ESP8266 RTOS SDK User Manual

Sep 03, 2021

Contents

This is the documentation for the new ESP8266_RTOS_SDK which refactored to be ESP-IDF Style. ESP8266_RTOS_SDK is the official development framework for the ESP8266EX chip.

Get Started	API Reference	API Guides	General Notes

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 ESP8266EX. Through a simple example we would like to illustrate how to use ESP8266_RTOS_SDK (ESP-IDF Style), including the menu based configuration, compiling the ESP8266_RTOS_SDK and firmware download to ESP8266EX boards.

1.1 Introduction

The ESP8266EX microcontroller integrates a Tensilica L106 32-bit RISC processor, which achieves extra-low power consumption and reaches a maximum clock speed of 160 MHz. The Real-Time Operating System (RTOS) and Wi-Fi stack allow about 80% of the processing power to be available for user application programming and development.

Espressif provides the basic hardware and software resources that help application developers to build their ideas around the ESP8266EX series hardware. The software development framework by Espressif is intended for rapidly developing Internet-of-Things (IoT) applications, with Wi-Fi, power management and several other system features.

1.2 What You Need

To develop applications for ESP8266EX you need:

- **PC** loaded with either Windows, Linux or Mac operating system
- **Toolchain** to build the **Application** for ESP8266EX
- **ESP8266_RTOS_SDK** that essentially contains API for ESP8266EX and scripts to operate the **Toolchain**
- A text editor to write programs (**Projects**) in C, e.g. Eclipse
- The **ESP8266EX** board itself and a **USB cable** to connect it to the **PC**

Preparation of development environment consists of three steps:

1. Setup of **Toolchain**

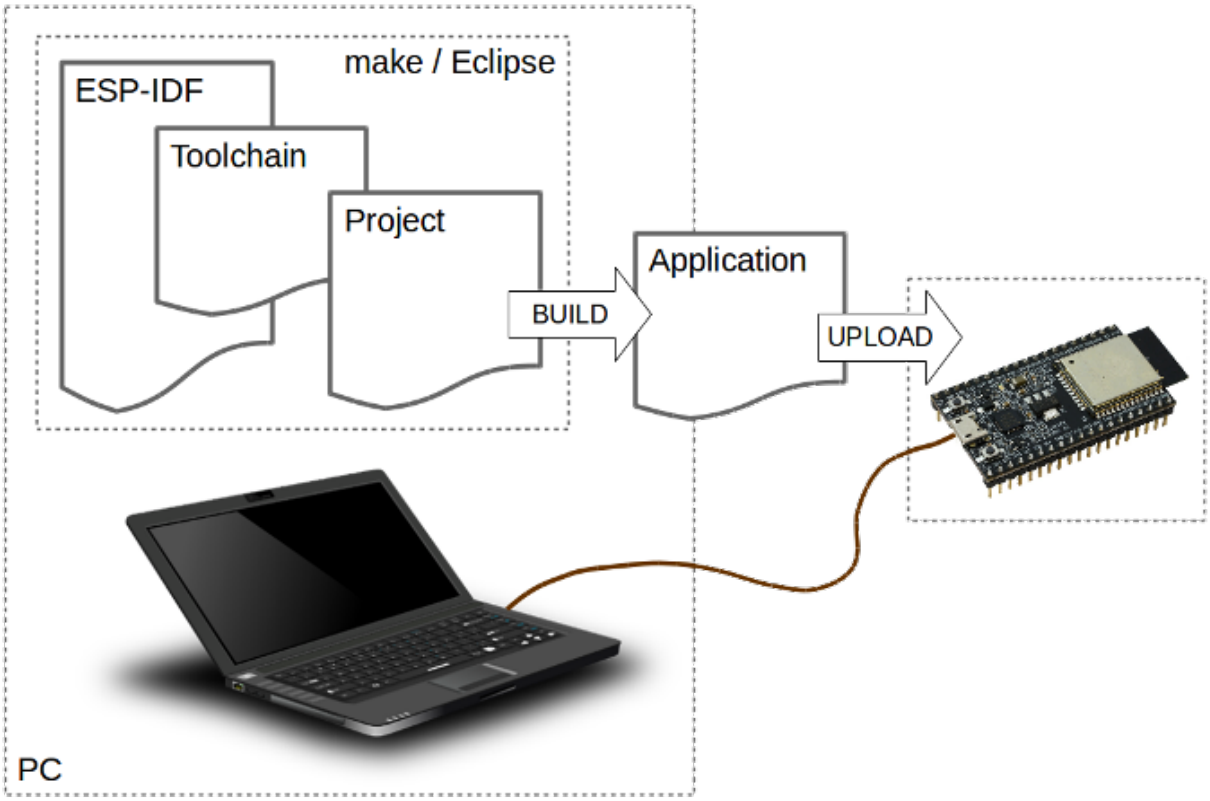


Fig. 1: Development of applications for ESP8266EX

2. Getting of **ESP8266_RTOS_SDK** 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 **ESP8266EX**
4. Monitoring / debugging of the **Application**

See instructions below that will walk you through these steps.

1.3 Guides

If you have one of ESP8266 development boards listed below, click on provided links to get you up and running.

1.3.1 ESP8266-DevKitC Getting Started Guide

This user guide shows how to get started with ESP8266-DevKitC development board.

What You Need

- 1 × ESP8266-DevKitC board
- 1 × USB A / micro USB B cable
- 1 × PC loaded with Windows, Linux or Mac OS

Overview

ESP8266-DevKitC is a small-sized ESP8266-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 ESP8266-DevKitC board.

ESP-WROOM-02D/U Module soldered to the ESP8266-DevKitC board. Optionally ESP-WROOM-02D or ESP-WROOM-02U module may be soldered.

5V to 3.3V LDO A LDO regulator with a maximum current output of 800 mA, which provides power supply for ESP8266 module and user's peripherals.

Dial Switch A dial switch used for switching between Auto Download and Flow Control.

- Bit1=OFF, Bit2=ON (Auto Download)
- Bit1=ON, Bit2=OFF (Flow Control)

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

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

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

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

I/O Connector All of the pins on the ESP8266 module are broken out to the pin headers on the board. Users can program ESP8266 to enable multiple functions. For details, please refer to ESP8266EX Datasheet .

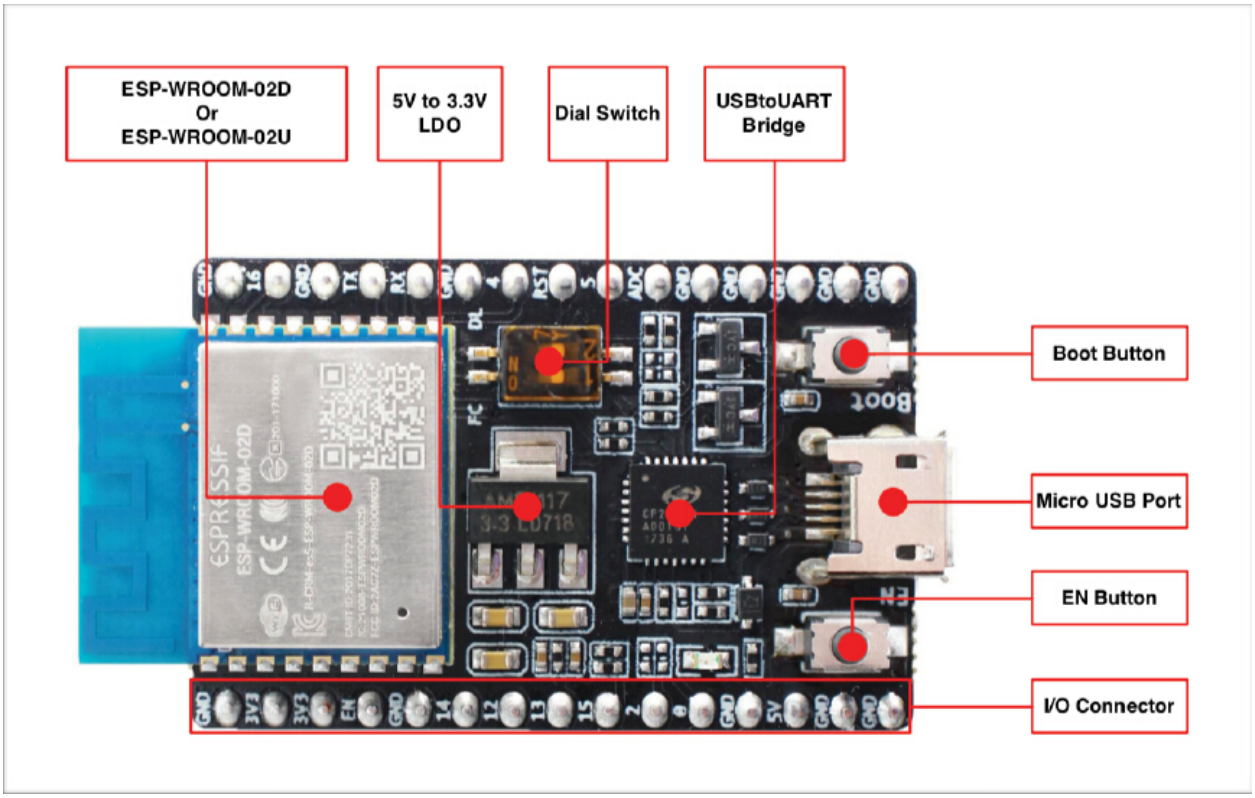


Fig. 2: ESP8266-DevKitC with ESP-WROOM-02D module soldered

Power Supply Options

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

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

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

Start Application Development

Before powering up the ESP8266-DevKitC, please make sure that the board has been received in good condition with no obvious signs of damage.

To start development of applications, you may walk through the following steps:

- setup toolchain in your PC to develop applications for ESP8266 in C language
- connect the module to the PC and verify if it is accessible
- build an example application to the ESP8266
- monitor instantly what the application is doing

Board Dimensions

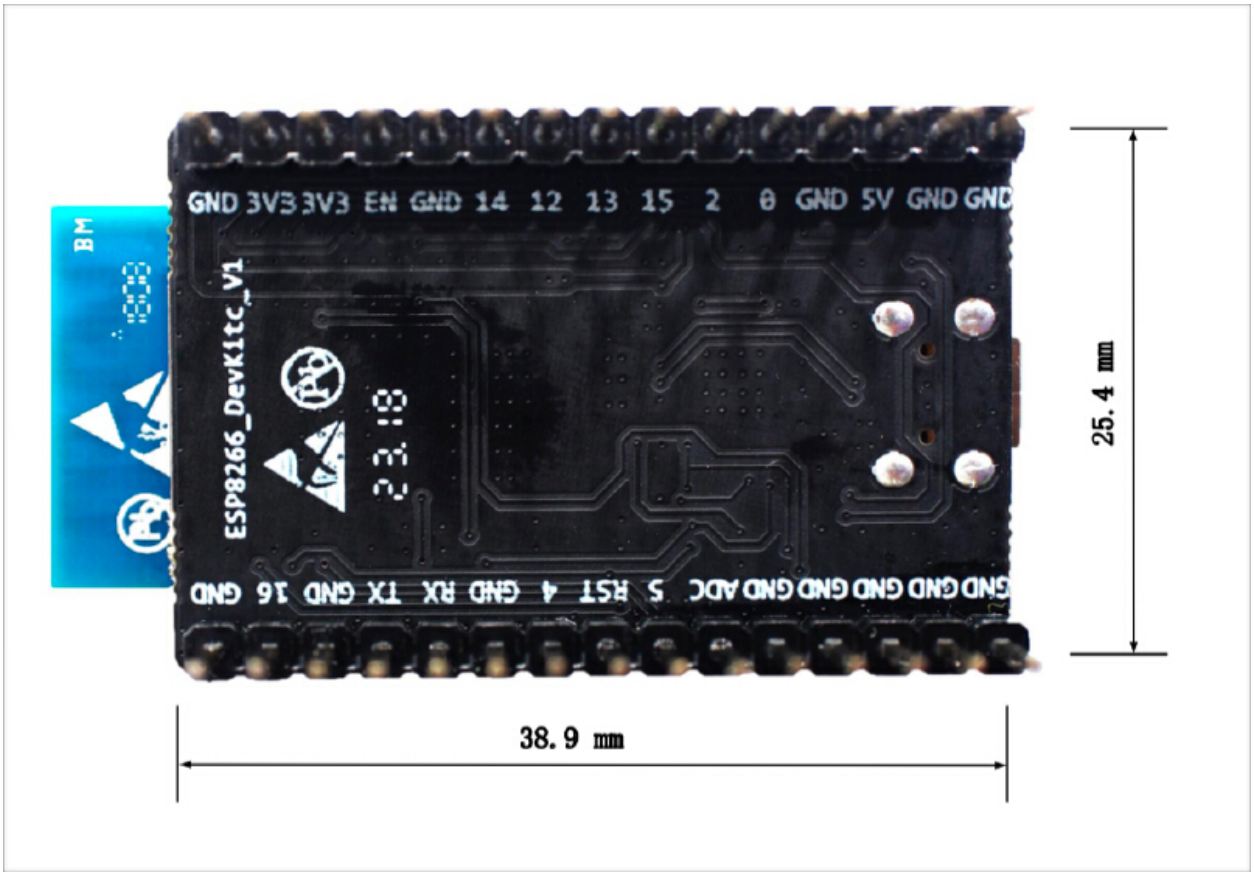


Fig. 3: ESP8266 DevKitC board dimensions - back

Related Documents

- ESP-WROOM-02 PCB Design and Module Placement Guide (WEB)
- ESP8266 Hardware Resources (WEB)
- ESP8266 App (WEB)

- ESP8266 BBS (WEB)
- ESP8266 Resources (WEB)

If you have different board, move to sections below.

1.4 Setup Toolchain

The quickest way to start development with ESP8266EX is by installing a prebuilt toolchain. Pick up your OS below and follow provided instructions.

1.4.1 Standard Setup of Toolchain for Windows

Introduction

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

Toolchain Setup

The quick setup is to download the Windows all-in-one toolchain & MSYS2 zip file from dl.espressif.com:

https://dl.espressif.com/dl/esp32_win32_msys2_environment_and_toolchain-20181001.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.

Download the toolchain for the ESP8266

v8.4.0

https://dl.espressif.com/dl/xtensa-lx106-elf-gcc8_4_0-esp-2020r3-win32.zip

If you are still using old version SDK(< 3.0), please use toolchain v4.8.5, as following:

https://dl.espressif.com/dl/xtensa-lx106-elf-win32-1.22.0-88-gde0bdc1-4.8.5.tar.gz

Check it Out

Open a MSYS2 MINGW32 terminal window by running `C:\msys32\mingw32.exe`. The environment in this window is a bash shell. Create a directory named `esp` that is a default location to develop ESP8266 applications. To do so, run the following shell command:

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

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

Use this window in the following steps setting up development environment for ESP8266.

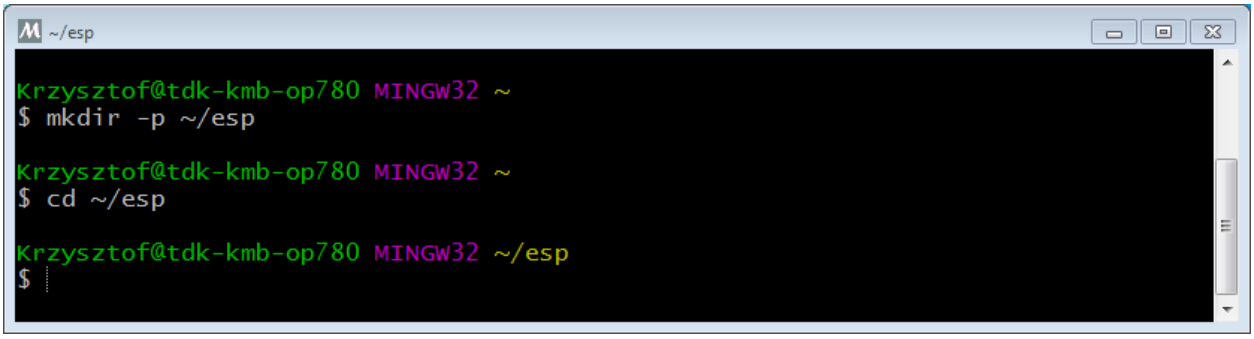


Fig. 4: MSYS2 MINGW32 shell window

Next Steps

To carry on with development environment setup, proceed to section *Get ESP8266_RTOS_SDK*.

Related Documents

1.4.2 Standard Setup of Toolchain for Linux

Install Prerequisites

To compile with ESP8266_RTOS_SDK you need to get the following packages:

- CentOS 7:

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

- Ubuntu and Debian:

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

- Arch:

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

Toolchain Setup

ESP8266 toolchain for Linux is available for download from Espressif website:

- for 64-bit Linux:

  https://dl.espressif.com/dl/xtensa-lx106-elf-gcc8_4_0-esp-2020r3-linux-amd64.tar.gz

- for 32-bit Linux:

  https://dl.espressif.com/dl/xtensa-lx106-elf-gcc8_4_0-esp-2020r3-linux-i686.tar.gz

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

```
mkdir -p ~/esp
cd ~/esp
tar -xzf ~/Downloads/xtensa-lx106-elf-linux64-1.22.0-100-ge567ec7-5.2.0.tar.gz
```

2. The toolchain will be extracted into `~/esp/xtensa-lx106-elf/` directory.

   To use it, you will need to update your `PATH` environment variable in `~/.profile` file. To make `xtensa-lx106-elf` available for all terminal sessions, add the following line to your `~/.profile` file:

   ```
   export PATH="$PATH:$HOME/esp/xtensa-lx106-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_lx106='export PATH="$PATH:$HOME/esp/xtensa-lx106-elf/bin"'
   ```

   Then when you need the toolchain you can type `get_lx106` 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/xtense-lx106-elf/bin
   ```

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

Permission issues /dev/ttyUSB0

With some Linux distributions you may get the `Failed to open port /dev/ttyUSB0` error message when flashing the ESP8266.

If this happens you may need to add your current user to the correct group (commonly "dialout") which has the appropriate permissions:

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

In addition, you can also use "sudo chmod" to set permissions on the "/dev/ttyUSB0" file before running the make command to resolve:

```
sudo chmod -R 777 /dev/ttyUSB0
```

Next Steps

To carry on with development environment setup, proceed to section *Get ESP8266_RTOS_SDK*.

Related Documents

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

ESP8266 toolchain for macOS is available for download from Espressif website:

https://dl.espressif.com/dl/xtensa-lx106-elf-gcc8_4_0-esp-2020r3-macos.tar.gz

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

```
mkdir -p ~/esp
cd ~/esp
tar -xzf ~/Downloads/xtensa-lx106-elf-macos-1.22.0-100-ge567ec7-5.2.0.tar.gz
```

The toolchain will be extracted into `~/esp/xtensa-lx106-elf/` directory.

To use it, you will need to update your `PATH` environment variable in `~/.profile` file. To make `xtensa-lx106-elf` available for all terminal sessions, add the following line to your `~/.profile` file:

```
export PATH=$PATH:$HOME/esp/xtensa-lx106-elf/bin
```

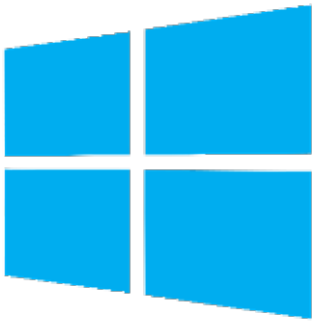
Alternatively, you may create an alias for the above command. This way you can get the toolchain only when you need it. To do this, add different line to your `~/.profile` file:

```
alias get_lx106="export PATH=$PATH:$HOME/esp/xtensa-lx106-elf/bin"
```

Then when you need the toolchain you can type `get_lx106` 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 ESP8266_RTOS_SDK*.

		
Windows	Linux	Mac OS

Note: We are using `~/esp` directory to install the prebuilt toolchain, ESP8266_RTOS_SDK 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..

Once you are done with setting up the toolchain then go to section *Get ESP8266_RTOS_SDK*.

1.5 Get ESP8266_RTOS_SDK

Besides the toolchain (that contains programs to compile and build the application), you also need ESP8266 specific API / libraries. They are provided by Espressif in ESP8266_RTOS_SDK repository.

To obtain a local copy: open terminal, navigate to the directory you want to put ESP8266_RTOS_SDK, and clone the repository using `git clone` command:

```
cd ~/esp
git clone --recursive https://github.com/espressif/ESP8266_RTOS_SDK.git
```

ESP8266_RTOS_SDK will be downloaded into `~/esp/ESP8266_RTOS_SDK`.

Note: This command will clone the master branch, which has the latest development ("bleeding edge") version of ESP8266_RTOS_SDK. It is fully functional and updated on weekly basis with the most recent features and bugfixes.

Note: GitHub's "Download zip file" feature does not work with ESP8266_RTOS_SDK, a `git clone` is required. As a fallback, Stable version can be installed without Git.

1.6 Setup Path to ESP8266_RTOS_SDK

The toolchain programs access ESP8266_RTOS_SDK 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.

1.7 Install the Required Python Packages

Python packages required by ESP8266_RTOS_SDK are located in the `$IDF_PATH/requirements.txt` file. You can install them by running:

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

Note: Please invoke that version of the Python interpreter which you will be using with ESP8266_RTOS_SDK. The version of the interpreter can be checked by running command `python --version` and depending on the result, you might want to use `python2`, `python2.7` or similar instead of `python`, e.g.:

```
python2.7 -m pip install --user -r $IDF_PATH/requirements.txt
```

1.8 Start a Project

Now you are ready to prepare your application for ESP8266. 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 ESP8266_RTOS_SDK build system does not support spaces in paths to ESP8266_RTOS_SDK or to projects.

1.9 Connect

You are almost there. To be able to proceed further, connect ESP8266 board to PC, check under what serial port the board is visible and verify if serial communication works. Note the port number, as it will be required in the next step.

1.10 Configure

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

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

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

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

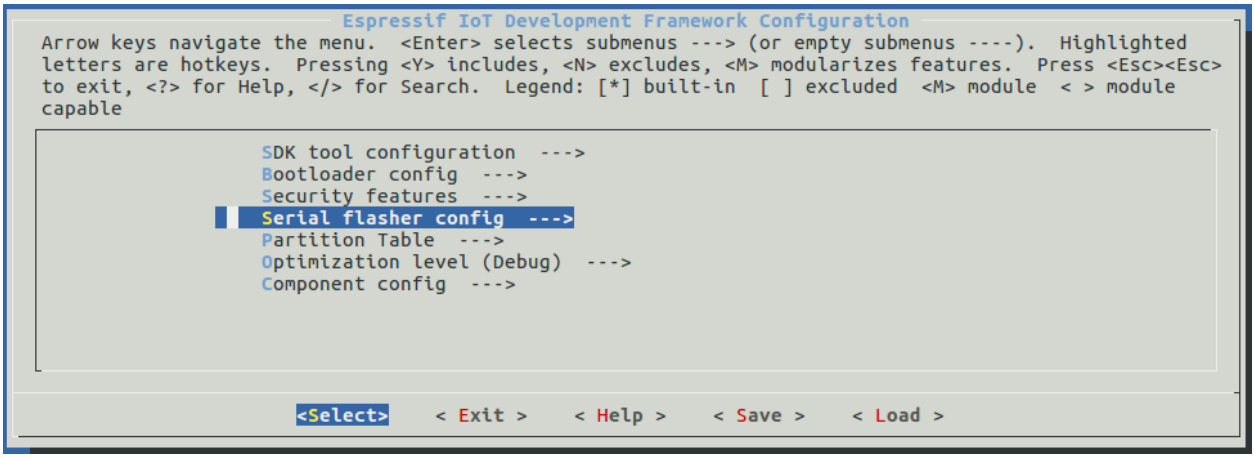


Fig. 5: Project configuration - Home window

Note: On Windows, serial ports have names like COM1. On MacOS, they start with `/dev/cu.`. On Linux, they start with `/dev/tty`.

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

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

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

1.11 Build and Flash

Now you can build and flash the application. Run:

```
make flash
```

This will compile the application and all the ESP8266_RTOS_SDK components, generate bootloader, partition table, and application binaries, and flash these binaries to your ESP8266 board.

