro), 301
 TOUCH_TRIGGER_SOURCE_MAX (C++ class), 303
 TOUCH_TRIGGER_SOURCE_SET1 (C++ class), 303
 touch_trigger_src_t (C++ type), 303
 touch_volt_atten_t (C++ type), 302
 transaction_cb_t (C++ type), 277

U

UART_BITRATE_MAX (C macro), 316
 UART_BREAK (C++ class), 318
 UART_BUFFER_FULL (C++ class), 318
 uart_clear_intr_status (C++ function), 308
 uart_config_t (C++ class), 315
 uart_config_t::baud_rate (C++ member), 315
 uart_config_t::data_bits (C++ member), 315
 uart_config_t::flow_ctrl (C++ member), 315
 uart_config_t::parity (C++ member), 315
 uart_config_t::rx_flow_ctrl_thresh (C++ member), 315
 uart_config_t::stop_bits (C++ member), 315
 UART_CTS_GPIO19_DIRECT_CHANNEL (C macro), 319
 UART_CTS_GPIO6_DIRECT_CHANNEL (C macro), 319
 UART_CTS_GPIO8_DIRECT_CHANNEL (C macro), 319
 UART_DATA (C++ class), 317
 UART_DATA_5_BITS (C++ class), 316
 UART_DATA_6_BITS (C++ class), 316
 UART_DATA_7_BITS (C++ class), 316
 UART_DATA_8_BITS (C++ class), 316
 UART_DATA_BITS_MAX (C++ class), 316
 UART_DATA_BREAK (C++ class), 318
 uart_disable_intr_mask (C++ function), 308
 uart_disable_pattern_det_intr (C++ function), 314
 uart_disable_rx_intr (C++ function), 309
 uart_disable_tx_intr (C++ function), 309
 uart_driver_delete (C++ function), 312
 uart_driver_install (C++ function), 311
 uart_enable_intr_mask (C++ function), 308
 uart_enable_pattern_det_intr (C++ function), 314
 uart_enable_rx_intr (C++ function), 308
 uart_enable_tx_intr (C++ function), 309
 UART_EVENT_MAX (C++ class), 318
 uart_event_t (C++ class), 315
 uart_event_t::size (C++ member), 315
 uart_event_t::type (C++ member), 315
 uart.event_type_t (C++ type), 317
 UART_FIFO_LEN (C macro), 316
 UART_FIFO_OVF (C++ class), 318
 uart_flush (C++ function), 313
 UART_FRAME_ERR (C++ class), 318
 uart_get_baudrate (C++ function), 307
 uart_get_buffered_data_len (C++ function), 314
 uart_get_hw_flow_ctrl (C++ function), 307
 uart_get_parity (C++ function), 306
 uart_get_stop_bits (C++ function), 306
 uart_get_word_length (C++ function), 305
 UART_GPIO10_DIRECT_CHANNEL (C macro), 319
 UART_GPIO11_DIRECT_CHANNEL (C macro), 319
 UART_GPIO16_DIRECT_CHANNEL (C macro), 319
 UART_GPIO17_DIRECT_CHANNEL (C macro), 319
 UART_GPIO19_DIRECT_CHANNEL (C macro), 318
 UART_GPIO1_DIRECT_CHANNEL (C macro), 318
 UART_GPIO22_DIRECT_CHANNEL (C macro), 318
 UART_GPIO3_DIRECT_CHANNEL (C macro), 318
 UART_GPIO6_DIRECT_CHANNEL (C macro), 319
 UART_GPIO7_DIRECT_CHANNEL (C macro), 319
 UART_GPIO8_DIRECT_CHANNEL (C macro), 319

UART_GPIO9_DIRECT_CHANNEL (C macro), 319
uart_hw_flowcontrol_t (C++ type), 317
UART_HW_FLOWCTRL_CTS (C++ class), 317
UART_HW_FLOWCTRL_CTS_RTS (C++ class), 317
UART_HW_FLOWCTRL_DISABLE (C++ class), 317
UART_HW_FLOWCTRL_MAX (C++ class), 317
UART_HW_FLOWCTRL_RTS (C++ class), 317
uart_intr_config (C++ function), 311
uart_intr_config_t (C++ class), 315
uart_intr_config_t::intr_enable_mask (C++ member), 315
uart_intr_config_t::rx_timeout_thresh (C++ member), 315
uart_intr_config_t::rxfifo_full_thresh (C++ member), 315
uart_intr_config_t::txfifo_empty_intr_thresh (C++ member), 315
UART_INTR_MASK (C macro), 316
UART_INVERSE_CTS (C macro), 316
UART_INVERSE_DISABLE (C macro), 316
UART_INVERSE_RTS (C macro), 316
UART_INVERSE_RXD (C macro), 316
UART_INVERSE_TXD (C macro), 316
uart_isr_free (C++ function), 310
uart_isr_handle_t (C++ type), 316
uart_isr_register (C++ function), 309
UART_LINE_INV_MASK (C macro), 316
UART_NUM_0 (C++ class), 317
UART_NUM_0_CTS_DIRECT_GPIO_NUM (macro), 318 (C)
UART_NUM_0_RTS_DIRECT_GPIO_NUM (C macro), 318
UART_NUM_0_RXD_DIRECT_GPIO_NUM (macro), 318 (C)
UART_NUM_0_TXD_DIRECT_GPIO_NUM (macro), 318 (C)
UART_NUM_1 (C++ class), 317
UART_NUM_1_CTS_DIRECT_GPIO_NUM (macro), 319 (C)
UART_NUM_1_RTS_DIRECT_GPIO_NUM (C macro), 319
UART_NUM_1_RXD_DIRECT_GPIO_NUM (macro), 319 (C)
UART_NUM_1_TXD_DIRECT_GPIO_NUM (macro), 319 (C)
UART_NUM_2 (C++ class), 317
UART_NUM_2_CTS_DIRECT_GPIO_NUM (macro), 319 (C)
UART_NUM_2_RTS_DIRECT_GPIO_NUM (C macro), 319
UART_NUM_2_RXD_DIRECT_GPIO_NUM (macro), 319 (C)
UART_NUM_2_TXD_DIRECT_GPIO_NUM (macro), 319 (C)
UART_NUM_MAX (C++ class), 317
uart_param_config (C++ function), 311