```
esptool.py v2.4.0
Flashing binaries to serial port /dev/ttyUSB0 (app at offset 0x10000)...
esptool.py v2.4.0
Connecting....
Chip is ESP8266EX
Features: WiFi
```

(continues on next page)

(continued from previous page)

```
MAC: ec:fa:bc:1d:33:2d
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Compressed 7952 bytes to 5488...
Wrote 7952 bytes (5488 compressed) at 0x00000000 in 0.5 seconds (effective 129.9 kbit/
↪s)...
Hash of data verified.
Compressed 234800 bytes to 162889...
Wrote 234800 bytes (162889 compressed) at 0x00010000 in 14.4 seconds (effective 130.6␣
↪kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 83...
Wrote 3072 bytes (83 compressed) at 0x00008000 in 0.0 seconds (effective 1789.8 kbit/
↪s)...
Hash of data verified.

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

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*.

1.12 Monitor

To see if "hello_world" application is indeed running, type `make monitor`.

> $ make monitor MONITOR — idf_monitor on /dev/ttyUSB0 74880 — — Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H —
>
> ets Jan 8 2013,rst cause:1, boot mode:(3,6)
>
> load 0x40100000, len 4400, room 16 0x40100000: _stext at ??:?
>
> tail 0 chksum 0x6f load 0x3ffe8408, len 3516, room 8 tail 4 chksum 0x5d . . .

Several lines below, after start up and diagnostic log, you should see "SDK version: xxxxxxx" printed out by the application.

```
...
SDK version:v3.1-dev-311-g824cd8c8-dirty
```

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

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

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

To execute `make flash` and `make monitor` in one go, type `make flash monitor`.

That's all what you need to get started with ESP8266!

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

1.13 Environment Variables

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

Variables	Description & Usage
`ESPPORT`	Overrides the serial port used in `flash` and `monitor`. Examples: `make flash ESPPORT=/dev/ttyUSB1`, `make monitor ESPPORT=COM1`
`ESPBAUD`	Overrides the serial baud rate when flashing the ESP32. Example: `make flash ESPBAUD=9600`
`MONITORBAUD`	Overrides the serial baud rate used when monitoring. Example: `make monitor MONITORBAUD=9600`

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

1.14 Related Documents

1.14.1 Build and Flash with Eclipse IDE

Installing Eclipse IDE

The Eclipse IDE gives you a graphical integrated development environment for writing, compiling and debugging ESP8266_RTOS_SDK projects.

- Start by installing the ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK. This means you need to start by creating an ESP8266_RTOS_SDK project. You can use the idf-template project from github, or open one of the examples in the ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK project. Don't specify the path to the ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK is installed.
- Edit the `PATH` environment variable. Keep the current value, and append the path to the Xtensa toolchain installed as part of ESP8266_RTOS_SDK setup, if this is not already listed on the PATH. A typical path to the toolchain looks like `/home/user-name/esp/xtensa-lx106-elf/bin`. Note that you need to add a colon `:` before the appended path.
- On macOS, add a `PYTHONPATH` environment variable and set it to `/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages`. This is so that the system Python, which has pyserial installed as part of the setup steps, overrides any built-in Eclipse Python.

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

- Click the "Providers" tab
- In the list of providers, click "CDT Cross GCC Built-in Compiler Settings". Change "Command to get compiler specs" to `xtensa-lx106-elf-gcc ${FLAGS} -E -P -v -dD "${INPUTS}"`.
- In the list of providers, click "CDT GCC Build Output Parser" and change the "Compiler command pattern" to `xtensa-lx106-elf-(gcc|g\+\+|c\+\+|cc|cpp|clang)`

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

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

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

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

Building in Eclipse

Before your project is first built, Eclipse may show a lot of errors and warnings about undefined values. This is because some source files are automatically generated as part of the ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK settings. This step currently has to be run outside Eclipse.

If you try to build without running a configuration step first, ESP8266_RTOS_SDKf will prompt for configuration on the command line - but Eclipse is not able to deal with this, so the build will hang or fail.

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

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

Flash from Eclipse

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

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

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

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

Related Documents

Eclipse IDE on Windows

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

(For OS X and Linux instructions, see the *Eclipse IDE page*.)

Installing Eclipse IDE

Follow the steps under *Installing Eclipse IDE* for all platforms.

Setting up Eclipse on Windows

Once your new Eclipse installation launches, follow these steps:

Import New Project

- Eclipse makes use of the Makefile support in ESP8266_RTOS_SDK. This means you need to start by creating an ESP8266_RTOS_SDK project. You can use the idf-template project from github, or open one of the examples in the ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK project. Don't specify the path to the ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK is installed. The IDF_PATH directory should be specified using forwards slashes not backslashes, ie *C:/Users/user-name/Development/ESP8266_RTOS_SDK*.
  - 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-lx106-elf\bin` (If you installed msys32 to a different directory then you'll need to change these paths to match).
- Click on "C/C++ General" -> "Preprocessor Include Paths, Macros, etc." property page:
  - Click the "Providers" tab
    - In the list of providers, click "CDT Cross GCC Built-in Compiler Settings". Change "Command to get compiler specs" to `xtensa-lx106-elf-gcc ${FLAGS} -E -P -v -dD "${INPUTS}"`.
    - In the list of providers, click "CDT GCC Build Output Parser" and change the "Compiler command pattern" to `xtensa-lx106-elf-(gcc|g\+\+|c\+\+|cc|cpp|clang)`

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

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

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

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

Building in Eclipse

Continue from *Building in Eclipse* for all platforms.

Technical Details

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

Explanations of the technical reasons for some of these steps. You don't need to know this to use ESP8266_RTOS_SDK with Eclipse on Windows, but it may be helpful background knowledge if you plan to do dig into the Eclipse support:

- The xtensa-lx106-elf-gcc cross-compiler is *not* a Cygwin toolchain, even though we tell Eclipse that it is one. This is because msys2 uses Cygwin and supports Unix-style paths (of the type `/c/blah` instead of `c:/blah` or `c:\\blah`). In particular, xtensa-lx106-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 ESP8266_RTOS_SDK. 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 ESP8266_RTOS_SDK 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.

CHAPTER 2

API Reference

2.1 Peripherals API

2.1.1 GPIO

API Reference

Header File

- esp8266/include/driver/gpio.h

Functions

esp_err_t **gpio_config**(**const** *gpio_config_t* *gpio_cfg)
GPIO common configuration.

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

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_cfg`: 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 GPIO12, gpio_num should be GPIO_NUM_12 (12);
- `intr_type`: Interrupt type, select from gpio_int_type_t

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.

Note If the pad is not configured for input (or input and output) the returned value is always 0.

Return

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

Parameters

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

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

GPIO set direction.

Configure GPIO direction,such as output_only,input_only

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.

Note The GPIO of esp8266 can not be pulled down except RTC GPIO which can not be pulled up.

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.

Note RTC IO can not use the wakeup 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.

Note RTC IO can not use the wakeup 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 *no_use*, *gpio_isr_handle_t* **handle_no_use*Register GPIO interrupt handler, the handler is an ISR.

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.

Return

- ESP_OK Success ;
- ESP_ERR_INVALID_ARG GPIO error
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags

Parameters

- `fn`: Interrupt handler function.
- `no_use`: In order to be compatible with esp32, the parameter has no practical meaning and can be filled with 0.
- `arg`: Parameter for handler function
- `handle_no_use`: Pointer to return handle. In order to be compatible with esp32,the parameter has no practical meaning and can be filled with NULL.

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

Note The GPIO of esp8266 can not be pulled down except RTC GPIO which can not be pulled up.

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.

Note The GPIO of esp8266 can not be pulled down except RTC GPIO which can not be pulled up.

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.

Note The GPIO of esp8266 can not be pulled down except RTC GPIO which can not be pulled up.

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.

Note The GPIO of esp8266 can not be pulled down except RTC GPIO which can not be pulled up.

Return

- ESP_OK Success

- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number

esp_err_t **gpio_install_isr_service**(int *no_use*)
Install the driver's GPIO ISR handler service, which allows per-pin GPIO interrupt handlers.

This function is incompatible with gpio_isr_register() - if that function is used, a single global ISR is registered for all GPIO interrupts. If this function is used, the ISR service provides a global GPIO ISR and individual pin handlers are registered via the gpio_isr_handler_add() function.

Return

- ESP_OK Success
- ESP_ERR_NO_MEM No memory to install this service
- ESP_ERR_INVALID_STATE ISR service already installed.
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags
- ESP_ERR_INVALID_ARG GPIO error

Parameters

- `no_use`: In order to be compatible with esp32, the parameter has no practical meaning and can be filled with 0.

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.

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

Structures

struct gpio_config_t

Configuration parameters of GPIO pad for gpio_config function.

Public Members

uint32_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

BIT(x)

GPIO_Pin_0

GPIO_Pin_1

GPIO_Pin_2

GPIO_Pin_3

GPIO_Pin_4

GPIO_Pin_5

GPIO_Pin_6

GPIO_Pin_7

GPIO_Pin_8

GPIO_Pin_9

GPIO_Pin_10

GPIO_Pin_11

GPIO_Pin_12

GPIO_Pin_13

GPIO_Pin_14

GPIO_Pin_15

GPIO_Pin_16

GPIO_Pin_All

GPIO_MODE_DEF_DISABLE

GPIO_MODE_DEF_INPUT

GPIO_MODE_DEF_OUTPUT

GPIO_MODE_DEF_OD

GPIO_PIN_COUNT

GPIO_IS_VALID_GPIO(gpio_num)
Check whether it is a valid GPIO number

RTC_GPIO_IS_VALID_GPIO(gpio_num)
Check whether it is a valid RTC GPIO number

Type Definitions

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

typedef void ***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

GPIO_NUM_3 = 3
GPIO3, input and output

GPIO_NUM_4 = 4
GPIO4, input and output

GPIO_NUM_5 = 5
GPIO5, input and output

GPIO_NUM_6 = 6
GPIO6, input and output

GPIO_NUM_7 = 7
GPIO7, input and output

GPIO_NUM_8 = 8
GPIO8, input and output

GPIO_NUM_9 = 9
GPIO9, input and output

GPIO_NUM_10 = 10
GPIO10, input and output

GPIO_NUM_11 = 11
GPIO11, input and output

GPIO_NUM_12 = 12
GPIO12, input and output

GPIO_NUM_13 = 13
GPIO13, input and output

GPIO_NUM_14 = 14
GPIO14, input and output

GPIO_NUM_15 = 15
GPIO15, input and output

GPIO_NUM_16 = 16
GPIO16, input and output

GPIO_NUM_MAX = 17

enum gpio_int_type_t
Values:

GPIO_INTR_DISABLE = 0
Disable GPIO interrupt

GPIO_INTR_POSEDGE = 1
GPIO interrupt type : rising edge

GPIO_INTR_NEGEDGE = 2
GPIO interrupt type : falling edge

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

GPIO_INTR_LOW_LEVEL = 4
GPIO interrupt type : input low level trigger

GPIO_INTR_HIGH_LEVEL = 5
GPIO interrupt type : input high level trigger

GPIO_INTR_MAX

enum gpio_mode_t
Values:

GPIO_MODE_DISABLE = GPIO_MODE_DEF_DISABLE
GPIO mode : disable input and output

GPIO_MODE_INPUT = GPIO_MODE_DEF_INPUT
GPIO mode : input only

GPIO_MODE_OUTPUT = GPIO_MODE_DEF_OUTPUT
GPIO mode : output only mode

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

enum gpio_pull_mode_t
Values:

GPIO_PULLUP_ONLY
Pad pull up

GPIO_PULLDOWN_ONLY
Pad pull down

GPIO_FLOATING
Pad floating

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

2.1.2 I2C

API Reference

Header File

- esp8266/include/driver/i2c.h

Functions

esp_err_t **i2c_driver_install**(*i2c_port_t i2c_num*, *i2c_mode_t mode*)
I2C driver install.

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)

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.

Note It must be used after calling i2c_driver_install

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_set_pin**(*i2c_port_t i2c_num*, int *sda_io_num*, int *scl_io_num*, *gpio_pullup_t sda_pullup_en*, *gpio_pullup_t scl_pullup_en*, *i2c_mode_t mode*)
Configure GPIO signal for I2C sck and sda.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2c_num`: I2C port number
- `sda_io_num`: GPIO number for I2C sda signal
- `scl_io_num`: GPIO number for I2C scl signal
- `sda_pullup_en`: Whether to enable the internal pullup for sda pin
- `scl_pullup_en`: Whether to enable the internal pullup for scl pin
- `mode`: I2C mode

i2c_cmd_handle_t **i2c_cmd_link_create**()
Create and init I2C command link.

Note Before we build I2C command link, we need to call i2c_cmd_link_create() to create a command link. After we finish sending the commands, we need to call i2c_cmd_link_delete() to release and return the resources.

Return i2c command link handler

void **i2c_cmd_link_delete**(*i2c_cmd_handle_t cmd_handle*)
Free I2C command link.

Note Before we build I2C command link, we need to call i2c_cmd_link_create() to create a command link. After we finish sending the commands, we need to call i2c_cmd_link_delete() to release and return the resources.

Parameters

- `cmd_handle`: I2C command handle

esp_err_t **i2c_master_start**(*i2c_cmd_handle_t cmd_handle*)

Queue command for I2C master to generate a start signal.

Note Only call this function in I2C master mode Call i2c_master_cmd_begin() to send all queued commands

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `cmd_handle`: I2C cmd link

esp_err_t **i2c_master_write_byte**(*i2c_cmd_handle_t cmd_handle*, uint8_t *data*, bool *ack_en*)

Queue command for I2C master to write one byte to I2C bus.

Note Only call this function in I2C master mode Call i2c_master_cmd_begin() to send all queued commands

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `cmd_handle`: I2C cmd link
- `data`: I2C one byte command to write to bus
- `ack_en`: enable ack check for master

esp_err_t **i2c_master_write**(*i2c_cmd_handle_t cmd_handle*, uint8_t **data*, size_t *data_len*, bool *ack_en*)

Queue command for I2C master to write buffer to I2C bus.

Note Only call this function in I2C master mode Call i2c_master_cmd_begin() to send all queued commands

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*, *i2c_ack_type_t ack*)
Queue command for I2C master to read one byte from I2C bus.

Note Only call this function in I2C master mode Call i2c_master_cmd_begin() to send all queued commands

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*, *i2c_ack_type_t ack*)
Queue command for I2C master to read data from I2C bus.

Note Only call this function in I2C master mode Call i2c_master_cmd_begin() to send all queued commands

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*, TickType_t *ticks_to_wait*)
I2C master send queued commands. This function will trigger sending all queued commands. The task will be blocked until all the commands have been sent out. The I2C APIs are not thread-safe, if you want to use one I2C port in different tasks, you need to take care of the multi-thread issue.

Note Only call this function in I2C master mode

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

Parameters

- `i2c_num`: I2C port number
- `cmd_handle`: I2C command handler
- `ticks_to_wait`: maximum wait ticks.

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_stretch_tick**
Clock Stretch time, depending on CPU frequency

Type Definitions

typedef void ***i2c_cmd_handle_t**
I2C command handle

Enumerations

enum i2c_mode_t
Values:

I2C_MODE_MASTER
I2C master mode

I2C_MODE_MAX

enum i2c_rw_t
Values:

I2C_MASTER_WRITE = 0
I2C write data

I2C_MASTER_READ
I2C read data

enum i2c_opmode_t
Values:

I2C_CMD_RESTART = 0
I2C restart command

I2C_CMD_WRITE
I2C write command

I2C_CMD_READ
I2C read command

I2C_CMD_STOP
I2C stop command

enum i2c_port_t
Values:

I2C_NUM_0 = 0
I2C port 0

I2C_NUM_MAX

enum i2c_ack_type_t
Values:

I2C_MASTER_ACK = 0x0
I2C ack for each byte read

I2C_MASTER_NACK = 0x1
I2C nack for each byte read

I2C_MASTER_LAST_NACK = 0x2
I2C nack for the last byte

I2C_MASTER_ACK_MAX

2.1.3 I2S

API Reference

Header File

- esp8266/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.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL IO error

Parameters

- `i2s_num`: I2S_NUM_0
- `pin`: I2S Pin structure

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.

Note This function must be called before any I2S driver read/write operations.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

Parameters

- `i2s_num`: I2S_NUM_0
- `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.

esp_err_t **i2s_driver_uninstall**(*i2s_port_t i2s_num*)

Uninstall I2S driver.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2s_num`: I2S_NUM_0

esp_err_t **i2s_write**(*i2s_port_t i2s_num*, **const** void **src*, size_t *size*, size_t **bytes_written*, TickType_t *ticks_to_wait*)

Write data to I2S DMA transmit buffer.

Note many ticks pass without space becoming available in the DMA transmit buffer, then the function will return (note that if the data is written to the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass portMAX_DELAY for no timeout.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- i2s_num: I2S_NUM_0
- src: Source address to write from
- size: Size of data in bytes
- bytes_written: Number of bytes written, if timeout, the result will be less than the size passed in.
- ticks_to_wait: TX buffer wait timeout in RTOS ticks. If this

esp_err_t **i2s_write_expand**(*i2s_port_t i2s_num*, **const** void **src*, size_t *size*, size_t *src_bits*, size_t *aim_bits*, size_t **bytes_written*, TickType_t *ticks_to_wait*)

Write data to I2S DMA transmit buffer while expanding the number of bits per sample. For example, expanding 16-bit PCM to 32-bit PCM.

Note 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. Format of the data in source buffer is determined by the I2S configuration (see *i2s_config_t*).

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- i2s_num: I2S_NUM_0
- src: Source address to write from
- size: Size of data in bytes
- src_bits: Source audio bit
- aim_bits: Bit wanted, no more than 32, and must be greater than src_bits
- bytes_written: Number of bytes written, if timeout, the result will be less than the size passed in.
- ticks_to_wait: TX buffer wait timeout in RTOS ticks. If this

esp_err_t **i2s_read**(*i2s_port_t i2s_num*, void **dest*, size_t *size*, size_t **bytes_read*, TickType_t *ticks_to_wait*)

Read data from I2S DMA receive buffer.

Note If the built-in ADC mode is enabled, we should call i2s_adc_start and i2s_adc_stop around the whole reading process, to prevent the data getting corrupted.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- i2s_num: I2S_NUM_0
- dest: Destination address to read into
- size: Size of data in bytes
- bytes_read: Number of bytes read, if timeout, bytes read will be less than the size passed in.

- `ticks_to_wait`: RX buffer wait timeout in RTOS ticks. If this many ticks pass without bytes becoming available in the DMA receive buffer, then the function will return (note that if data is read from the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass portMAX_DELAY for no timeout.

esp_err_t **i2s_set_sample_rates**(*i2s_port_t i2s_num*, uint32_t *rate*)
Set sample rate used for I2S RX and TX.

Note The bit clock rate is determined by the sample rate and *i2s_config_t* configuration parameters (number of channels, bits_per_sample). `bit_clock = rate * (number of channels) * bits_per_sample`

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

Parameters

- `i2s_num`: I2S_NUM_0
- `rate`: I2S sample rate (ex: 8000, 44100...)

esp_err_t **i2s_stop**(*i2s_port_t i2s_num*)
Stop I2S driver.

Note Disables I2S TX/RX, until i2s_start() is called.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2s_num`: I2S_NUM_0

esp_err_t **i2s_start**(*i2s_port_t i2s_num*)
Start I2S driver.

Note It is not necessary to call this function after i2s_driver_install() (it is started automatically), however it is necessary to call it after i2s_stop().

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2s_num`: I2S_NUM_0

esp_err_t **i2s_zero_dma_buffer**(*i2s_port_t i2s_num*)
Zero the contents of the TX DMA buffer.

Note Pushes zero-byte samples into the TX DMA buffer, until it is full.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2s_num`: I2S_NUM_0

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.

Note Similar to i2s_set_sample_rates(), but also sets bit width.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

Parameters

- `i2s_num`: I2S_NUM_0
- `rate`: I2S sample rate (ex: 8000, 44100...)
- `bits`: I2S bit width (I2S_BITS_PER_SAMPLE_16BIT, I2S_BITS_PER_SAMPLE_24BIT)
- `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 **dma_buf_count**
I2S DMA Buffer Count

int **dma_buf_len**
I2S DMA Buffer Length

bool **tx_desc_auto_clear**
I2S auto clear tx descriptor if there is underflow condition (helps in avoiding noise in case of data unavailability)

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 enable for i2s_set_pin.

Public Members

int **bck_o_en**
BCK out pin

int **ws_o_en**
WS out pin

int **bck_i_en**
BCK in pin

int **ws_i_en**
WS in pin

int **data_out_en**
DATA out pin

int **data_in_en**
DATA in pin

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

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

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 i2s_port_t
I2S Peripheral, 0.

Values:

I2S_NUM_0 = 0x0
I2S 0

I2S_NUM_MAX

enum i2s_mode_t
I2S Mode, defaut is I2S_MODE_MASTER | I2S_MODE_TX.

Values:

I2S_MODE_MASTER = 1

I2S_MODE_SLAVE = 2

I2S_MODE_TX = 4

I2S_MODE_RX = 8

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

2.1.4 SPI

API Reference

Header File

- esp8266/include/driver/spi.h

Functions

esp_err_t **spi_get_clk_div**(*spi_host_t host*, *spi_clk_div_t *clk_div*)
Get the SPI clock division factor.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `clk_div`: Pointer to accept clock division factor

esp_err_t **spi_get_intr_enable**(*spi_host_t host*, *spi_intr_enable_t *intr_enable*)
Get SPI Interrupt Enable.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `intr_enable`: Pointer to accept interrupt enable

esp_err_t **spi_get_mode**(*spi_host_t host*, *spi_mode_t *mode*)
Get SPI working mode.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `mode`: Pointer to accept working mode

esp_err_t **spi_get_interface**(*spi_host_t host*, *spi_interface_t *interface*)

Get SPI bus interface configuration.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `interface`: Pointer to accept bus interface configuration

esp_err_t **spi_get_event_callback**(*spi_host_t host*, *spi_event_callback_t *event_cb*)

Get the SPI event callback function.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `event_cb`: Pointer to accept event callback function

esp_err_t **spi_set_clk_div**(*spi_host_t host*, *spi_clk_div_t *clk_div*)

Set the SPI clock division factor.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `clk_div`: Pointer to deliver clock division factor

esp_err_t **spi_set_intr_enable**(*spi_host_t host*, *spi_intr_enable_t *intr_enable*)
Set SPI interrupt enable.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `intr_enable`: Pointer to deliver interrupt enable

esp_err_t **spi_set_mode**(*spi_host_t host*, *spi_mode_t *mode*)
Set the SPI mode of operation.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `mode`: Pointer to deliver working mode

esp_err_t **spi_get_dummy**(*spi_host_t host*, uint16_t *bitlen*)
Get SPI dummy bitlen.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `bitlen`: Pointer to accept dummy bitlen

esp_err_t **spi_set_dummy**(*spi_host_t host*, uint16_t **bitlen*)
Set SPI dummy bitlen.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `bitlen`: Pointer to deliver dummy bitlen

esp_err_t **spi_set_interface**(*spi_host_t host*, *spi_interface_t *interface*)
Set SPI bus interface configuration.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `interface`: Pointer to deliver bus interface configuration

esp_err_t **spi_set_event_callback**(*spi_host_t host*, *spi_event_callback_t *event_cb*)
Set the SPI event callback function.

Note This event_cb 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 SPI interrupt handler due to the additional level of indirection.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `event_cb`: Pointer to deliver event callback function

esp_err_t **spi_slave_get_status**(*spi_host_t host*, uint32_t **status*)
Get SPI slave wr_status register.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `status`: Pointer to accept wr_status register

esp_err_t **spi_slave_set_status**(*spi_host_t host*, uint32_t **status*)
Set SPI slave rd_status register.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `status`: Pointer to deliver rd_status register

esp_err_t **spi_trans**(*spi_host_t host*, *spi_trans_t *trans*)
SPI data transfer function.

Note If the bit of the corresponding phase in the transmission parameter is 0, its data will not work. For example: trans.bits.cmd = 0, cmd will not be transmitted

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `trans`: Pointer to transmission parameter structure

esp_err_t **spi_deinit** (*spi_host_t host*)
Deinit the spi.

Return

- ESP_OK Success
- ESP_FAIL spi has not been initialized yet

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1

esp_err_t **spi_init** (*spi_host_t host*, *spi_config_t *config*)
Initialize the spi.

Note SPI0 has been used by FLASH and cannot be used by the user temporarily.

Return

- ESP_OK Success
- ESP_ERR_NO_MEM malloc fail
- ESP_FAIL spi has been initialized

Parameters

- `host`: SPI peripheral number
  - CSPI_HOST SPI0
  - HSPI_HOST SPI1
- `config`: Pointer to deliver initialize configuration parameter

Unions

union spi_intr_enable_t
#include <*spi.h*> SPI interrupt enable union type definition.

Public Members

uint32_t **read_buffer**
configurate intterrupt to enable reading

uint32_t **write_buffer**
configurate intterrupt to enable writing

uint32_t **read_status**
configurate intterrupt to enable reading status

uint32_t **write_status**
configurate intterrupt to enable writing status

uint32_t **trans_done**
configurate intterrupt to enable transmission done

uint32_t **reserved5**
reserved

struct *spi_intr_enable_t*::**[anonymous] [anonymous]**
not filled

uint32_t **val**
union fill

union spi_interface_t
#include <*spi.h*> SPI bus interface parameter union type definition.

Public Members

uint32_t **cpol**
Clock Polarity

uint32_t **cpha**
Clock Phase

uint32_t **bit_tx_order**
Tx bit order

uint32_t **bit_rx_order**
Rx bit order

uint32_t **byte_tx_order**
Tx byte order

uint32_t **byte_rx_order**
Rx byte order

uint32_t **mosi_en**
MOSI line enable

uint32_t **miso_en**
MISO line enable

uint32_t **cs_en**
CS line enable

uint32_t **reserved9**
resserved

struct *spi_interface_t*::**[anonymous] [anonymous]**
not filled

uint32_t **val**
union fill

Structures

struct spi_trans_t
SPI transmission parameter structure type definition.

Public Members

uint16_t ***cmd**
SPI transmission command

uint32_t ***addr**
SPI transmission address

uint32_t ***mosi**
SPI transmission MOSI buffer, in order to improve the transmission efficiency, it is recommended that the external incoming data is (uint32_t *) type data, do not use other type data.

uint32_t ***miso**
SPI transmission MISO buffer, in order to improve the transmission efficiency, it is recommended that the external incoming data is (uint32_t *) type data, do not use other type data.

uint32_t **cmd**
SPI transmission command bits

uint32_t **addr**
SPI transmission address bits

uint32_t **mosi**
SPI transmission MOSI buffer bits

uint32_t **miso**
SPI transmission MISO buffer bits

uint32_t **val**
union fill

union *spi_trans_t*::**[anonymous]** **bits**
SPI transmission packet members' bits

struct spi_config_t
SPI initialization parameter structure type definition.

Public Members

spi_interface_t **interface**
SPI bus interface

spi_intr_enable_t **intr_enable**
check if enable SPI interrupt

spi_event_callback_t **event_cb**
SPI interrupt event callback

spi_mode_t **mode**
SPI mode

spi_clk_div_t **clk_div**
SPI clock divider

Macros

SPI_NUM_MAX

SPI_CPOL_LOW

SPI_CPOL_HIGH

SPI_CPHA_LOW

SPI_CPHA_HIGH

SPI_BIT_ORDER_MSB_FIRST

SPI_BIT_ORDER_LSB_FIRST

SPI_BYTE_ORDER_MSB_FIRST

SPI_BYTE_ORDER_LSB_FIRST

SPI_DEFAULT_INTERFACE

SPI_MASTER_DEFAULT_INTR_ENABLE

SPI_SLAVE_DEFAULT_INTR_ENABLE

SPI_INIT_EVENT

SPI_TRANS_START_EVENT

SPI_TRANS_DONE_EVENT

SPI_DEINIT_EVENT

SPI_MASTER_WRITE_DATA_TO_SLAVE_CMD

SPI_MASTER_READ_DATA_FROM_SLAVE_CMD

SPI_MASTER_WRITE_STATUS_TO_SLAVE_CMD

SPI_MASTER_READ_STATUS_FROM_SLAVE_CMD

SPI_SLV_RD_BUF_DONE

SPI_SLV_WR_BUF_DONE

SPI_SLV_RD_STA_DONE

SPI_SLV_WR_STA_DONE

SPI_TRANS_DONE

Type Definitions

typedef void (***spi_event_callback_t**)(int event, void *arg)

Enumerations

enum spi_host_t

SPI peripheral enumeration.

Note ESP8266 has two hardware SPI, CSPI and HSPI. Currently, HSPI can be used arbitrarily.

Values:

CSPI_HOST = 0

HSPI_HOST

enum spi_clk_div_t
SPI clock division factor enumeration.

Values:

SPI_2MHz_DIV = 40

SPI_4MHz_DIV = 20

SPI_5MHz_DIV = 16

SPI_8MHz_DIV = 10

SPI_10MHz_DIV = 8

SPI_16MHz_DIV = 5

SPI_20MHz_DIV = 4

SPI_40MHz_DIV = 2

SPI_80MHz_DIV = 1

enum spi_mode_t
SPI working mode enumeration.

Values:

SPI_MASTER_MODE

SPI_SLAVE_MODE

2.1.5 PWM

API Reference

Header File

- esp8266/include/driver/pwm.h

Functions

esp_err_t **pwm_init** (uint32_t *period*, uint32_t **duties*, uint8_t *channel_num*, **const** uint32_t **pin_num*)
PWM function initialization, including GPIO, frequency and duty cycle.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Init error

Parameters

- `period`: PWM period, unit: us. e.g. For 1KHz PWM, period is 1000 us. Do not set the period below 20us.
- `duties`: duty cycle of each channels.

- `channel_num`: PWM channel number, maximum is 8
- `pin_num`: GPIO number of PWM channel

esp_err_t **pwm_deinit**(void)
PWM function uninstall.

Return

- ESP_OK Success
- ESP_FAIL Init error

esp_err_t **pwm_set_duty**(uint8_t *channel_num*, uint32_t *duty*)
Set the duty cycle of a PWM channel. Set the time that high level or low(if you invert the output of this channel) signal will last, the duty cycle cannot exceed the period.

Note After set configuration, pwm_start needs to be called to take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel_num`: PWM channel number the channel_num cannot exceed the value initialized by pwm_init.
- `duty`: duty cycle

esp_err_t **pwm_get_duty**(uint8_t *channel_num*, uint32_t **duty_p*)
Get the duty cycle of a PWM channel.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel_num`: PWM channel number the channel_num cannot exceed the value initialized by pwm_init.
- `duty_p`: pointer saves the address of the specified channel duty cycle

esp_err_t **pwm_set_period**(uint32_t *period*)
Set PWM period, unit: us.

Note After set configuration, pwm_start needs to be called to take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `period`: PWM period, unit: us For example, for 1KHz PWM, period is 1000. Do not set the period below 20us.

esp_err_t **pwm_get_period**(uint32_t **period_p*)
Get PWM period, unit: us.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `period_p`: pointer saves the address of the period

esp_err_t **pwm_start**(void)
Starts PWM.

Note This function needs to be called after PWM configuration is changed.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **pwm_stop**(uint32_t *stop_level_mask*)
Stop all PWM channel. Stop PWM and set the output of each channel to the specified level. Calling pwm_start can re-start PWM output.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `stop_level_mask`: Out put level after PWM is stoped e.g. We initialize 8 channels, if stop_level_mask = 0x0f, channel 0,1,2 and 3 will output high level, and channel 4,5,6 and 7 will output low level.

esp_err_t **pwm_set_duties**(uint32_t **duties*)
Set the duty cycle of all channels.

Note After set configuration, pwm_start needs to be called to take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `duties`: An array that store the duty cycle of each channel, the array elements number needs to be the same as the number of channels.

esp_err_t **pwm_set_phase**(uint8_t *channel_num*, float *phase*)
Set the phase of a PWM channel.

Note After set configuration, pwm_start needs to be called to take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel_num`: PWM channel number the channel_num cannot exceed the value initialized by pwm_init.
- `phase`: The phase of this PWM channel, the phase range is (-180 ~ 180].

esp_err_t **pwm_set_phases**(float **phases*)
Set the phase of all channels.

Note After set configuration, pwm_start needs to be called to take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `phases`: An array that store the phase of each channel, the array elements number needs to be the same as the number of channels.

esp_err_t **pwm_get_phase**(uint8_t *channel_num*, float **phase_p*)
Get the phase of a PWM channel.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel_num`: PWM channel number the channel_num cannot exceed the value initialized by pwm_init.
- `phase_p`: pointer saves the address of the specified channel phase

esp_err_t **pwm_set_period_duties**(uint32_t *period*, uint32_t **duties*)
Set PWM period and duty of each PWM channel.

Note After set configuration, pwm_start needs to be called to take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `period`: PWM period, unit: us For example, for 1KHz PWM, period is 1000.
- `duties`: An array that store the duty cycle of each channel, the array elements number needs to be the same as the number of channels.

esp_err_t **pwm_set_channel_invert**(uint16_t *channel_mask*)
Set the inverting output PWM channel.

Note After set configuration, pwm_start needs to be called to take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel_mask`: The channel bitmask that used to invert the output e.g. We initialize 8 channels, if channel_mask = 0x0f, channels 0, 1, 2 and 3 will invert the output.

esp_err_t **pwm_clear_channel_invert**(uint16_t *channel_mask*)
Clear the inverting output PWM channel. This function only works for the PWM channel that is already in the inverted output states.

Note After set configuration, pwm_start needs to be called to take effect.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel_mask`: The channel bitmask that need to clear e.g. The outputs of channels 0, 1, 2 and 3 are already in inverted state. If channel_mask = 0x07, the output of channel 0, 1, and 2 will return to normal, the channel 3 will keep inverting output.

2.1.6 UART

API Reference

Header File

- esp8266/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_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `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_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `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_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number
- `stop_bits`: Uart stop bits

esp_err_t **uart_get_stop_bits**(*uart_port_t uart_num*, *uart_stop_bits_t *stop_bits*)
Get UART stop bits.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `stop_bits`: Pointer to accept value of UART stop bits.

esp_err_t **uart_set_parity**(*uart_port_t uart_num*, *uart_parity_t parity_mode*)
Set UART parity mode.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `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_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number
- `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_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number
- `baudrate`: UART baud rate.

esp_err_t **uart_get_baudrate**(*uart_port_t uart_num*, uint32_t **baudrate*)
Get UART baud rate.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `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_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART_NUM_0
- `inverse_mask`: Choose the wires that need to be inverted. Inverse_mask should be chosen from UART_INVERSE_RXD / UART_INVERSE_TXD / UART_INVERSE_RTS / UART_INVERSE_CTS, combined with OR operation.

esp_err_t **uart_set_hw_flow_ctrl**(*uart_port_t uart_num*, *uart_hw_flowcontrol_t flow_ctrl*, uint8_t *rx_thresh*)
Configure Hardware flow control.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `flow_ctrl`: Hardware flow control mode.
- `rx_thresh`: Threshold of Hardware flow control.

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_OK Success, result will be put in (*flow_ctrl)
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `flow_ctrl`: Option for different flow control mode.

esp_err_t **uart_enable_swap** (void)
UART0 swap. Use MTCK as UART0 RX, MTDO as UART0 TX, so ROM log will not output from this new UART0. We also need to use MTDO (U0RTS) and MTCK (U0CTS) as UART0 in hardware.

Return

- ESP_OK Success

esp_err_t **uart_disable_swap** (void)
Disable UART0 swap. Use the original UART0, not MTCK and MTDO.

Return

- ESP_OK Success

esp_err_t **uart_clear_intr_status** (*uart_port_t uart_num*, uint32_t *mask*)
Clear uart interrupts status.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `mask`: Uart interrupt bits mask.

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_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number

- `enable_mask`: Bit mask of the enable bits. The bit mask should be composed from the fields of register UART_INT_ENA_REG.

esp_err_t **uart_disable_intr_mask**(*uart_port_t uart_num*, uint32_t *disable_mask*)
Clear UART interrupt enable bits.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number
- `disable_mask`: Bit mask of the disable bits. The bit mask should be composed from the fields of register UART_INT_ENA_REG.

esp_err_t **uart_enable_rx_intr**(*uart_port_t uart_num*)
Enable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART_NUM_0

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_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART_NUM_0

esp_err_t **uart_disable_tx_intr**(*uart_port_t uart_num*)
Disable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART_NUM_0

esp_err_t **uart_enable_tx_intr**(*uart_port_t uart_num*, int *enable*, int *thresh*)
Enable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART_NUM_0
- `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*Register UART interrupt handler (ISR).

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART_NUM_0
- `fn`: Interrupt handler function.
- `arg`: parameter for handler function

esp_err_t **uart_param_config**(*uart_port_t uart_num*, *uart_config_t *uart_conf*)
Config Common parameters of serial ports.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `uart_conf`: Uart config parameters.

esp_err_t **uart_intr_config**(*uart_port_t uart_num*, *uart_intr_config_t *uart_intr_conf*)
Config types of uarts.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `uart_intr_conf`: Uart interrupt config parameters.

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 *no_use*)
Install UART driver.

Note Rx_buffer_size should be greater than UART_FIFO_LEN. Tx_buffer_size should be either zero or greater than UART_FIFO_LEN.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `rx_buffer_size`: UART RX ring buffer size.
- `tx_buffer_size`: UART TX ring buffer size. If set to zero, driver will not use TX buffer, TX function will block task until all data have been sent out.
- `queue_size`: UART event queue size/depth.
- `uart_queue`: UART event queue handle (out param). On success, a new queue handle is written here to provide access to UART events. If set to NULL, driver will not use an event queue.
- `no_use`: Invalid parameters, just to fit some modules.

esp_err_t **uart_driver_delete**(*uart_port_t uart_num*)

Uninstall UART driver.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.

esp_err_t **uart_wait_tx_done**(*uart_port_t uart_num*, TickType_t *ticks_to_wait*)

Waiting for the last byte of data to be sent.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart port number.
- `ticks_to_wait`: Timeout, count in RTOS ticks

int **uart_tx_chars**(*uart_port_t uart_num*, **const** char **buffer*, uint32_t *len*)

Send data to the UART port from a given buffer and length.

This function will not wait for enough space in TX FIFO. It will just fill the available TX FIFO and return when the FIFO is full.

Note This function should only be used when UART TX buffer is not enabled.

Return

- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

Parameters

- `uart_num`: Uart port number.
- `buffer`: data buffer address
- `len`: data length to send

int **uart_write_bytes**(*uart_port_t uart_num*, **const** char ***src*, size_t *size*)

Send data to the UART port from a given buffer and length,.

If the UART driver's parameter 'tx_buffer_size' is set to zero: This function will not return until all the data have been sent out, or at least pushed into TX FIFO.

Otherwise, if the 'tx_buffer_size' > 0, this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually.