UART_PARITY_DISABLE (C++ class), 317
UART_PARITY_ERR (C++ class), 318
UART_PARITY EVEN (C++ class), 317
UART_PARITY ODD (C++ class), 317
uart_parity_t (C++ type), 317
UART_PATTERN_DET (C++ class), 318
UART_PIN_NO_CHANGE (C macro), 316
uart_port_t (C++ type), 317
uart_read_bytes (C++ function), 313
UART_RTS_GPIO11_DIRECT_CHANNEL (C macro), 319
UART_RTS_GPIO22_DIRECT_CHANNEL (C macro), 319
UART_RTS_GPIO7_DIRECT_CHANNEL (C macro), 319
UART_RXD_GPIO16_DIRECT_CHANNEL (C macro), 319
UART_RXD_GPIO3_DIRECT_CHANNEL (C macro), 319
UART_RXD_GPIO9_DIRECT_CHANNEL (C macro), 319
uart_set_baudrate (C++ function), 306
uart_set_dtr (C++ function), 310
uart_set_hw_flow_ctrl (C++ function), 307
uart_set_line_inverse (C++ function), 307
uart_setparity (C++ function), 306
uart_set_pin (C++ function), 310
uart_set_rts (C++ function), 310
uart_set_stop_bits (C++ function), 305
uart_set_word_length (C++ function), 305
UART_STOP_BITS_1 (C++ class), 317
UART_STOP_BITS_1_5 (C++ class), 317
UART_STOP_BITS_2 (C++ class), 317
UART_STOP_BITS_MAX (C++ class), 317
uart_stop_bits_t (C++ type), 316
uart_tx_chars (C++ function), 312
UART_RXD_GPIO10_DIRECT_CHANNEL (C macro), 319
UART_RXD_GPIO17_DIRECT_CHANNEL (C macro), 319
UART_RXD_GPIO1_DIRECT_CHANNEL (C macro), 319
uart_wait_tx_done (C++ function), 312
uart_word_length_t (C++ type), 316
uart_write_bytes (C++ function), 312
uart_write_bytes_with_break (C++ function), 313
ulp_load_binary (C++ function), 598
ulp_process_macros_and_load (C++ function), 592
ulp_run (C++ function), 592, 599
ulp_set_wakeup_period (C++ function), 600

V

vprintf_like_t (C++ type), 410
VSPI_HOST (C++ class), 272

W

WIFI_AMPDU_ENABLED (C macro), [64](#)
WIFI_DEFAULT_RX_BA_WIN (C macro), [64](#)
WIFI_DEFAULT_TX_BA_WIN (C macro), [64](#)
WIFI_DYNAMIC_TX_BUFFER_NUM (C macro), [64](#)
WIFI_INIT_CONFIG_DEFAULT (C macro), [64](#)
WIFI_INIT_CONFIG_MAGIC (C macro), [64](#)
wifi_init_config_t (C++ class), [62](#)
wifi_init_config_t::ampdu_enable (C++ member), [63](#)
wifi_init_config_t::dynamic_rx_buf_num (C++ member),
[62](#)
wifi_init_config_t::dynamic_tx_buf_num (C++ member),
[63](#)
wifi_init_config_t::event_handler (C++ member), [62](#)
wifi_init_config_t::magic (C++ member), [63](#)
wifi_init_config_t::nano_enable (C++ member), [63](#)
wifi_init_config_t::nvs_enable (C++ member), [63](#)
wifi_init_config_t::rx_ba_win (C++ member), [63](#)
wifi_init_config_t::static_rx_buf_num (C++ member), [62](#)
wifi_init_config_t::static_tx_buf_num (C++ member), [62](#)
wifi_init_config_t::tx_ba_win (C++ member), [63](#)
wifi_init_config_t::tx_buf_type (C++ member), [62](#)
wifi_init_config_t::wpa_crypto_funcs (C++ member), [62](#)
WIFI_NANO_FORMAT_ENABLED (C macro), [64](#)
WIFI_NVFS_ENABLED (C macro), [64](#)
wifi_promiscuous_cb_t (C++ type), [64](#)
WIFI_STATIC_TX_BUFFER_NUM (C macro), [64](#)
wl_erase_range (C++ function), [368](#)
wl_handle_t (C++ type), [370](#)
WL_INVALID_HANDLE (C macro), [370](#)
wl_mount (C++ function), [368](#)
wl_read (C++ function), [369](#)
wl_sector_size (C++ function), [369](#)
wl_size (C++ function), [369](#)
wl_unmount (C++ function), [368](#)
wl_write (C++ function), [368](#)