Return

- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

Parameters

- `uart_num`: Uart port number.
- `src`: data buffer address
- `size`: data length to send

int **uart_read_bytes**(*uart_port_t uart_num*, uint8_t ***buf*, uint32_t *length*, TickType_t *ticks_to_wait*)

UART read bytes from UART buffer.

Return

- (-1) Error
- OTHERS (>=0) The number of bytes read from UART FIFO

Parameters

- `uart_num`: Uart port number.
- `buf`: pointer to the buffer.
- `length`: data length
- `ticks_to_wait`: sTimeout, count in RTOS ticks

esp_err_t **uart_flush**(*uart_port_t uart_num*)

Alias of uart_flush_input. UART ring buffer flush. This will discard all data in the UART RX buffer.

Note Instead of waiting the data sent out, this function will clear UART rx buffer. In order to send all the data in tx FIFO, we can use uart_wait_tx_done function.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART port number.

esp_err_t **uart_flush_input** (*uart_port_t uart_num*)
Clear input buffer, discard all the data is in the ring-buffer.

Note In order to send all the data in tx FIFO, we can use uart_wait_tx_done function.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART port number.

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_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: UART port number.
- `size`: Pointer of size_t to accept cached data length

esp_err_t **uart_set_rx_timeout** (*uart_port_t uart_num*, **const** uint8_t *tout_thresh*)
UART set threshold timeout for TOUT feature.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart number to configure
- `tout_thresh`: This parameter defines timeout threshold in uart symbol periods. The maximum value of threshold is 126. tout_thresh = 1, defines TOUT interrupt timeout equal to transmission time of one symbol (~11 bit) on current baudrate. If the time is expired the UART_RXFIFO_TOUT_INT interrupt is triggered. If tout_thresh == 0, the TOUT feature is disabled.

bool **uart_is_driver_installed** (*uart_port_t uart_num*)
Checks whether the driver is installed or not.

Return

- true driver is installed
- false driver is not installed

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

Structures

struct uart_config_t
UART configuration parameters for uart_param_config function.

Public Members

int **baud_rate**
UART baud rate

uart_word_length_t **data_bits**
UART byte size

uart_parity_t **parity**
UART parity mode

uart_stop_bits_t **stop_bits**
UART stop bits

uart_hw_flowcontrol_t **flow_ctrl**
UART HW flow control mode (cts/rts)

uint8_t **rx_flow_ctrl_thresh**
UART HW RTS threshold

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_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

Enumerations

enum uart_mode_t
UART mode selection.

Values:

UART_MODE_UART = 0x00
mode: regular UART mode

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_NUM_1 = 0x1

UART_NUM_MAX

enum uart_parity_t
UART parity constants.

Values:

UART_PARITY_DISABLE = 0x0
Disable UART parity

UART_PARITY_EVEN = 0x2
Enable UART even parity

UART_PARITY_ODD = 0x3
Enable UART odd parity

enum uart_hw_flowcontrol_t
UART hardware flow control modes.

Values:

UART_HW_FLOWCTRL_DISABLE = 0x0
disable hardware flow control

UART_HW_FLOWCTRL_RTS = 0x1
enable RX hardware flow control (rts)

UART_HW_FLOWCTRL_CTS = 0x2
enable TX hardware flow control (cts)

UART_HW_FLOWCTRL_CTS_RTS = 0x3
enable hardware flow control

UART_HW_FLOWCTRL_MAX = 0x4

enum uart_event_type_t
UART event types used in the ring buffer.

Values:

UART_DATA
UART data event

UART_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_EVENT_MAX
UART event max index

2.1.7 ADC

API Reference

Header File

- esp8266/include/driver/adc.h

Functions

esp_err_t **adc_read**(uint16_t **data*)
Single measurement of TOUT(ADC) pin, unit : 1/1023 V or VDD pin, uint: 1 mV.

Note When measuring VDD pin voltage, the TOUT(ADC) pin must be left floating.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL adc has not been initialized yet

Parameters

- `data`: Pointer to accept adc value.

esp_err_t **adc_read_fast**(uint16_t **data*, uint16_t *len*)
Measure the input voltage of TOUT(ADC) pin, unit : 1/1023 V.

Note Wi-Fi and interrupts need to be turned off.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL adc has not been initialized yet

Parameters

- `data`: Pointer to accept adc value. Input voltage of TOUT(ADC) pin, unit : 1/1023 V
- `len`: Receiving length of ADC value, range [1, 65535]

esp_err_t **adc_deinit**()
Deinit the adc.

Return

- ESP_OK Success
- ESP_FAIL adc has not been initialized yet

esp_err_t **adc_init** (*adc_config_t* **config*)
Initialize the adc.

Note First modify menuconfig->Component config->PHY->vdd33_const value, vdd33_const provides ADC mode settings, i.e. selecting system voltage or external voltage measurements. When measuring system voltage, it must be set to 255. To read the external voltage on TOUT(ADC) pin, vdd33_const need less than 255 When the ADC reference voltage is set to the actual VDD33 power supply voltage, the value range of vdd33_const is [18,36], the unit is 0.1V. When the ADC reference voltage is set to the default value of 3.3V as the supply voltage, the range of vdd33_const is [0, 18] or (36, 255).

Return

- ESP_OK Success
- ESP_ERR_NO_MEM malloc fail
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL adc has been initialized

Parameters

- `config`: Pointer to deliver initialize configuration parameter

Structures

struct adc_config_t
ADC initialization parameter structure type definition.

Public Members

adc_mode_t **mode**
ADC mode

uint8_t **clk_div**
ADC sample collection clock=80M/clk_div, range[8, 32]

Enumerations

enum adc_mode_t
ADC working mode enumeration.

Values:

ADC_READ_TOUT_MODE = 0

ADC_READ_VDD_MODE

ADC_READ_MAX_MODE

2.1.8 Hardware Timer

API Reference

Header File

- esp8266/include/driver/hw_timer.h

Functions

esp_err_t **hw_timer_set_clkdiv**(*hw_timer_clkdiv_t clkdiv*)
Set the frequency division coefficient of hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL hardware timer has been initialized

Parameters

- `clkdiv`: frequency division coefficient

uint32_t **hw_timer_get_clkdiv**()
Get the frequency division coefficient of hardware timer.

Return

- 0 TIMER_CLKDIV_1
- 4 TIMER_CLKDIV_16
- 8 TIMER_CLKDIV_256

esp_err_t **hw_timer_set_intr_type**(*hw_timer_intr_type_t intr_type*)
Set the interrupt type of hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL hardware timer has been initialized

Parameters

- `intr_type`: interrupt type

uint32_t **hw_timer_get_intr_type**()
Get the interrupt type of hardware timer.

Return

- 0 TIMER_EDGE_INT
- 1 TIMER_LEVEL_INT

esp_err_t **hw_timer_set_reload**(bool *reload*)
Enable hardware timer reload.

Return

- ESP_OK Success
- ESP_FAIL hardware timer has been initialized

Parameters

- `reload`: false, one-shot mode; true, reload mode

bool **hw_timer_get_reload**()
Get the hardware timer reload status.

Return

- true reload mode
- false one-shot mode

esp_err_t **hw_timer_enable**(bool *en*)
Enable hardware timer.

Return

- ESP_OK Success
- ESP_FAIL hardware timer has been initialized

Parameters

- `en`: false, hardware timer disable; true, hardware timer enable

bool **hw_timer_get_enable**()
Get the hardware timer enable status.

Return

- true hardware timer has been enabled
- false hardware timer is not yet enabled

esp_err_t **hw_timer_set_load_data**(uint32_t *load_data*)
Set the hardware timer load value.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL hardware timer has been initialized

Parameters

- `load_data`: hardware timer load value
  - FRC1 hardware timer, range : less than 0x1000000

uint32_t **hw_timer_get_load_data**()
Get the hardware timer load value.

Return load value

uint32_t **hw_timer_get_count_data**()
Get the hardware timer count value.

Return count value

esp_err_t **hw_timer_deinit**(void)
deinit the hardware timer

Return

- ESP_OK Success
- ESP_FAIL hardware timer has not been initialized yet

esp_err_t **hw_timer_init**(*hw_timer_callback_t callback*, void **arg*)
Initialize the hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL hardware timer has been initialized

Parameters

- `callback`: user hardware timer callback function
- `arg`: parameter for ISR handler

esp_err_t **hw_timer_alarm_us**(uint32_t *value*, bool *reload*)
Set a trigger timer us delay to enable this timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL hardware timer has not been initialized yet

Parameters

- `value`:
  - If reload is true, range : 50 ~ 0x199999
  - If reload is false, range : 10 ~ 0x199999
- `reload`: false, one-shot mode; true, reload mode.

esp_err_t **hw_timer_disarm**(void)
disable this timer

Return

- ESP_OK Success
- ESP_FAIL hardware timer has not been initialized yet

Macros

TIMER_BASE_CLK

Type Definitions

typedef void (***hw_timer_callback_t**)(void *arg)

Enumerations

enum hw_timer_clkdiv_t

Values:

TIMER_CLKDIV_1 = 0

TIMER_CLKDIV_16 = 4

TIMER_CLKDIV_256 = 8

enum hw_timer_intr_type_t

Values:

TIMER_EDGE_INT = 0

TIMER_LEVEL_INT = 1

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

2.2 Wi-Fi API

2.2.1 Wi-Fi

Introduction

The WiFi libraries provide support for configuring and monitoring the ESP8266 WiFi networking functionality. This includes configuration for:

- Station mode (aka STA mode or WiFi client mode). ESP8266 connects to an access point.
- AP mode (aka Soft-AP mode or Access Point mode). Stations connect to the ESP8266.
- Combined AP-STA mode (ESP8266 is concurrently an access point and a station connected to another access point).
- Various security modes for the above (WPA, WPA2, WEP, etc.)
- Scanning for access points (active & passive scanning).
- Promiscuous mode monitoring of IEEE802.11 WiFi packets.

Application Examples

See wifi directory of ESP8266_RTOS_SDK examples that contains the following applications:

- Simple application showing how to connect ESP8266 module to an Access Point - template.

API Reference

Header File

- esp8266/include/esp_wifi.h

Functions

esp_err_t **esp_wifi_init**(**const** *wifi_init_config_t* *config)

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

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

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

Return

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

Parameters

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

esp_err_t **esp_wifi_deinit**(void)

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

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

Return ESP_OK: succeed

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

Set the WiFi operating mode.

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

Return

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

Parameters

- `mode`: WiFi operating mode

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

Get current operating mode of WiFi.

Return

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

Parameters

- `mode`: store current WiFi mode

esp_err_t **esp_wifi_start**(void)

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

Return

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

esp_err_t **esp_wifi_stop**(void)

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

Return

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

esp_err_t **esp_wifi_restore**(void)

Restore WiFi stack persistent settings to default values.

This function will reset settings made using the following APIs:

- esp_wifi_get_auto_connect,
- esp_wifi_set_protocol,
- esp_wifi_set_config related
- esp_wifi_set_mode

Return

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

esp_err_t **esp_wifi_connect**(void)

Connect the ESP8266 WiFi station to the AP.

Attention 1. This API only impact WIFI_MODE_STA or WIFI_MODE_APSTA mode

Attention 2. If the ESP8266 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 ESP8266 WiFi station from the AP.

Return

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

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

Return

- ESP_OK: succeed
- others: fail

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

Return

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

Parameters

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

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

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

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

Return

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

Parameters

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

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

Return

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

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

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

Return

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

Parameters

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

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

Return

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

Parameters

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

esp_err_t **esp_wifi_sta_get_ap_info**(*wifi_ap_record_t* *ap_info)
Get information of AP which the ESP8266 station is associated with.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_CONN: The station interface don't initialized
- ESP_ERR_WIFI_NOT_CONNECT: The station is in disconnect status

Parameters

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

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

Attention Default power save type is WIFI_PS_NONE.

Return ESP_ERR_NOT_SUPPORTED: 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_NOT_SUPPORTED: 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)

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

Attention Please call this API in SYSTEM_EVENT_STA_START event

Return

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

Parameters

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

esp_err_t **esp_wifi_get_protocol**(*wifi_interface_t ifx*, uint8_t **protocol_bitmap*)
Get the current protocol bitmap of the specified interface.

Return

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

Parameters

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

esp_err_t **esp_wifi_set_bandwidth**(*wifi_interface_t ifx*, *wifi_bandwidth_t bw*)
Set the bandwidth of ESP8266 specified interface.

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

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

Return

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

Parameters

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

esp_err_t **esp_wifi_get_bandwidth**(*wifi_interface_t ifx*, *wifi_bandwidth_t *bw*)
Get the bandwidth of ESP8266 specified interface.

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

Return

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

Parameters

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

esp_err_t **esp_wifi_set_channel**(uint8_t *primary*, *wifi_second_chan_t second*)
Set primary/secondary channel of ESP8266.

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_INVALID_ARG: invalid argument

Parameters

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

esp_err_t **esp_wifi_get_channel**(uint8_t **primary*, *wifi_second_chan_t *second*)
Get the primary/secondary channel of ESP8266.

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

Return

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

Parameters

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

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

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

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

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

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

Attention 5. The country configuration is not stored into flash

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

Return

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

Parameters

- `country`: the configured country info

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

Return

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

Parameters

- `country`: country info

esp_err_t **esp_wifi_set_mac**(*wifi_interface_t ifx*, **const** uint8_t *mac*[6])
Set MAC address of the ESP8266 WiFi station or the soft-AP interface.

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

Attention 2. ESP8266 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 ESP8266 MAC address can not be 1. For example, the MAC address can set to be "1a:XX:XX:XX:XX:XX", but can not be "15:XX:XX:XX:XX:XX".

Return

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

Parameters

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

esp_err_t **esp_wifi_get_mac**(*wifi_interface_t ifx*, uint8_t *mac*[6])
Get mac of specified interface.

Return

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

Parameters

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

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

Register the RX callback function in the promiscuous mode.

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

Return

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

Parameters

- `cb`: callback

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

Enable the promiscuous mode.

Return

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

Parameters

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

esp_err_t **esp_wifi_get_promiscuous**(bool **en*)

Get the promiscuous mode.

Return

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

Parameters

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

esp_err_t **esp_wifi_set_promiscuous_filter**(**const** *wifi_promiscuous_filter_t *filter*)

Enable the promiscuous mode packet type filter.

Note The default filter is to filter all packets except WIFI_PKT_MISC

Return

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

Parameters

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

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

Return

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

Parameters

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

esp_err_t **esp_wifi_set_config**(*wifi_interface_t interface*, *wifi_config_t *conf*)
Set the configuration of the ESP8266 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. ESP8266 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 ESP8266 station.

Return

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

Parameters

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

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

Note The default filter is to filter none control packet.

Return

- ESP_OK: succeed

- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

Parameters

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

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

Return

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

Parameters

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

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

Return

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

Parameters

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

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

Attention SSC only API

Return

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

Parameters

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

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

Attention 1. The default value is WIFI_STORAGE_FLASH

Return

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

Parameters

- `storage`: : storage type

esp_err_t **esp_wifi_set_auto_connect** (bool *en*)

Set auto connect The default value is true.

Return

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

Parameters

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

esp_err_t **esp_wifi_get_auto_connect** (bool **en*)

Get the auto connect flag.

Return

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

Parameters

- `en`: store current auto connect configuration

esp_err_t **esp_wifi_set_vendor_ie** (bool *enable*, *wifi_vendor_ie_type_t type*, *wifi_vendor_ie_id_t idx*, **const** void **vnd_ie*)

Set 802.11 Vendor-Specific Information Element.

Return

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

Parameters

- `enable`: If true, specified IE is enabled. If false, specified IE is removed.
- `type`: Information Element type. Determines the frame type to associate with the IE.

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

esp_err_t **esp_wifi_set_vendor_ie_cb** (*esp_vendor_ie_cb_t cb*, void **ctx*)
Register Vendor-Specific Information Element monitoring callback.

Return

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

Parameters

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

esp_err_t **esp_wifi_set_max_tx_power** (int8_t *power*)
Set maximum WiFi transmiting power.

Attention Please Call this API after calling esp_wifi_start()

Attention WiFi transmiting power is divided to six levels in phy init data. Level0 represents highest transmiting power and level5 represents lowest transmiting power. Packets of different rates are transmitted in different powers according to the configuration in phy init data. This API only sets maximum WiFi transmiting power. If this API is called, the transmiting 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:

- [82, 127]: level0
- [78, 81]: level1
- [74, 77]: level2
- [68, 73]: level3
- [64, 67]: level4
- [56, 63]: level5
- [49, 55]: level5 - 2dBm
- [33, 48]: level5 - 6dBm
- [25, 32]: level5 - 8dBm
- [13, 24]: level5 - 11dBm
- [1, 12]: level5 - 14dBm
- [-128, 0]: level5 - 17.5dBm

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 transmiting power.

void **esp_wifi_set_max_tx_power_via_vdd33**(uint16_t *vdd33*)
Adjust RF Tx Power according to VDD33; unit : 1/1024 V.

Attention When TOUT pin is suspended, VDD33 can be got by esp_wifi_get_vdd33. When TOUT pin is wired to external circuitry, esp_wifi_get_vdd33 can not be used.

Attention This api only worked when it is called, please call this api every day or hour according to power consumption.

Parameters

- `vdd33`: unit is 1/1024V, range [1900, 3300].

uint16_t **esp_wifi_get_vdd33**(void)
Measure the power voltage of VDD3P3 pin 3 and 4; unit: 1/1024 V.

Attention esp_wifi_get_vdd33 can only be called when TOUT pin is suspended.

Attention The 107th byte in esp_init_data_default.bin (0 ~ 127 bytes) is named as vdd33_const. When TOUT pin is suspended, vdd33_const must be set as 0xFF, which is 255.

Attention The return value of esp_wifi_get_vdd33 may be different in different Wi-Fi modes, for example, in Modem-sleep mode or in normal Wi-Fi working mode.

Return the power voltage of vdd33 pin 3 and 4

esp_err_t **esp_wifi_get_max_tx_power**(int8_t **power*)
Get maximum WiFi transmiting 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_INVALID_ARG: invalid argument

Parameters

- `power`: Maximum WiFi transmiting power.

esp_err_t **esp_wifi_set_event_mask** (uint32_t *mask*)
Set mask to enable or disable some WiFi events.

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

Attention 2. Default WiFi event mask is WIFI_EVENT_MASK_AP_PROBEREQRECVED.

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

Return

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

Parameters

- `mask`: WiFi event mask.

esp_err_t **esp_wifi_get_event_mask** (uint32_t **mask*)
Get mask of WiFi events.

Return

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

Parameters

- `mask`: WiFi event mask.

esp_err_t **esp_wifi_80211_tx** (*wifi_interface_t ifx*, **const** void ***buffer*, int *len*, bool *en_sys_seq*)
Send user-define 802.11 packets.

Attention 1. Packet has to be the whole 802.11 packet, does not include the FCS. The length of the packet has to be longer than the minimum length of the header of 802.11 packet which is 24 bytes, and less than 1400 bytes.

Attention 2. Duration area is invalid for user, it will be filled in SDK.

Attention 3. The rate of sending packet is same as the management packet which is the same as the system rate of sending packets.

Attention 4. Only after the previous packet was sent, entered the sent callback, the next packet is allowed to send. Otherwise, wifi_send_pkt_freedom will return fail.

Return ESP_OK, succeed;

Return ESP_FAIL, fail.

Parameters

- `ifx`: interface if the Wi-Fi mode is Station, the ifx should be WIFI_IF_STA. If the Wi-Fi mode is SoftAP, the ifx should be WIFI_IF_AP. If the Wi-Fi mode is Station+SoftAP, the ifx should be WIFI_IF_STA or WIFI_IF_AP. If the ifx is wrong, the API returns ESP_ERR_WIFI_IF.
- `buffer`: pointer of packet
- `len`: packet length
- `en_sys_seq`: follow the system's 802.11 packets sequence number or not, if it is true, the sequence number will be increased 1 every time a packet sent.

wifi_state_t **esp_wifi_get_state**(void)

Operation system start check time and enter sleep.

Note This function is called by system, user should not call this

Return

- wifi state

esp_err_t **esp_wifi_set_rssi_threshold**(int32_t *rssi*)

Set RSSI threshold below which APP will get an event.

Attention This API needs to be called every time after WIFI_EVENT_STA_BSS_RSSI_LOW event is received.

Return

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

Parameters

- `rssi`: threshold value in dbm between -100 to 0

int64_t **esp_wifi_get_tsf_time**(*wifi_interface_t interface*)

Get the TSF time In Station mode or SoftAP+Station mode if station is not connected or station doesn't receive at least one beacon after connected, will return 0.

Attention Enabling power save may cause the return value inaccurate, except WiFi modem sleep

Return 0 or the TSF time

Parameters

- `interface`: The interface whose tsf_time is to be retrieved.

esp_err_t **esp_wifi_set_inactive_time**(*wifi_interface_t ifx*, uint16_t *sec*)

Set the inactive time of the ESP32 STA or AP.

Attention 1. For Station, If the station does not receive a beacon frame from the connected SoftAP during the inactive time, disconnect from SoftAP. Default 6s.

Attention 2. For SoftAP, If the softAP doesn't receive any data from the connected STA during inactive time, the softAP will force deauth the STA. Default is 300s.

Attention 3. The inactive time configuration is not stored into flash

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_WIFI_ARG: invalid argument, For Station, if sec is less than 3. For SoftAP, if sec is less than 10.

Parameters

- `ifx`: interface to be configured.
- `sec`: Inactive time. Unit seconds.

esp_err_t **esp_wifi_get_inactive_time**(*wifi_interface_t ifx*, uint16_t **sec*)
Get inactive time of specified interface.

Return

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

Parameters

- `ifx`: Interface to be configured.
- `sec`: Inactive time. Unit seconds.

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

void ***osi_funcs**
WiFi OS functions

uint8_t **qos_enable**
WiFi QOS feature enable flag

uint8_t **ampdu_rx_enable**
WiFi AMPDU RX feature enable flag

uint8_t **rx_ba_win**
WiFi Block Ack RX window size

uint8_t **rx_ampdu_buf_num**
WiFi AMPDU RX buffer number

uint32_t **rx_ampdu_buf_len**
WiFi AMPDU RX buffer length

uint32_t **rx_max_single_pkt_len**
WiFi RX max single packet size

uint32_t **rx_buf_len**
WiFi RX buffer size

uint8_t **amsdu_rx_enable**
WiFi AMSDU RX feature enable flag

uint8_t **rx_buf_num**
WiFi RX buffer number

uint8_t **rx_pkt_num**
WiFi RX packet number

uint8_t **left_continuous_rx_buf_num**
WiFi Rx left continuous rx buffer number

uint8_t **tx_buf_num**
WiFi TX buffer number

uint8_t **nvs_enable**
WiFi NVS flash enable flag

uint8_t **nano_enable**
Nano option for printf/scan family enable flag

uint8_t **wpa3_sae_enable**
WiFi WPA3 feature enable flag

uint32_t **magic**
WiFi init magic number, it should be the last field

Macros

ESP_ERR_WIFI_NOT_INIT
WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED
WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED
WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF
WiFi interface error

ESP_ERR_WIFI_MODE
WiFi mode error

ESP_ERR_WIFI_STATE
WiFi internal state error

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

ESP_ERR_WIFI_NVS
WiFi internal NVS module error

ESP_ERR_WIFI_MAC
MAC address is invalid

ESP_ERR_WIFI_SSID
SSID is invalid

ESP_ERR_WIFI_PASSWORD
Password is invalid

ESP_ERR_WIFI_TIMEOUT
Timeout error

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

ESP_ERR_WIFI_WOULD_BLOCK
The caller would block

ESP_ERR_WIFI_NOT_CONNECT
Station still in disconnect status

ESP_ERR_WIFI_PM_MODE_OPEN
Wifi is in min/max modem sleep mode

ESP_ERR_WIFI_FPM_MODE
Have not enable fpm mode

ESP_WIFI_PARAM_USE_NVS

WIFI_AMPDU_RX_ENABLED

WIFI_AMPDU_RX_BA_WIN

WIFI_RX_MAX_SINGLE_PKT_LEN

WIFI_AMPDU_RX_AMPDU_BUF_LEN

WIFI_AMPDU_RX_AMPDU_BUF_NUM

WIFI_HW_RX_BUFFER_LEN

WIFI_QOS_ENABLED

WIFI_AMSDU_RX_ENABLED

WIFI_NVS_ENABLED

WIFI_WPA3_ENABLED

WIFI_INIT_CONFIG_MAGIC

WIFI_INIT_CONFIG_DEFAULT()

Type Definitions

typedef void (***wifi_promiscuous_cb_t**)(void *buf, *wifi_promiscuous_pkt_type_t* type)
The RX callback function in the promiscuous mode. Each time a packet is received, the callback function will be called.

Parameters

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

typedef void (***esp_vendor_ie_cb_t**)(void *ctx, *wifi_vendor_ie_type_t* type, **const** uint8_t sa[6], **const** *vendor_ie_data_t* *vnd_ie, int rssi)
Function signature for received Vendor-Specific Information Element callback.

Parameters

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

Header File

- esp8266/include/esp_wifi_types.h

Unions

union wifi_scan_time_t

#include <esp_wifi_types.h> Aggregate of active & passive scan time per channel.

Public Members

wifi_active_scan_time_t **active**

active scan time per channel, units: millisecond.

uint32_t **passive**

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

union wifi_config_t

#include <esp_wifi_types.h> Configuration data for ESP8266 AP or STA.

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

Public Members

wifi_ap_config_t **ap**

configuration of AP

wifi_sta_config_t **sta**

configuration of STA

Structures

struct wifi_country_t

Structure describing WiFi country-based regional restrictions.

Public Members

char **cc**[3]
country code string

uint8_t **schan**
start channel

uint8_t **nchan**
total channel number

int8_t **max_tx_power**
maximum tx power

wifi_country_policy_t **policy**
country policy

struct wifi_active_scan_time_t
Range of active scan times per channel.

Public Members

uint32_t **min**
minimum active scan time per channel, units: millisecond

uint32_t **max**
maximum active scan time per channel, units: millisecond, values above 1500ms may cause station to disconnect from AP and are not recommended.

struct wifi_scan_config_t
Parameters for an SSID scan.

Public Members

uint8_t ***ssid**
SSID of AP

uint8_t ***bssid**
MAC address of AP

uint8_t **channel**
channel, scan the specific channel

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

wifi_scan_type_t **scan_type**
scan type, active or passive

wifi_scan_time_t **scan_time**
scan time per channel

struct wifi_ap_record_t
Description of a WiFi AP.

Public Members

uint8_t **bssid**[6]
MAC address of AP

uint8_t **ssid**[33]
SSID of AP

uint8_t **primary**
channel of AP

wifi_second_chan_t **second**
secondary channel of AP

int8_t **rssi**
signal strength of AP

int16_t **freq_offset**
frequency offset of AP

wifi_auth_mode_t **authmode**
authmode of AP

wifi_cipher_type_t **pairwise_cipher**
pairwise cipher of AP

wifi_cipher_type_t **group_cipher**
group cipher of AP

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

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

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

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

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

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

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

wifi_country_t **country**
country information of AP

struct wifi_fast_scan_threshold_t
Structure describing parameters for a WiFi fast scan.

Public Members

int8_t **rssi**
The minimum rssi to accept in the fast scan mode

wifi_auth_mode_t **authmode**
The weakest authmode to accept in the fast scan mode

struct esp_pm_config_t
Wi-Fi Power management config for ESP8266.

Pass a pointer to this structure as an argument to esp_wifi_set_pm_config function.

Public Members

uint8_t **max_bcn_early_ms**
max beacon early time(2~15ms), default 4ms.

uint8_t **max_bcn_timeout_ms**
max beacon timeout time(12~32ms), default 24ms.

uint8_t **wait_time**
wait time before close RF (10~100ms), default 20ms.

uint8_t **wait_tx_cnt**
wait cnt after tx packet done(1~20), default 2, real time = wait_tx_cnt * wait_time.

uint8_t **wait_rx_bdata_cnt**
wait cnt after rx broadcast packet(1~100), default 2, real time = wait_tx_cnt * wait_time.

uint8_t **wait_rx_udata_cnt**
wait cnt after rx unicast packet(1~100), default 4, real time = wait_tx_cnt * wait_time.

bool **recv_bdata**
Receive broadcast/multicast packet or not when WiFi in power save. default true(receive broadcast/multicast packet)

struct esp_pm_config_esp8266_t
Power management config for ESP8266.

Pass a pointer to this structure as an argument to esp_pm_configure function.

Public Members

int **max_freq_mhz**
Not used in ESP8266

int **min_freq_mhz**
Not used in ESP8266

bool **light_sleep_enable**
Enter light sleep when no locks are taken

struct wifi_pmf_config_t
Configuration structure for Protected Management Frame

Public Members

bool **capable**
Advertizes support for Protected Management Frame. Device will prefer to connect in PMF mode if other device also advertizes PMF capability.

bool **required**
Advertizes that Protected Management Frame is required. Device will not associate to non-PMF capable devices.

struct wifi_ap_config_t
Soft-AP configuration settings for the ESP8266.

Public Members

uint8_t **ssid**[32]
SSID of ESP8266 soft-AP

uint8_t **password**[64]
Password of ESP8266 soft-AP

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

uint8_t **channel**
Channel of ESP8266 soft-AP

wifi_auth_mode_t **authmode**
Auth mode of ESP8266 soft-AP. Do not support AUTH_WEP in soft-AP mode

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

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

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

struct wifi_sta_config_t
STA configuration settings for the ESP8266.

Public Members

uint8_t **ssid**[32]
SSID of target AP

uint8_t **password**[64]
password of target AP

wifi_scan_method_t **scan_method**
do all channel scan or fast scan

bool **bssid_set**
whether set MAC address of target AP or not. Generally, station_config.bssid_set needs to be 0; and it needs to be 1 only when users need to check the MAC address of the AP.

uint8_t **bssid**[6]
MAC address of target AP

uint8_t **channel**
channel of target AP. Set to 1~13 to scan starting from the specified channel before connecting to AP. If the channel of AP is unknown, set it to 0.

uint16_t **listen_interval**
Listen interval for ESP8266 station to receive beacon when WIFI_PS_MAX_MODEM is set. Units: AP beacon intervals. Defaults to 3 if set to 0.

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

wifi_fast_scan_threshold_t **threshold**
When scan_method is set to WIFI_FAST_SCAN, only APs which have an auth mode that is more secure than the selected auth mode and a signal stronger than the minimum RSSI will be used.

wifi_pmf_config_t **pmf_cfg**
Configuration for Protected Management Frame. Will be advertized in RSN Capabilities in RSN IE.

uint32_t **rm_enabled**
Whether radio measurements are enabled for the connection

uint32_t **btm_enabled**
Whether BTM is enabled for the connection

uint32_t **reserved**
Reserved for future feature set

struct wifi_sta_info_t
Description of STA associated with AP.

Public Members

uint8_t **mac**[6]
mac address

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

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

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

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

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

struct wifi_sta_list_t
List of stations associated with the ESP8266 Soft-AP.

Public Members

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

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

struct vendor_ie_data_t
Vendor Information Element header.

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

Public Members

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

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

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

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

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

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

Public Members

signed **rssi**
signal intensity of packet

unsigned **rate**
data rate

unsigned **is_group**
usually not used

unsigned **__pad0__**
reserve

unsigned **sig_mode**
0:is not 11n packet; 1:is 11n packet

unsigned **legacy_length**
Length of 11bg mode packet

unsigned **damatch0**
usually not used

unsigned **damatch1**
usually not used

unsigned **bssidmatch0**
usually not used

unsigned **bssidmatch1**
usually not used

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

unsigned **cwb**
if is 11n packet, shows if is HT40 packet or not

unsigned **HT_length**
Length of 11n mode packet

unsigned **smoothing**
reserve

unsigned **not_sounding**
reserve

unsigned **__pad1__**
reserve

unsigned **aggregation**
Aggregation

unsigned **stbc**
STBC

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

unsigned **sgi**
SGI

unsigned **rxend_state**
usually not used

unsigned **ampdu_cnt**
ampdu cnt

unsigned **channel**
which channel this packet in

unsigned **__pad2__**
reserve

signed **noise_floor**
usually not used

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

Public Members

wifi_pkt_rx_ctrl_t **rx_ctrl**
metadata header

uint8_t **payload**[0]
Data or management frame payload. Length of payload is min(112, (pkt->rx_ctrl.sig_mode ? pkt->rx_ctrl.HT_length : pkt->rx_ctrl.legacy_length)) Type of content determined by packet type argument of callback.

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

Public Members

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

struct wifi_tx_status_t
WIFI hardware TX status.

Public Members

unsigned **wifi_tx_result**
TX status code, descripted by "wifi_tx_result_t"

unsigned **wifi_tx_src**
TX status SRC

unsigned **wifi_tx_lrc**
TX status LRC

unsigned **wifi_tx_rate**
TX rate, descripted by "wifi_tx_rate_t"

unsigned **unused**
Resolved

struct wifi_event_sta_scan_done_t
Argument structure for WIFI_EVENT_SCAN_DONE event

Public Members

uint32_t **status**
status of scanning APs: 0 — success, 1 - failure

uint8_t **number**
number of scan results

uint8_t **scan_id**
scan sequence number, used for block scan

struct wifi_event_sta_connected_t
Argument structure for WIFI_EVENT_STA_CONNECTED event

Public Members

uint8_t **ssid**[32]
SSID of connected AP

uint8_t **ssid_len**
SSID length of connected AP

uint8_t **bssid**[6]
BSSID of connected AP

uint8_t **channel**
channel of connected AP

wifi_auth_mode_t **authmode**
authentication mode used by AP

struct wifi_event_sta_authmode_change_t
Argument structure for WIFI_EVENT_STA_AUTHMODE_CHANGE event

Public Members

wifi_auth_mode_t **old_mode**
the old auth mode of AP

wifi_auth_mode_t **new_mode**
the new auth mode of AP

struct wifi_event_sta_wps_er_pin_t
Argument structure for WIFI_EVENT_STA_WPS_ER_PIN event

Public Members

uint8_t **pin_code**[8]
PIN code of station in enrollee mode

struct wifi_event_ap_staconnected_t
Argument structure for WIFI_EVENT_AP_STACONNECTED event

Public Members

uint8_t **mac**[6]
MAC address of the station connected to soft-AP

uint8_t **aid**
the aid that soft-AP gives to the station connected to

struct wifi_event_ap_stadisconnected_t
Argument structure for WIFI_EVENT_AP_STADISCONNECTED event

Public Members

uint8_t **mac**[6]
MAC address of the station disconnects to soft-AP

uint8_t **aid**
the aid that soft-AP gave to the station disconnects to

struct wifi_event_ap_probe_req_rx_t
Argument structure for WIFI_EVENT_AP_PROBEREQRECVED event

Public Members

int **rssi**
Received probe request signal strength

uint8_t **mac**[6]
MAC address of the station which send probe request

struct wifi_event_sta_disconnected_t
Argument structure for WIFI_EVENT_STA_DISCONNECTED event

Public Members

uint8_t **ssid**[32]
SSID of disconnected AP

uint8_t **ssid_len**
SSID length of disconnected AP

uint8_t **bssid**[6]
BSSID of disconnected AP

uint8_t **reason**
reason of disconnection

struct wifi_event_bss_rssi_low_t
Argument structure for WIFI_EVENT_STA_BSS_RSSI_LOW event

Public Members

int32_t **rssi**
RSSI value of bss

Macros

WIFI_IF_STA

WIFI_IF_AP

WIFI_PS_MODEM

WIFI_PROTOCOL_11B

WIFI_PROTOCOL_11G

WIFI_PROTOCOL_11N

WIFI_PROTOCOL_LR

ESP_WIFI_MAX_CONN_NUM
max number of stations which can connect to ESP8266 soft-AP

WIFI_VENDOR_IE_ELEMENT_ID

WIFI_PROMIS_FILTER_MASK_ALL
filter all packets

WIFI_PROMIS_FILTER_MASK_MGMT
filter the packets with type of WIFI_PKT_MGMT

WIFI_PROMIS_FILTER_MASK_CTRL
filter the packets with type of WIFI_PKT_CTRL

WIFI_PROMIS_FILTER_MASK_DATA
filter the packets with type of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_MISC
filter the packets with type of WIFI_PKT_MISC

WIFI_PROMIS_CTRL_FILTER_MASK_ALL
filter all control packets

WIFI_PROMIS_CTRL_FILTER_MASK_WRAPPER
filter the control packets with subtype of Control Wrapper

WIFI_PROMIS_CTRL_FILTER_MASK_BAR
filter the control packets with subtype of Block Ack Request

WIFI_PROMIS_CTRL_FILTER_MASK_BA
filter the control packets with subtype of Block Ack

WIFI_PROMIS_CTRL_FILTER_MASK_PSPOLL
filter the control packets with subtype of PS-Poll

WIFI_PROMIS_CTRL_FILTER_MASK_RTS
filter the control packets with subtype of RTS

WIFI_PROMIS_CTRL_FILTER_MASK_CTS
filter the control packets with subtype of CTS

WIFI_PROMIS_CTRL_FILTER_MASK_ACK
filter the control packets with subtype of ACK

WIFI_PROMIS_CTRL_FILTER_MASK_CFEND
filter the control packets with subtype of CF-END

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

WIFI_EVENT_MASK_ALL
mask all WiFi events

WIFI_EVENT_MASK_NONE
mask none of the WiFi events

WIFI_EVENT_MASK_AP_PROBEREQRECVED
mask SYSTEM_EVENT_AP_PROBEREQRECVED event

MAX_SSID_LEN

MAX_PASSPHRASE_LEN

MAX_WPS_AP_CRED

Type Definitions

typedef esp_interface_t **wifi_interface_t**

Enumerations

enum wifi_mode_t
Values:

WIFI_MODE_NULL = 0
null mode

WIFI_MODE_STA
WiFi station mode

WIFI_MODE_AP
WiFi soft-AP mode

WIFI_MODE_APSTA
WiFi station + soft-AP mode

WIFI_MODE_MAX

enum wifi_country_policy_t
Values:

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

WIFI_COUNTRY_POLICY_MANUAL
Country policy is manual, always use the configured country info

enum wifi_auth_mode_t
Values:

WIFI_AUTH_OPEN = 0
authenticate mode : open

WIFI_AUTH_WEP
authenticate mode : WEP

WIFI_AUTH_WPA_PSK
authenticate mode : WPA_PSK

WIFI_AUTH_WPA2_PSK
authenticate mode : WPA2_PSK

WIFI_AUTH_WPA_WPA2_PSK
authenticate mode : WPA_WPA2_PSK

WIFI_AUTH_WPA2_ENTERPRISE
authenticate mode : WPA2_ENTERPRISE

WIFI_AUTH_WPA3_PSK
authenticate mode : WPA3_PSK

WIFI_AUTH_WPA2_WPA3_PSK
authenticate mode : WPA2_WPA3_PSK

WIFI_AUTH_MAX

enum wifi_err_reason_t
Values:

WIFI_REASON_UNSPECIFIED = 1

WIFI_REASON_AUTH_EXPIRE = 2

WIFI_REASON_AUTH_LEAVE = 3

WIFI_REASON_ASSOC_EXPIRE = 4

WIFI_REASON_ASSOC_TOOMANY = 5

WIFI_REASON_NOT_AUTHED = 6

WIFI_REASON_NOT_ASSOCED = 7

WIFI_REASON_ASSOC_LEAVE = 8

WIFI_REASON_ASSOC_NOT_AUTHED = 9

WIFI_REASON_DISASSOC_PWRCAP_BAD = 10

WIFI_REASON_DISASSOC_SUPCHAN_BAD = 11

WIFI_REASON_IE_INVALID = 13

WIFI_REASON_MIC_FAILURE = 14

WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT = 15

WIFI_REASON_GROUP_KEY_UPDATE_TIMEOUT = 16

WIFI_REASON_IE_IN_4WAY_DIFFERS = 17

WIFI_REASON_GROUP_CIPHER_INVALID = 18

WIFI_REASON_PAIRWISE_CIPHER_INVALID = 19

WIFI_REASON_AKMP_INVALID = 20

WIFI_REASON_UNSUPP_RSN_IE_VERSION = 21

WIFI_REASON_INVALID_RSN_IE_CAP = 22

WIFI_REASON_802_1X_AUTH_FAILED = 23

WIFI_REASON_CIPHER_SUITE_REJECTED = 24

WIFI_REASON_INVALID_PMKID = 53

WIFI_REASON_BEACON_TIMEOUT = 200

WIFI_REASON_NO_AP_FOUND = 201

WIFI_REASON_AUTH_FAIL = 202

WIFI_REASON_ASSOC_FAIL = 203

WIFI_REASON_HANDSHAKE_TIMEOUT = 204

WIFI_REASON_CONNECTION_FAIL = 205

WIFI_REASON_AP_TSF_RESET = 206

WIFI_REASON_BASIC_RATE_NOT_SUPPORT = 207

enum wifi_second_chan_t

Values:

WIFI_SECOND_CHAN_NONE = 0
the channel width is HT20

WIFI_SECOND_CHAN_ABOVE
the channel width is HT40 and the second channel is above the primary channel

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

enum wifi_scan_type_t

Values:

WIFI_SCAN_TYPE_ACTIVE = 0
active scan

WIFI_SCAN_TYPE_PASSIVE
passive scan

enum wifi_cipher_type_t

Values:

WIFI_CIPHER_TYPE_NONE = 0
the cipher type is none

WIFI_CIPHER_TYPE_WEP40
the cipher type is WEP40

WIFI_CIPHER_TYPE_WEP104
the cipher type is WEP104

WIFI_CIPHER_TYPE_TKIP
the cipher type is TKIP

WIFI_CIPHER_TYPE_CCMP
the cipher type is CCMP

WIFI_CIPHER_TYPE_TKIP_CCMP
the cipher type is TKIP and CCMP

WIFI_CIPHER_TYPE_AES_CMAC128
the cipher type is AES-CMAC-128

WIFI_CIPHER_TYPE_UNKNOWN
the cipher type is unknown

enum wifi_ant_t
Values:

WIFI_ANT_ANT0
WiFi antenna 0

WIFI_ANT_ANT1
WiFi antenna 1

WIFI_ANT_MAX
Invalid WiFi antenna

enum wifi_scan_method_t
Values:

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

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

enum wifi_sort_method_t
Values:

WIFI_CONNECT_AP_BY_SIGNAL = 0
Sort match AP in scan list by RSSI

WIFI_CONNECT_AP_BY_SECURITY
Sort match AP in scan list by security mode

enum wifi_state_t
Values:

WIFI_STATE_DEINIT =0

WIFI_STATE_INIT

WIFI_STATE_START

enum wifi_ps_type_t
Values:

WIFI_PS_NONE
No power save

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

WIFI_PS_MAX_MODEM
Maximum modem power saving. In this mode, interval to receive beacons is determined by the listen_interval parameter in *wifi_sta_config_t*. Attention: Using this option may cause ping failures. Not recommended

enum wifi_bandwidth_t

Values:

WIFI_BW_HT20 = 1

WIFI_BW_HT40

enum wifi_storage_t

Values:

WIFI_STORAGE_FLASH
all configuration will strore in both memory and flash

WIFI_STORAGE_RAM
all configuration will only store in the memory

enum wifi_vendor_ie_type_t

Vendor Information Element type.

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

Values:

WIFI_VND_IE_TYPE_BEACON

WIFI_VND_IE_TYPE_PROBE_REQ

WIFI_VND_IE_TYPE_PROBE_RESP

WIFI_VND_IE_TYPE_ASSOC_REQ

WIFI_VND_IE_TYPE_ASSOC_RESP

enum wifi_vendor_ie_id_t

Vendor Information Element index.

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

Values:

WIFI_VND_IE_ID_0

WIFI_VND_IE_ID_1

enum wifi_promiscuous_pkt_type_t

Promiscuous frame type.

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

Values:

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

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

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

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

enum wifi_tx_result_t

WIFI hardware TX result code.

Values:

TX_STATUS_SUCCESS = 1

TX_STATUS_SRC_EXCEED

TX_STATUS_LRC_EXCEED

TX_STATUS_DISCARD

enum wifi_tx_rate_t

WIFI hardware TX rate.

Values:

PHY_RATE_1_LONG

PHY_RATE_2_LONG

PHY_RATE_5_LONG

PHY_RATE_11_LONG

PHY_RATE_RESERVED

PHY_RATE_2_SHORT

PHY_RATE_5_SHORT

PHY_RATE_11_SHORT

PHY_RATE_48

PHY_RATE_24

PHY_RATE_12

PHY_RATE_6

PHY_RATE_54

PHY_RATE_36

PHY_RATE_18

PHY_RATE_9

enum wifi_event_t

WiFi event declarations

Values:

WIFI_EVENT_WIFI_READY = 0

WiFi ready

WIFI_EVENT_SCAN_DONE

finish scanning AP

WIFI_EVENT_STA_START

station start

WIFI_EVENT_STA_STOP

station stop

WIFI_EVENT_STA_CONNECTED

station connected to AP

WIFI_EVENT_STA_DISCONNECTED

station disconnected from AP

WIFI_EVENT_STA_AUTHMODE_CHANGE
the auth mode of AP connected by station changed

WIFI_EVENT_STA_BSS_RSSI_LOW
AP's RSSI crossed configured threshold

WIFI_EVENT_STA_WPS_ER_SUCCESS
station wps succeeds in enrollee mode

WIFI_EVENT_STA_WPS_ER_FAILED
station wps fails in enrollee mode

WIFI_EVENT_STA_WPS_ER_TIMEOUT
station wps timeout in enrollee mode

WIFI_EVENT_STA_WPS_ER_PIN
station wps pin code in enrollee mode

WIFI_EVENT_AP_START
soft-AP start

WIFI_EVENT_AP_STOP
soft-AP stop

WIFI_EVENT_AP_STACONNECTED
a station connected to soft-AP

WIFI_EVENT_AP_STADISCONNECTED
a station disconnected from soft-AP

WIFI_EVENT_AP_PROBEREQRECVED
Receive probe request packet in soft-AP interface

enum wifi_event_sta_wps_fail_reason_t
Argument structure for WIFI_EVENT_STA_WPS_ER_FAILED event

Values:

WPS_FAIL_REASON_NORMAL = 0
WPS normal fail reason

WPS_FAIL_REASON_RECV_M2D
WPS receive M2D frame

WPS_FAIL_REASON_MAX

2.2.2 Smart Config

API Reference

Header File

- esp8266/include/esp_smartconfig.h

Functions

const char ***esp_smartconfig_get_version**(void)
Get the version of SmartConfig.

Return

- SmartConfig version const char.

esp_err_t **esp_smartconfig_start**(**const** *smartconfig_start_config_t* *config)

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

Attention 1. This API can be called in station mode.

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

Return

- ESP_OK: succeed
- others: fail

Parameters

- `config`: pointer to smartconfig start configure structure

esp_err_t **esp_smartconfig_stop**(void)

Stop SmartConfig, free the buffer taken by esp_smartconfig_start.

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

Return

- ESP_OK: succeed
- others: fail

esp_err_t **esp_esptouch_set_timeout**(uint8_t *time_s*)

Set timeout of SmartConfig process.

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

Return

- ESP_OK: succeed
- others: fail

Parameters

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

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

Set protocol type of SmartConfig.

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

Return

- ESP_OK: succeed
- others: fail

Parameters

- `type`: Choose from the smartconfig_type_t.

esp_err_t **esp_smartconfig_fast_mode**(bool *enable*)
Set mode of SmartConfig. default normal mode.

Attention 1. Please call it before API esp_smartconfig_start.

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

Attention 3. Two mode is compatible.

Return

- ESP_OK: succeed
- others: fail

Parameters

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

esp_err_t **esp_smartconfig_get_rvd_data**(uint8_t **rvd_data*, uint8_t *len*)
Get reserved data of ESPTouch_v2.

Return

- ESP_OK: succeed
- others: fail

Parameters

- `rvd_data`: reserved data
- `len`: length of reserved data

Structures

struct smartconfig_event_got_ssid_pswd_t
Argument structure for SC_EVENT_GOT_SSID_PSWD event

Public Members

uint8_t **ssid**[32]
SSID of the AP. Null terminated string.

uint8_t **password**[64]
Password of the AP. Null terminated string.

bool **bssid_set**
whether set MAC address of target AP or not.

uint8_t **bssid**[6]
MAC address of target AP.

smartconfig_type_t **type**
Type of smartconfig(ESPTouch or AirKiss).

uint8_t **token**
Token from cellphone which is used to send ACK to cellphone.

uint8_t **cellphone_ip**[4]
IP address of cellphone.

struct smartconfig_start_config_t
Configure structure for esp_smartconfig_start

Public Members

bool **enable_log**
Enable smartconfig logs.

bool **esp_touch_v2_enable_crypt**
Enable ESPTOUCH V2 crypt.

char ***esp_touch_v2_key**
ESPTOUCH V2 crypt key, len should be 16.

Macros

SMARTCONFIG_START_CONFIG_DEFAULT()

Enumerations

enum smartconfig_type_t
Values:

SC_TYPE_ESPTOUCH = 0
protocol: ESPTouch

SC_TYPE_AIRKISS
protocol: AirKiss

SC_TYPE_ESPTOUCH_AIRKISS
protocol: ESPTouch and AirKiss

SC_TYPE_ESPTOUCH_V2
protocol: ESPTouch V2

enum smartconfig_event_t
Smartconfig event declarations

Values:

SC_EVENT_SCAN_DONE
ESP32 station smartconfig has finished to scan for APs

SC_EVENT_FOUND_CHANNEL
ESP32 station smartconfig has found the channel of the target AP

SC_EVENT_GOT_SSID_PSWD
ESP32 station smartconfig got the SSID and password

SC_EVENT_SEND_ACK_DONE
ESP32 station smartconfig has sent ACK to cellphone

Example code for this API section is provided in wifi directory of SDK examples.

2.3 TCP-IP API

2.3.1 TCPIP Adapter

API Reference

Header File

- tcpip_adapter/include/tcpip_adapter.h

Functions

void **tcpip_adapter_init**(void)
Initialize tcpip adapter.

This will initialize TCPIP stack inside.

esp_err_t **tcpip_adapter_start**(*tcpip_adapter_if_t* *tcpip_if*, uint8_t **mac*, *tcpip_adapter_ip_info_t* **ip_info*)
Start the Wi-Fi station/AP interface with specific MAC and IP.

Station/AP interface will be initialized, connect WiFi stack with TCPIP stack.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS ESP_ERR_NO_MEM

Parameters

- `tcpip_if`: Station/AP interface
- `mac`: set MAC address of this interface
- `ip_info`: set IP address of this interface

esp_err_t **tcpip_adapter_stop**(*tcpip_adapter_if_t tcpip_if*)
Stop an interface.

The interface will be cleanup in this API, if DHCP server/client are started, will be stopped.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY

Parameters

- `tcpip_if`: the interface which will be started

esp_err_t **tcpip_adapter_up**(*tcpip_adapter_if_t tcpip_if*)
Bring up an interface.

Only station interface need to be brought up, since station interface will be shut down when disconnect.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY

Parameters

- `tcpip_if`: the interface which will be up

esp_err_t **tcpip_adapter_down**(*tcpip_adapter_if_t tcpip_if*)
Shut down an interface.

Only station interface need to be shut down, since station interface will be brought up when connect.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY

Parameters

- `tcpip_if`: the interface which will be down

esp_err_t **tcpip_adapter_get_ip_info**(*tcpip_adapter_if_t tcpip_if*, *tcpip_adapter_ip_info_t *ip_info*)

Get interface's IP information.

There has an IP information copy in adapter library, if interface is up, get IP information from interface, otherwise get from copy.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS

Parameters

- `tcpip_if`: the interface which we want to get IP information
- `ip_info`: If successful, IP information will be returned in this argument.

esp_err_t **tcpip_adapter_set_ip_info**(*tcpip_adapter_if_t tcpip_if*, *tcpip_adapter_ip_info_t *ip_info*)

Set interface's IP information.

There has an IP information copy in adapter library, if interface is up, also set interface's IP. DHCP client/server should be stopped before set new IP information.

This function is mainly used for setting static IP.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS

Parameters

- `tcpip_if`: the interface which we want to set IP information
- `ip_info`: store the IP information which needs to be set to specified interface

esp_err_t **tcpip_adapter_set_dns_info**(*tcpip_adapter_if_t tcpip_if*, *tcpip_adapter_dns_type_t type*, *tcpip_adapter_dns_info_t *dns*)

Set DNS Server's information.

There has an DNS Server information copy in adapter library, set DNS Server for appointed interface and type.

1.In station mode, if dhcp client is enabled, then only the fallback DNS server can be set(TCPIP_ADAPTER_DNS_FALLBACK). Fallback DNS server is only used if no DNS servers are set via DHCP. If dhcp client is disabled, then need to set main/backup dns server(TCPIP_ADAPTER_DNS_MAIN, TCPIP_ADAPTER_DNS_BACKUP).

2.In soft-AP mode, the DNS Server's main dns server offered to the station is the IP address of soft-AP, if the application don't want to use the IP address of soft-AP, they can set the main dns server.

This function is mainly used for setting static or Fallback DNS Server.

Return

- ESP_OK on success
- ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS invalid params

Parameters

- `tcpip_if`: the interface which we want to set DNS Server information

- `type`: the type of DNS Server,including TCPIP_ADAPTER_DNS_MAIN, TCPIP_ADAPTER_DNS_BACKUP, TCPIP_ADAPTER_DNS_FALLBACK
- `dns`: the DNS Server address to be set

esp_err_t **tcpip_adapter_get_dns_info**(*tcpip_adapter_if_t tcpip_if*, *tcpip_adapter_dns_type_t type*, *tcpip_adapter_dns_info_t *dns*)

Get DNS Server's information.

When set the DNS Server information successfully, can get the DNS Server's information via the appointed tcpip_if and type

This function is mainly used for getting DNS Server information.

Return

- ESP_OK on success
- ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS invalid params

Parameters

- `tcpip_if`: the interface which we want to get DNS Server information
- `type`: the type of DNS Server,including TCPIP_ADAPTER_DNS_MAIN, TCPIP_ADAPTER_DNS_BACKUP, TCPIP_ADAPTER_DNS_FALLBACK
- `dns`: the DNS Server address to be get

esp_err_t **tcpip_adapter_get_old_ip_info**(*tcpip_adapter_if_t tcpip_if*, *tcpip_adapter_ip_info_t *ip_info*)

Get interface's old IP information.

When the interface successfully gets a valid IP from DHCP server or static configured, a copy of the IP information is set to the old IP information. When IP lost timer expires, the old IP information is reset to 0.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS

Parameters

- `tcpip_if`: the interface which we want to get old IP information
- `ip_info`: If successful, IP information will be returned in this argument.

esp_err_t **tcpip_adapter_set_old_ip_info**(*tcpip_adapter_if_t tcpip_if*, *tcpip_adapter_ip_info_t *ip_info*)

Set interface's old IP information.

When the interface successfully gets a valid IP from DHCP server or static configured, a copy of the IP information is set to the old IP information. When IP lost timer expires, the old IP information is reset to 0.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS

Parameters

- `tcpip_if`: the interface which we want to set old IP information
- `ip_info`: store the IP information which needs to be set to specified interface

esp_err_t **tcpip_adapter_create_ip6_linklocal**(*tcpip_adapter_if_t tcpip_if*)

create interface's linklocal IPv6 information

Note this function will create a linklocal IPv6 address about input interface, if this address status changed to preferred, will call event call back , notify user linklocal IPv6 address has been verified

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS

Parameters

- `tcpip_if`: the interface which we want to set IP information

esp_err_t **tcpip_adapter_dhcps_get_status**(*tcpip_adapter_if_t tcpip_if*, *tcpip_adapter_dhcp_status_t *status*)

Get DHCP server's status.

Return ESP_OK

Parameters

- `tcpip_if`: the interface which we will get status of DHCP server
- `status`: If successful, the status of DHCP server will be return in this argument.

esp_err_t **tcpip_adapter_dhcps_option**(*tcpip_adapter_option_mode_t opt_op*, *tcpip_adapter_option_id_t opt_id*, void *opt_val, uint32_t *opt_len*)

Set or Get DHCP server's option.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_ST
ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STARTED

Parameters

- `opt_op`: option operate type, 1 for SET, 2 for GET.
- `opt_id`: option index, 32 for ROUTER, 50 for IP POLL, 51 for LEASE TIME, 52 for REQUEST TIME
- `opt_val`: option parameter
- `opt_len`: option length

esp_err_t **tcpip_adapter_dhcps_start**(*tcpip_adapter_if_t tcpip_if*)

Start DHCP server.

Note Currently DHCP server is bind to softAP interface.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_ST

Parameters

- `tcpip_if`: the interface which we will start DHCP server

esp_err_t **tcpip_adapter_dhcps_stop**(*tcpip_adapter_if_t tcpip_if*)

Stop DHCP server.

Note Currently DHCP server is bind to softAP interface.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_ST
ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY

Parameters

- `tcpip_if`: the interface which we will stop DHCP server

esp_err_t **tcpip_adapter_dhcpc_get_status**(*tcpip_adapter_if_t tcpip_if*, *tcpip_adapter_dhcp_status_t* **status*)

Get DHCP client status.

Return ESP_OK

Parameters

- `tcpip_if`: the interface which we will get status of DHCP client
- `status`: If successful, the status of DHCP client will be return in this argument.

esp_err_t **tcpip_adapter_dhcpc_option**(*tcpip_adapter_option_mode_t opt_op*, *tcpip_adapter_option_id_t opt_id*, void **opt_val*, uint32_t *opt_len*)

Set or Get DHCP client's option.

Note This function is not implement now.

Return ESP_OK

Parameters

- `opt_op`: option operate type, 1 for SET, 2 for GET.
- `opt_id`: option index, 32 for ROUTER, 50 for IP POLL, 51 for LEASE TIME, 52 for REQUEST TIME
- `opt_val`: option parameter
- `opt_len`: option length

esp_err_t **tcpip_adapter_dhcpc_start**(*tcpip_adapter_if_t tcpip_if*)

Start DHCP client.

Note Currently DHCP client is bind to station interface.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STA ESP_ERR_TCPIP_ADAPTER_DHCPC_START_FAILED

Parameters

- `tcpip_if`: the interface which we will start DHCP client

esp_err_t **tcpip_adapter_dhcpc_stop**(*tcpip_adapter_if_t tcpip_if*)

Stop DHCP client.

Note Currently DHCP client is bind to station interface.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STO ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY

Parameters

- `tcpip_if`: the interface which we will stop DHCP client

esp_err_t **tcpip_adapter_eth_input**(void **buffer*, uint16_t *len*, void **eb*)

esp_err_t **tcpip_adapter_sta_input**(void **buffer*, uint16_t *len*, void **eb*)

Get data from station interface.

This function should be installed by esp_wifi_reg_rxcb, so WiFi packets will be forward to TCPIP stack.

Return ESP_OK

Parameters

- `buffer`: the received data point
- `len`: the received data length
- `eb`: parameter

esp_err_t **tcpip_adapter_ap_input**(void *_buffer_, uint16_t _len_, void *_eb_)
Get data from softAP interface.

This function should be installed by esp_wifi_reg_rxcb, so WiFi packets will be forward to TCPIP stack.

Return ESP_OK

Parameters

- `buffer`: the received data point
- `len`: the received data length
- `eb`: parameter

esp_interface_t **tcpip_adapter_get_esp_if**(void *_dev_)
Get WiFi interface index.

Get WiFi interface from TCPIP interface struct pointer.

Return ESP_IF_WIFI_STA ESP_IF_WIFI_AP ESP_IF_ETH ESP_IF_MAX

Parameters

- `dev`: adapter interface

esp_err_t **tcpip_adapter_get_sta_list**(*wifi_sta_list_t* *_wifi_sta_list_, *tcpip_adapter_sta_list_t* *_tcpip_sta_list_)
Get the station information list.

Return ESP_OK ESP_ERR_TCPIP_ADAPTER_NO_MEM ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS

Parameters

- `wifi_sta_list`: station list info
- `tcpip_sta_list`: station list info

esp_err_t **tcpip_adapter_set_hostname**(*tcpip_adapter_if_t* _tcpip_if_, **const** char *_hostname_)
Set the hostname to the interface.

Return ESP_OK:success ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY:interface status error ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS:parameter error

Parameters

- `tcpip_if`: the interface which we will set the hostname
- `hostname`: the host name for set the interface, the max length of hostname is 32 bytes

esp_err_t **tcpip_adapter_get_hostname**(*tcpip_adapter_if_t* _tcpip_if_, **const** char **_hostname_)
Get the hostname from the interface.

Return ESP_OK:success ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY:interface status error ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS:parameter error

Parameters

- `tcpip_if`: the interface which we will get the hostname
- `hostname`: the host name from the interface

esp_err_t **tcpip_adapter_get_netif** (*tcpip_adapter_if_t tcpip_if*, void ***netif*)
Get the LwIP netif* that is assigned to the interface.

Return ESP_OK:success ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY:interface status error ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS:parameter error

Parameters

- `tcpip_if`: the interface which we will get the hostname
- `netif`: pointer to fill the resulting interface

bool **tcpip_adapter_is_netif_up** (*tcpip_adapter_if_t tcpip_if*)
Test if supplied interface is up or down.

Return true: tcpip_if is UP false: tcpip_if id DOWN

Parameters

- `tcpip_if`: the interface which we will get the hostname

esp_err_t **tcpip_adapter_set_default_wifi_handlers** ()
Install default event handlers for Wi-Fi interfaces (station and AP)

Return

- ESP_OK on success
- one of the errors from esp_event on failure

esp_err_t **tcpip_adapter_clear_default_wifi_handlers** ()
Uninstall default event handlers for Wi-Fi interfaces (station and AP)

Return

- ESP_OK on success
- one of the errors from esp_event on failure

int **tcpip_adapter_get_netif_index** (*tcpip_adapter_if_t tcpip_if*)
Search nefit index through netif interface.

Return

- netif_index on success
- -1 if an invalid parameter is supplied

Parameters

- `tcpip_if`: Interface to search for netif index

Structures

struct tcpip_adapter_ip_info_t
TCP-IP adapter IPV4 address information.

Public Members

ip4_addr_t **ip**
TCP-IP adatpter IPV4 addresss

ip4_addr_t **netmask**
TCP-IP adatpter IPV4 netmask

ip4_addr_t **gw**
TCP-IP adatpter IPV4 gateway

struct tcpip_adapter_ip6_info_t
TCP-IP adapter IPV6 address information if disable IPV6 of LwIP.

Public Members

uint32_t **addr**[4]
TCP-IP adatpter IPV4 addresss data

struct *tcpip_adapter_ip6_info_t*::**[anonymous] ip**
TCP-IP adatpter IPV4 addresss

struct tcpip_adapter_sta_info_t
TCP-IP adapter station information.

Public Members

uint8_t **mac**[6]
TCP-IP adatpter station MAC address

ip4_addr_t **ip**
TCP-IP adatpter station IPV4 addresss

struct tcpip_adapter_sta_list_t
TCP-IP adapter station information table.

Public Members

tcpip_adapter_sta_info_t **sta**[**ESP_WIFI_MAX_CONN_NUM**]
adapter station information array

int **num**
adapter station information number

struct tcpip_adapter_dns_info_t
TCP-IP adapter DNS server information.

Public Members

ip_addr_t **ip**
DNS IP addresss

struct ip_event_ap_staipassigned_t
Event structure for IP_EVENT_AP_STAIPASSIGNED event

Public Members

ip4_addr_t **ip**
IP address which was assigned to the station

struct ip_event_got_ip_t
Event structure for IP_EVENT_STA_GOT_IP, IP_EVENT_ETH_GOT_IP events

Public Members

tcpip_adapter_if_t **if_index**
Interface for which the event is received

tcpip_adapter_ip_info_t **ip_info**
IP address, netmask, gatway IP address

bool **ip_changed**
Whether the assigned IP has changed or not

struct ip_event_got_ip6_t
Event structure for IP_EVENT_GOT_IP6 event

Public Members

tcpip_adapter_if_t **if_index**
Interface for which the event is received

tcpip_adapter_ip6_info_t **ip6_info**
IPv6 address of the interface

struct tcpip_adapter_api_msg_s
TCP-IP adapter async messsage.

Public Members

int **type**
TCP-IP adatpter API message type

int **ret**
TCP-IP adatpter API message process result

tcpip_adapter_api_fn **api_fn**
TCP-IP adatpter API message function

tcpip_adapter_if_t **tcpip_if**
TCP-IP adatpter API message interface type

tcpip_adapter_ip_info_t ***ip_info**
TCP-IP adatpter API message IP information

uint8_t ***mac**
TCP-IP adatpter API message MAC address

void ***data**
TCP-IP adatpter API message MAC private data

struct tcpip_adapter_dns_param_s
TCP-IP adapter DNS parameters.

Public Members

tcpip_adapter_dns_type_t **dns_type**
DNS type

tcpip_adapter_dns_info_t ***dns_info**
DNS information

struct tcpip_adatper_ip_lost_timer_s
TCP-IP adapter IP lost checking timer.

Public Members

bool **timer_running**
check if the timer if running

Macros

CONFIG_TCPIP_LWIP
TCPIP adapter library.

The aim of this adapter is to provide an abstract layer upon TCPIP stack. With this layer, switch to other TCPIP stack is possible and easy in ESP8266_RTOS_SDK.

If users want to use other TCPIP stack, all those functions should be implemented by using the specific APIs of that stack.

tcpip_adapter_init should be called in the start of app_main for only once.

Currently most adapter APIs are called in event_default_handlers.c.

We recommend users only use set/get IP APIs, DHCP server/client APIs, get free station list APIs in application side. Other APIs are used in ESP8266_RTOS_SDK internal, otherwise the state maybe wrong.

TODO: ipv6 support will be added, use menuconfig to disable CONFIG_TCPIP_LWIP

CONFIG_DHCP_STA_LIST

TCPIP_ADAPTER_IPV6

IP2STR(ipaddr)

IPSTR

IPV62STR(ipaddr)

IPV6STR

ESP_ERR_TCPIP_ADAPTER_BASE

ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS

ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY

ESP_ERR_TCPIP_ADAPTER_DHCPC_START_FAILED

ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STARTED

ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STOPPED

ESP_ERR_TCPIP_ADAPTER_NO_MEM

ESP_ERR_TCPIP_ADAPTER_DHCP_NOT_STOPPED

TCPIP_ADAPTER_TRHEAD_SAFE

TCPIP_ADAPTER_IPC_LOCAL

TCPIP_ADAPTER_IPC_REMOTE

TCPIP_HOSTNAME_MAX_SIZE

Type Definitions

typedef dhcps_lease_t **tcpip_adapter_dhcps_lease_t**

typedef int (*__tcpip_adapter_api_fn__)(**struct** *tcpip_adapter_api_msg_s* *msg)

typedef struct *tcpip_adapter_api_msg_s* **tcpip_adapter_api_msg_t**
TCP-IP adapter async messsage.

typedef struct *tcpip_adapter_dns_param_s* **tcpip_adapter_dns_param_t**
TCP-IP adapter DNS parameters.

typedef struct *tcpip_adatper_ip_lost_timer_s* **tcpip_adapter_ip_lost_timer_t**
TCP-IP adapter IP lost checking timer.

Enumerations

enum tcpip_adapter_if_t
Values:

TCPIP_ADAPTER_IF_STA = 0
Wi-Fi STA (station) interface

TCPIP_ADAPTER_IF_AP
Wi-Fi soft-AP interface

TCPIP_ADAPTER_IF_ETH
Ethernet interface

TCPIP_ADAPTER_IF_TEST
tcpip stack test interface

TCPIP_ADAPTER_IF_MAX

enum tcpip_adapter_dns_type_t
Values:

TCPIP_ADAPTER_DNS_MAIN = 0

TCPIP_ADAPTER_DNS_BACKUP
DNS main server address

TCPIP_ADAPTER_DNS_FALLBACK
DNS backup server address,for STA only,support soft-AP in future

TCPIP_ADAPTER_DNS_MAX
DNS fallback server address,for STA only Max DNS

enum tcpip_adapter_dhcp_status_t
Values:

TCPIP_ADAPTER_DHCP_INIT = 0
DHCP client/server in initial state

TCPIP_ADAPTER_DHCP_STARTED
DHCP client/server already been started

TCPIP_ADAPTER_DHCP_STOPPED
DHCP client/server already been stopped

TCPIP_ADAPTER_DHCP_STATUS_MAX

enum tcpip_adapter_option_mode_t
Values:

TCPIP_ADAPTER_OP_START = 0

TCPIP_ADAPTER_OP_SET
set option mode

TCPIP_ADAPTER_OP_GET
get option mode

TCPIP_ADAPTER_OP_MAX

enum tcpip_adapter_option_id_t
Values:

TCPIP_ADAPTER_SUBNET_MASK = 1
network mask

TCPIP_ADAPTER_DOMAIN_NAME_SERVER = 6
domain name server

TCPIP_ADAPTER_ROUTER_SOLICITATION_ADDRESS = 32
solicitation router address

TCPIP_ADAPTER_REQUESTED_IP_ADDRESS = 50
request IP address pool

TCPIP_ADAPTER_IP_ADDRESS_LEASE_TIME = 51
request IP address lease time

TCPIP_ADAPTER_IP_REQUEST_RETRY_TIME = 52
request IP address retry counter

enum ip_event_t
IP event declarations

Values:

IP_EVENT_STA_GOT_IP
station got IP from connected AP

IP_EVENT_STA_LOST_IP
station lost IP and the IP is reset to 0

IP_EVENT_AP_STAIPASSIGNED
soft-AP assign an IP to a connected station

IP_EVENT_GOT_IP6
station or ap or ethernet interface v6IP addr is preferred

2.4 System API

2.4.1 Mem alloc

API Reference

Header File

- heap/include/esp_heap_caps.h

Functions

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.

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.

Return Amount of free bytes in the regions

Parameters

- `caps`: Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

void **esp_heap_caps_init_region**(*heap_region_t* **region*, size_t *max_num*)
Initialize regions of memory to the collection of heaps at runtime.

Parameters

- `region`: region table head point
- `max_num`: region table size

void *__heap_caps_malloc**(size_t *size*, uint32_t *caps*, **const** char **file*, size_t *line*)

void __heap_caps_free**(void **ptr*, **const** char **file*, size_t *line*)

void *__heap_caps_calloc**(size_t *count*, size_t *size*, uint32_t *caps*, **const** char **file*, size_t *line*)

void *__heap_caps_realloc(void *mem, size_t newsize, uint32_t caps, const char *file, size_t line)

void *__heap_caps_zalloc(size_t size, uint32_t caps, const char *file, size_t line)

Structures

struct mem_blk
First type memory block.

Public Members

struct *mem_blk* ***prev**
Point to previous memory block.

struct *mem_blk* ***next**
Point to next memory block.

struct heap_region
User region information.

Public Members

void ***start_addr**
Heap region start address.

size_t **total_size**
Heap region total size by byte.

uint32_t **caps**
Heap capacity.

void ***free_blk**
First free memory block.

size_t **free_bytes**
Current free heap size by byte.

size_t **min_free_bytes**
Minimum free heap size by byte ever.

Macros

HEAP_ALIGN(ptr)
Get "HEAP_ALIGN_SIZE" bytes aligned data(HEAP_ALIGN(ptr) >= ptr).

MALLOC_CAP_EXEC
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-bit data accesses.

MALLOC_CAP_DMA
Memory must be able to accessed by DMA.

MALLOC_CAP_INTERNAL
Just for code compatibility.

MALLOC_CAP_SPIRAM
Just for code compatibility.

MEM_HEAD_SIZE
Size of first type memory block.

MEM2_HEAD_SIZE
Size of second type memory block.

heap_caps_malloc(size, caps)
Allocate a chunk of memory which has the given capabilities.

Equivalent semantics to libc malloc(), for capability-aware memory.

In SDK, `malloc(s)` is equivalent to `heap_caps_malloc(s, MALLOC_CAP_32BIT)`.

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

heap_caps_free(ptr)
Free memory previously allocated via heap_caps_(m/c/re/z)alloc().

Equivalent semantics to libc free(), for capability-aware memory.

In SDK, `free(p)` is equivalent to `heap_caps_free(p)`.

Parameters

- `ptr`: Pointer to memory previously returned from heap_caps_(m/c/re/z)alloc(). Can be NULL.

heap_caps_calloc(n, size, caps)
Allocate a chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

Equivalent semantics to libc calloc(), for capability-aware memory.

In IDF, `calloc(c, s)` is equivalent to `heap_caps_calloc(c, s, MALLOC_CAP_32BIT)`.

Return A pointer to the memory allocated on success, NULL on failure

Parameters

- `n`: Number of continuing chunks of memory to allocate
- `size`: Size, in bytes, of a chunk of memory to allocate
- `caps`: Bitwise OR of MALLOC_CAP_* flags indicating the type of memory to be returned

heap_caps_realloc(ptr, size, caps)
Reallocate memory previously allocated via heap_caps_(m/c/re/z)alloc().

Equivalent semantics to libc realloc(), for capability-aware memory.

In SDK, `realloc(p, s)` is equivalent to `heap_caps_realloc(p, s, MALLOC_CAP_32BIT)`.

'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.

heap_caps_zalloc(size, caps)

Allocate a chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

Equivalent semantics to libc calloc(), for capability-aware memory.

In IDF, `calloc(c, s)` is equivalent to `heap_caps_calloc(c, s, MALLOC_CAP_32BIT)`.

Return A pointer to the memory allocated on success, NULL on failure

Parameters

- `size`: Size, in bytes, of a chunk of memory to allocate
- `caps`: Bitwise OR of MALLOC_CAP_* flags indicating the type of memory to be returned

Type Definitions

typedef struct *mem_blk* **mem_blk_t**

First type memory block.

typedef *mem_blk_t* **mem2_blk_t**

Second type memory block.

typedef struct *heap_region* **heap_region_t**

User region information.

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 ESP8266 startup code. Do not call it at other times.

2.4.2 Heap debug

API Reference

Header File

- heap/include/esp_heap_trace.h

2.4.3 Watch dog task

API Reference

Header File

- esp8266/include/esp_task_wdt.h

Functions

esp_err_t **esp_task_wdt_init**(void)
Initialize the Task Watchdog Timer (TWDT)

Return

- ESP_OK: Initialization was successful
- ESP_ERR_NO_MEM: Initialization failed due to lack of memory

Note esp_task_wdt_init() must only be called after the scheduler started

void **esp_task_wdt_reset**(void)
Reset(Feed) the Task Watchdog Timer (TWDT) on behalf of the currently running task.

2.4.4 Log

API Reference

Header File

- log/include/esp_log.h

Functions

putchar_like_t **esp_log_set_putchar**(*putchar_like_t func*)
Set function used to output log entries.

By default, log output goes to UART0. This function can be used to redirect log output to some other destination, such as file or network. Returns the original log handler, which may be necessary to return output to the previous destination.

Return func old Function used for output.

Parameters

- `func`: new Function used for output. Must have same signature as putchar.

uint32_t **esp_log_timestamp**(void)
Function which returns timestamp to be used in log output.

This function is used in expansion of ESP_LOGx macros. In the 2nd stage bootloader, and at early application startup stage this function uses CPU cycle counter as time source. Later when FreeRTOS scheduler start running, it switches to FreeRTOS tick count.

For now, we ignore millisecond counter overflow.

Return timestamp, in milliseconds

uint32_t **esp_log_early_timestamp**(void)
Function which returns timestamp to be used in log output.

This function uses HW cycle counter and does not depend on OS, so it can be safely used after application crash.

Return timestamp, in milliseconds

void **esp_log_write**(*esp_log_level_t level*, **const** char ***tag*, **const** char ***format*, ...)
Write message into the log.

This function is not intended to be used directly. Instead, use one of ESP_LOGE, ESP_LOGW, ESP_LOGI, ESP_LOGD, ESP_LOGV macros.

This function or these macros should not be used from an interrupt.

void **esp_early_log_write**(*esp_log_level_t level*, **const** char ***tag*, **const** char ***format*, ...)
Write message into the log at system startup or critical state.

This function is not intended to be used directly. Instead, use one of ESP_EARLY_LOGE, ESP_EARLY_LOGW, ESP_LEARLY_OGI, ESP_EARLY_LOGD, ESP_EARLY_LOGV macros.

This function or these macros can be used from an interrupt or NMI exception.

Macros

esp_log_level_set(tag, level)

ESP_LOG_BUFFER_HEX_LEVEL(tag, buffer, buff_len, level)
Log a buffer of hex bytes at specified level, separated into 16 bytes each line.

Parameters

- `tag`: description tag
- `buffer`: Pointer to the buffer array
- `buff_len`: length of buffer in bytes
- `level`: level of the log

ESP_LOG_BUFFER_CHAR_LEVEL(tag, buffer, buff_len, level)
Log a buffer of characters at specified level, separated into 16 bytes each line. Buffer should contain only printable characters.

Parameters

- `tag`: description tag
- `buffer`: Pointer to the buffer array
- `buff_len`: length of buffer in bytes
- `level`: level of the log

ESP_LOG_BUFFER_HEXDUMP (tag, buffer, buff_len, level)
Dump a buffer to the log at specified level.

The dump log shows just like the one below:

```
 W (195) log_example: 0x3ffb4280   45 53 50 33 32 20 69 73  20 67 72 65 61 74 2c␣
↪20  |ESP32 is great, |
 W (195) log_example: 0x3ffb4290   77 6f 72 6b 69 6e 67 20  61 6c 6f 6e 67 20 77␣
↪69  |working along wi|
 W (205) log_example: 0x3ffb42a0   74 68 20 74 68 65 20 49  44 46 2e 00          ␣
↪   |th the IDF..|
```

It is highly recommend to use terminals with over 102 text width.

Parameters

- `tag`: description tag
- `buffer`: Pointer to the buffer array
- `buff_len`: length of buffer in bytes
- `level`: level of the log

ESP_LOG_BUFFER_HEX (tag, buffer, buff_len)
Log a buffer of hex bytes at Info level.

See `esp_log_buffer_hex_level`

Parameters

- `tag`: description tag
- `buffer`: Pointer to the buffer array
- `buff_len`: length of buffer in bytes

ESP_LOG_BUFFER_CHAR (tag, buffer, buff_len)
Log a buffer of characters at Info level. Buffer should contain only printable characters.

See `esp_log_buffer_char_level`

Parameters

- `tag`: description tag
- `buffer`: Pointer to the buffer array
- `buff_len`: length of buffer in bytes

ESP_EARLY_LOGE (tag, format, ...)
macro to output logs in startup code, before heap allocator and syscalls have been initialized. log at `ESP_LOG_ERROR` level.

See `printf`,`ESP_LOGE`

ESP_EARLY_LOGW (tag, format, ...)
macro to output logs in startup code at `ESP_LOG_WARN` level.

See `ESP_EARLY_LOGE`,`ESP_LOGE`, `printf`

ESP_EARLY_LOGI (tag, format, ...)
macro to output logs in startup code at ESP_LOG_INFO level.

See ESP_EARLY_LOGE,ESP_LOGE, printf

ESP_EARLY_LOGD (tag, format, ...)
macro to output logs in startup code at ESP_LOG_DEBUG level.

See ESP_EARLY_LOGE,ESP_LOGE, printf

ESP_EARLY_LOGV (tag, format, ...)
macro to output logs in startup code at ESP_LOG_VERBOSE level.

See ESP_EARLY_LOGE,ESP_LOGE, printf

ESP_LOG_EARLY_IMPL (tag, format, log_level, log_tag_letter, ...)

ESP_LOGE (tag, format, ...)

ESP_LOGW (tag, format, ...)

ESP_LOGI (tag, format, ...)

ESP_LOGD (tag, format, ...)

ESP_LOGV (tag, format, ...)

ESP_LOG_LEVEL (level, tag, format, ...)
runtime macro to output logs at a specified level.

See printf

Parameters

- tag: tag of the log, which can be used to change the log level by esp_log_level_set at runtime.
- level: level of the output log.
- format: format of the output log. see printf
- ...: variables to be replaced into the log. see printf

ESP_LOG_LEVEL_LOCAL (level, tag, format, ...)
runtime macro to output logs at a specified level. Also check the level with LOG_LOCAL_LEVEL.

See printf, ESP_LOG_LEVEL

Type Definitions

typedef int (***putchar_like_t**) (int ch)

Enumerations

enum esp_log_level_t
Log level.

Values:

ESP_LOG_NONE = 0
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.

ESP_LOG_MAX

2.4.5 Sleep modes

API Reference

Header File

- esp8266/include/esp_sleep.h

Functions

void **esp_deep_sleep**(uint64_t *time_in_us*)
Enter deep-sleep mode.

The device will automatically wake up after the deep-sleep time set by the users. Upon waking up, the device boots up from user_init.

Attention 1. XPD_DCDC should be connected to EXT_RSTB through 0 ohm resistor in order to support deep-sleep wakeup.

Attention 2. system_deep_sleep(0): there is no wake up timer; in order to wake up, connect a GPIO to pin RST, the chip will wake up by a falling-edge on pin RST

Attention 3. esp_deep_sleep does not shut down WiFi and higher level protocol connections gracefully. Make sure esp_wifi_stop are called to close any connections and deinitialize the peripherals.

Return null

Parameters

- `time_in_us`: deep-sleep time, unit: microsecond

esp_err_t **esp_pm_configure**(**const** void **config*)
Set implementation-specific power management configuration.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the configuration values are not correct

- ESP_ERR_NOT_SUPPORTED if certain combination of values is not supported.

Parameters

- `config`: pointer to implementation-specific configuration structure (e.g. esp_pm_config_esp32)

void **esp_deep_sleep_set_rf_option**(uint8_t *option*)

Call this API before esp_deep_sleep and esp_wifi_init to set the activity after the next deep-sleep wakeup.

If this API is not called, default to be esp_deep_sleep_set_rf_option(1).

Return null

Parameters

- `option`: radio option 0 : Radio calibration after the deep-sleep wakeup is decided by byte 108 of esp_init_data_default.bin (0~127byte). 1 : Radio calibration will be done after the deep-sleep wakeup. This will lead to stronger current. 2 : Radio calibration will not be done after the deep-sleep wakeup. This will lead to weaker current. 4 : Disable radio calibration after the deep-sleep wakeup (the same as modem-sleep). This will lead to the weakest current, but the device can't receive or transmit data after waking up.

void **esp_wifi_fpm_open**(void)

Enable force sleep function.

Attention Force sleep function is disabled by default.

Return null

void **esp_wifi_fpm_close**(void)

Disable force sleep function.

Return null

void **esp_wifi_fpm_do_wakeup**(void)

Wake ESP8266 up from MODEM_SLEEP_T force sleep.

Attention This API can only be called when MODEM_SLEEP_T force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.

Return null

void **esp_wifi_fpm_set_wakeup_cb**(*fpm_wakeup_cb cb*)

Set a callback of waken up from force sleep because of time out.

Attention 1. This API can only be called when force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.

Attention 2. fpm_wakeup_cb_func will be called after system woke up only if the force sleep time out (wifi_fpm_do_sleep and the parameter is not 0xFFFFFFF).

Attention 3. fpm_wakeup_cb_func will not be called if woke up by wifi_fpm_do_wakeup from MODEM_SLEEP_T type force sleep.

Return null

Parameters

- `cb`: callback of waken up

esp_err_t **esp_wifi_fpm_do_sleep**(uint32_t *sleep_time_in_us*)

Force ESP8266 enter sleep mode, and it will wake up automatically when time out.

Attention 1. This API can only be called when force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.

Attention 2. If this API returned 0 means that the configuration is set successfully, but the ESP8266 will not enter sleep mode immediately, it is going to sleep in the system idle task. Please do not call other WiFi related function right after calling this API.

Return ESP_OK, setting succeed;

Return ESP_ERR_WIFI_FPM_MODE, fail to sleep, force sleep function is not enabled.

Return ESP_ERR_WIFI_PM_MODE_OPEN, fail to sleep, Please call esp_wifi_set_ps(WIFI_PS_NONE) first.

Return ESP_ERR_WIFI_MODE, fail to sleep, Please call esp_wifi_set_mode(WIFI_MODE_NULL) first.

Parameters

- `sleep_time_in_us`: sleep time, ESP8266 will wake up automatically when time out. Unit: us. Range: 10000 ~ 268435455(0xFFFFFFF).
  - If sleep_time_in_us is 0xFFFFFFF, the ESP8266 will sleep till
  - if wifi_fpm_set_sleep_type is set to be LIGHT_SLEEP_T, ESP8266 can wake up by GPIO.
  - if wifi_fpm_set_sleep_type is set to be MODEM_SLEEP_T, ESP8266 can wake up by wifi_fpm_do_wakeup.

void **esp_wifi_fpm_set_sleep_type**(*wifi_sleep_type_t type*)

Set sleep type for force sleep function.

Attention This API can only be called before wifi_fpm_open.

Return null

Parameters

- `type`: sleep type

wifi_sleep_type_t **esp_wifi_fpm_get_sleep_type**(void)

Get sleep type of force sleep function.

Return sleep type

void **esp_wifi_enable_gpio_wakeup**(uint32_t *gpio_num*, *gpio_int_type_t intr_status*)

Set a GPIO to wake the ESP8266 up from light-sleep mode ESP8266 will be wakened from Light-sleep, when the GPIO is in low-level.

If the ESP8266 enters light-sleep automatically(esp_wifi_set_sleep_type(LIGHT_SLEEP_T);), after being waken up by GPIO, when the chip attempts to sleep again, it will check the status of the GPIO: Note: • If the GPIO is still in the wakeup status, the EP8266 will enter modem-sleep mode instead; • If the GPIO is NOT in the wakeup status, the ESP8266 will enter light-sleep mode

Return null

Parameters

- `gpio_num`: GPIO number, range: [0, 15]. gpio_int_type_t intr_status: status of GPIO interrupt to trigger the wakeup process.

  - if esp_wifi_fpm_set_sleep_type is set to be LIGHT_SLEEP_T, ESP8266 can wake up by GPIO.
  - if esp_wifi_fpm_set_sleep_type is set to be MODEM_SLEEP_T, ESP8266 can wake up by esp_wifi_fpm_do_wakeup.
- `intr_status`: GPIO interrupt type

void **esp_wifi_disable_gpio_wakeup**(void)
Disable the function that the GPIO can wake the ESP8266 up from light-sleep mode.

esp_err_t **esp_sleep_enable_timer_wakeup**(uint32_t *time_in_us*)
Enable wakeup by timer.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if value is out of range (TBD)

Parameters

- `time_in_us`: time before wakeup, in microseconds

esp_err_t **esp_light_sleep_start**(void)
Enter light sleep with the configured wakeup options.

Attention esp_deep_sleep does not shut down WiFi and higher level protocol connections gracefully. Make sure esp_wifi_stop are called to close any connections and deinitialize the peripherals.

Return

- ESP_OK on success (returned after wakeup)
- ESP_ERR_INVALID_STATE if WiFi is not stopped

void **esp_sleep_start**(void)
Operation system start check time and enter sleep.

Note This function is called by system, user should not call this

esp_err_t **esp_sleep_enable_gpio_wakeup**(void)
Enable wakeup from light sleep using GPIOs.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if wakeup triggers conflict

esp_err_t **esp_sleep_disable_wakeup_source**(*esp_sleep_source_t source*)
Disable wakeup source.

This function is used to deactivate wake up trigger for source defined as parameter of the function.

Note This function does not modify wake up configuration in RTC. It will be performed in esp_sleep_start function.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if trigger was not active

Parameters

- `source`: - number of source to disable of type esp_sleep_source_t

void **esp_power_consumption_info**(bool *clear_old_data*)
Print power consumption information.

Note This function is used to print power consumption data. The current when the RF and CPU are both turned on is 70 mA. The current when only the CPU is turned on is 18 mA. 900uA when both CPU and RF are off. There may be some errors compared to the actual power consumption. The power consumption is based on the actual measurement, and the printing in the function is for reference only.

Return null

Parameters

- `clear_old_data`: - Recalculate power consumption info or not.

Type Definitions

typedef enum *esp_sleep_mode* **esp_sleep_mode_t**

typedef void (***fpm_wakeup_cb**)(void)

Enumerations

enum wifi_sleep_type_t
Values:

WIFI_NONE_SLEEP_T = 0

WIFI_LIGHT_SLEEP_T

WIFI_MODEM_SLEEP_T

enum esp_sleep_mode
Values:

ESP_CPU_WAIT = 0

ESP_CPU_LIGHTSLEEP

enum esp_sleep_source_t
Sleep wakeup cause.

Values:

ESP_SLEEP_WAKEUP_UNDEFINED
In case of deep sleep, reset was not caused by exit from deep sleep.

ESP_SLEEP_WAKEUP_ALL
Not a wakeup cause, used to disable all wakeup sources with esp_sleep_disable_wakeup_source.

ESP_SLEEP_WAKEUP_TIMER
Wakeup caused by timer.

ESP_SLEEP_WAKEUP_GPIO
Wakeup caused by GPIO (light sleep only)

2.4.6 System

API Reference

Header File

- esp8266/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 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 EFUSE or external storage, call this API to set base MAC address with the MAC address which is stored in EFUSE or external storage before initializing WiFi.

Return ESP_OK on success

Parameters

- `mac`: base MAC address, length: 6 bytes.

esp_err_t **esp_base_mac_addr_get**(uint8_t *_mac_)

Return base MAC address which is set using esp_base_mac_addr_set.

Return ESP_OK on success ESP_ERR_INVALID_MAC base MAC address has not been set

Parameters

- `mac`: base MAC address, length: 6 bytes.

esp_err_t **esp_efuse_mac_get_default**(uint8_t *_mac_)

Return base MAC address which is factory-programmed by Espressif in EFUSE.

Return ESP_OK on success

Parameters

- `mac`: base MAC address, length: 6 bytes.

esp_err_t **esp_read_mac**(uint8_t *_mac_, *esp_mac_type_t* _type_)

Read base MAC address and set MAC address of the interface.

This function first get base MAC address using esp_base_mac_addr_get or reads base MAC address from EFUSE. Then set the MAC address of the interface including wifi station and wifi softap.

Return ESP_OK on success

Parameters

- `mac`: MAC address of the interface, length: 6 bytes.
- `type`: type of MAC address, 0:wifi station, 1:wifi softap.

esp_err_t **esp_derive_local_mac**(uint8_t *_local_mac_, **const** uint8_t *_universal_mac_)
Derive local MAC address from universal MAC address.

This function derives a local MAC address from an universal MAC address. A `definition of local vs universal MAC address can be found on Wikipedia <>`. In ESP8266, universal MAC address is generated from base MAC address in EFUSE or other external storage. Local MAC address is derived from the universal MAC address.

Return ESP_OK on success

Parameters

- `local_mac`: Derived local MAC address, length: 6 bytes.
- `universal_mac`: Source universal MAC address, length: 6 bytes.

void **esp_set_cpu_freq**(*esp_cpu_freq_t cpu_freq*)
Switch CPU frequency.

If a PLL-derived frequency is requested (80, 160), this function will enable the PLL. Otherwise, PLL will be disabled. Note: this function is not optimized for switching speed. It may take several hundred microseconds to perform frequency switch.

Parameters

- `cpu_freq`: new CPU frequency

void **system_restore**(void)
Reset to default settings.

void **esp_restart**(void)
Restart CPU.

This function does not return.

esp_reset_reason_t **esp_reset_reason**(void)
Get reason of last reset.

Return See description of esp_reset_reason_t for explanation of each value.

uint32_t **esp_get_free_heap_size**(void)
Get the size of available heap.

Note that the returned value may be larger than the maximum contiguous block which can be allocated.

Return Available heap size, in bytes.

uint32_t **esp_get_minimum_free_heap_size**(void)
Get the minimum heap that has ever been available.

Return Minimum free heap ever available

uint32_t **esp_random**(void)
Get one random 32-bit word from hardware RNG.

Return Random value between 0 and UINT32_MAX

void **esp_fill_random**(void *_buf_, size_t _len_)
Fill a buffer with random bytes from hardware RNG.

Note This function has the same restrictions regarding available entropy as esp_random()

Parameters

- `buf`: Pointer to buffer to fill with random numbers.
- `len`: Length of buffer in bytes

esp_err_t **esp_mac_init**(void)
Initialize MAC address.

Return 0 if sucess or others failed

void **esp_chip_info**(*esp_chip_info_t* **out_info*)
Fill an *esp_chip_info_t* structure with information about the chip.

Parameters

- `out_info`: structure to be filled

flash_size_map **system_get_flash_size_map**(void)
Get the current Flash size and Flash map.

Flash map depends on the selection when compiling, more details in document "2A-ESP8266__IOT_SDK_User_Manual"

Return enum flash_size_map

Structures

struct esp_chip_info_t
The structure represents information about the chip.

Public Members

esp_chip_model_t **model**
chip model, one of esp_chip_model_t

uint32_t **features**
bit mask of CHIP_FEATURE_x feature flags

uint8_t **cores**
number of CPU cores

uint8_t **revision**
chip revision number

Macros

CRYSTAL_USED

CHIP_FEATURE_EMB_FLASH
Chip has embedded flash memory.

Chip feature flags, used in *esp_chip_info_t*

CHIP_FEATURE_WIFI_BGN
Chip has 2.4GHz WiFi.

CHIP_FEATURE_BLE
Chip has Bluetooth LE.

CHIP_FEATURE_BT
Chip has Bluetooth Classic.

Enumerations

enum esp_mac_type_t
Values:

ESP_MAC_WIFI_STA

ESP_MAC_WIFI_SOFTAP

enum esp_reset_reason_t
Reset reasons.

Values:

ESP_RST_UNKNOWN = 0
Reset reason can not be determined.

ESP_RST_POWERON
Reset due to power-on event.

ESP_RST_EXT
Reset by external pin (not applicable for ESP8266)

ESP_RST_SW
Software reset via esp_restart.

ESP_RST_PANIC
Software reset due to exception/panic.

ESP_RST_INT_WDT
Reset (software or hardware) due to interrupt watchdog.

ESP_RST_TASK_WDT
Reset due to task watchdog.

ESP_RST_WDT
Reset due to other watchdogs.

ESP_RST_DEEPSLEEP
Reset after exiting deep sleep mode.

ESP_RST_BROWNOUT
Brownout reset (software or hardware)

ESP_RST_SDIO
Reset over SDIO.

ESP_RST_FAST_SW
Fast reboot.

enum esp_cpu_freq_t
CPU frequency values.

Values:

ESP_CPU_FREQ_80M = 1
80 MHz

ESP_CPU_FREQ_160M = 2
160 MHz

enum flash_size_map
Values:

FLASH_SIZE_4M_MAP_256_256 = 0
Flash size : 4Mbits. Map : 256KBytes + 256KBytes

FLASH_SIZE_2M
Flash size : 2Mbits. Map : 256KBytes

FLASH_SIZE_8M_MAP_512_512
Flash size : 8Mbits. Map : 512KBytes + 512KBytes

FLASH_SIZE_16M_MAP_512_512
Flash size : 16Mbits. Map : 512KBytes + 512KBytes

FLASH_SIZE_32M_MAP_512_512
Flash size : 32Mbits. Map : 512KBytes + 512KBytes

FLASH_SIZE_16M_MAP_1024_1024
Flash size : 16Mbits. Map : 1024KBytes + 1024KBytes

FLASH_SIZE_32M_MAP_1024_1024
Flash size : 32Mbits. Map : 1024KBytes + 1024KBytes

FLASH_SIZE_32M_MAP_2048_2048
attention: don't support now ,just compatible for nodemcu; Flash size : 32Mbits. Map : 2048KBytes + 2048KBytes

FLASH_SIZE_64M_MAP_1024_1024
Flash size : 64Mbits. Map : 1024KBytes + 1024KBytes

FLASH_SIZE_128M_MAP_1024_1024
Flash size : 128Mbits. Map : 1024KBytes + 1024KBytes

FALSH_SIZE_MAP_MAX

enum esp_chip_model_t
Chip models.

Values:

CHIP_ESP8266 = 0
ESP8266.

CHIP_ESP32 = 1
ESP32.

CHAPTER 3

API Guides

3.1 Build System

This document explains the Espressif IoT Development Framework (ESP-IDF) build system and the concept of "components"

Read this document if you want to know how to organise a new ESP8266_RTOS-SDK (ESP-IDF Style) project.

We recommend using the hello_world project at directory of examples/get-started as a starting point for your project.

3.1.1 Using the Build System

The ESP8266_RTOS_SDK README file contains a description of how to use the build system to build your project.

3.1.2 Overview

An ESP8266_RTOS_SDK project can be seen as an amalgamation of a number of components. For example, for a http request example that shows the current humidity, there could be:

- The SoC base libraries (libc, rom bindings etc)
- The WiFi drivers
- A TCP/IP stack
- The FreeRTOS operating system
- Main code tying it all together

ESP8266_RTOS_SDK 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 SDK directories, the project directories and (optionally) in additional custom component directories. It then allows the user to configure the ESP8266_RTOS_SDK project using a a text-based menu system to customize each component. After the components in the project are configured, the build process will compile the project.

Concepts

- A "project" is a directory that contains all the files and configuration to build a single "app" (executable), as well as additional supporting output such as a partition table, data/filesystem partitions, and a bootloader.
- "Project configuration" is held in a single file called sdkconfig in the root directory of the project. This configuration file is modified via `make menuconfig` to customise the configuration of the project. A single project contains exactly one project configuration.
- An "app" is an executable which is built by ESP8266_RTOS_SDK. 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 ESP8266_RTOS_SDK itself, others may be sourced from other places.

Some things are not part of the project:

- "ESP8266_RTOS_SDK" 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 `ESP8266_RTOS_SDK` 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 ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK.

- "main" directory is a special "pseudo-component" that contains source code for the project itself. "main" is a default name, the Makefile variable `COMPONENT_DIRS` includes this component but you can modify this variable (or set `EXTRA_COMPONENT_DIRS`) to look for components in other places.
- "build" directory is where build output is created. After the make process is run, this directory will contain interim object files and libraries as well as final binary output files. This directory is usually not added to source control or distributed with the project source code.

Component directories contain a component makefile - `component.mk`. This may contain variable definitions to control the build process of the component, and its integration into the overall project. See *Component Makefiles* for more details.

Each component may also include a `Kconfig` file defining the *component configuration* options that can be set via the project configuration. Some components may also include `Kconfig.projbuild` and `Makefile.projbuild` files, which are special files for *overriding parts of the project*.

Project Makefiles

Each project has a single Makefile that contains build settings for the entire project. By default, the project Makefile can be quite minimal.

Minimal Example Makefile

```
PROJECT_NAME := myProject

include $(IDF_PATH)/make/project.mk
```

Mandatory Project Variables

- `PROJECT_NAME`: Name of the project. Binary output files will use this name - ie myProject.bin, myProject.elf.

Optional Project Variables

These variables all have default values that can be overridden for custom behaviour. Look in `make/project.mk` for all of the implementation details.

- `PROJECT_PATH`: Top-level project directory. Defaults to the directory containing the Makefile. Many other project variables are based on this variable. The project path cannot contain spaces.
- `BUILD_DIR_BASE`: The build directory for all objects/libraries/binaries. Defaults to `$(PROJECT_PATH)/build`.
- `COMPONENT_DIRS`: Directories to search for components. Defaults to *$(IDF_PATH)/components*, *$(PROJECT_PATH)/components*, `$(PROJECT_PATH)/main` and `EXTRA_COMPONENT_DIRS`. Override this variable if you don't want to search for components in these places.
- `EXTRA_COMPONENT_DIRS`: Optional list of additional directories to search for components.
- `COMPONENTS`: A list of component names to build into the project. Defaults to all components found in the COMPONENT_DIRS directories.
- `EXCLUDE_COMPONENTS`: Optional list of component names to exclude during the build process. Note that this decreases build time, but not binary size.

Any paths in these Makefile variables should be absolute paths. You can convert relative paths using `$(PROJECT_PATH)/xxx`, `$(IDF_PATH)/xxx`, or use the Make function `$(abspath xxx)`.

These variables should all be set before the line `include $(IDF_PATH)/make/project.mk` in the Makefile.

Component Makefiles

Each project contains one or more components, which can either be part of ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK components with a modified version by simply copying the component from the ESP8266_RTOS_SDK component directory to the project component tree and then modifying it there. If used in this way, the ESP8266_RTOS_SDK directory itself can remain untouched.

Minimal Component Makefile

The minimal `component.mk` file is an empty file(!). If the file is empty, the default component behaviour is set:

- All source files in the same directory as the makefile (`*.c`, `*.cpp`, `*.cc`, `*.S`) will be compiled into the component library
- A sub-directory "include" will be added to the global include search path for all other components.
- The component library will be linked into the project app.

See *example component makefiles* for more complete component makefile examples.

Note that there is a difference between an empty `component.mk` file (which invokes default component build behaviour) and no `component.mk` file (which means no default component build behaviour will occur.) It is possible for a component to have no *component.mk* file, if it only contains other files which influence the project configuration or build process.

Preset Component Variables

The following component-specific variables are available for use inside `component.mk`, but should not be modified:

- COMPONENT_PATH: The component directory. Evaluates to the absolute path of the directory containing component.mk. The component path cannot contain spaces.
- COMPONENT_NAME: Name of the component. Defaults to the name of the component directory.
- COMPONENT_BUILD_DIR: The component build directory. Evaluates to the absolute path of a directory inside *$(BUILD_DIR_BASE)* where this component's source files are to be built. This is also the Current Working Directory any time the component is being built, so relative paths in make targets, etc. will be relative to this directory.
- COMPONENT_LIBRARY: Name of the static library file (relative to the component build directory) that will be built for this component. Defaults to $(COMPONENT_NAME).a.

The following variables are set at the project level, but exported for use in the component build:

- PROJECT_NAME: Name of the project, as set in project Makefile
- PROJECT_PATH: Absolute path of the project directory containing the project Makefile.
- COMPONENTS: Name of all components that are included in this build.
- CONFIG_*: Each value in the project configuration has a corresponding variable available in make. All names begin with CONFIG_.
- CC, LD, AR, OBJCOPY: Full paths to each tool from the gcc xtensa cross-toolchain.
- HOSTCC, HOSTLD, HOSTAR: Full names of each tool from the host native toolchain.
- IDF_VER: ESP8266_RTOS_SDK version, retrieved from either $(IDF_PATH)/version.txt file (if present) else using git command git describe. Recommended format here is single liner that specifies major IDF release version, e.g. v2.0 for a tagged release or v2.0-275-g0efaa4f for an arbitrary commit. Application can make use of this by calling esp_get_idf_version().

If you modify any of these variables inside component.mk then this will not prevent other components from building but it may make your component hard to build and/or debug.

Optional Project-Wide Component Variables

The following variables can be set inside component.mk to control build settings across the entire project:

- COMPONENT_ADD_INCLUDEDIRS: Paths, relative to the component directory, which will be added to the include search path for all components in the project. Defaults to include if not overridden. If an include directory is only needed to compile this specific component, add it to COMPONENT_PRIV_INCLUDEDIRS instead.
- COMPONENT_ADD_LDFLAGS: Add linker arguments to the LDFLAGS for the app executable. Defaults to -l$(COMPONENT_NAME). If adding pre-compiled libraries to this directory, add them as absolute paths - ie $(COMPONENT_PATH)/libwhatever.a
- COMPONENT_DEPENDS: Optional list of component names that should be compiled before this component. This is not necessary for link-time dependencies, because all component include directories are available at all times. It is necessary if one component generates an include file which you then want to include in another component. Most components do not need to set this variable.
- COMPONENT_ADD_LINKER_DEPS: Optional list of component-relative paths to files which should trigger a re-link of the ELF file if they change. Typically used for linker script files and binary libraries. Most components do not need to set this variable.

The following variable only works for components that are part of ESP8266_RTOS_SDK itself:

- COMPONENT_SUBMODULES: Optional list of git submodule paths (relative to COMPONENT_PATH) used by the component. These will be checked (and initialised if necessary) by the build process. This variable is ignored if the component is outside the IDF_PATH directory.

Optional Component-Specific Variables

The following variables can be set inside component.mk to control the build of that component:

- COMPONENT_PRIV_INCLUDEDIRS: Directory paths, must be relative to the component directory, which will be added to the include search path for this component's source files only.
- COMPONENT_EXTRA_INCLUDES: Any extra include paths used when compiling the component's source files. These will be prefixed with '-I' and passed as-is to the compiler. Similar to the COMPONENT_PRIV_INCLUDEDIRS variable, except these paths are not expanded relative to the component directory.
- COMPONENT_SRCDIRS: Directory paths, must be relative to the component directory, which will be searched for source files (*.cpp, *.c, *.S). Defaults to '.', ie the component directory itself. Override this to specify a different list of directories which contain source files.
- COMPONENT_OBJS: Object files to compile. Default value is a .o file for each source file that is found in COMPONENT_SRCDIRS. Overriding this list allows you to exclude source files in COMPONENT_SRCDIRS that would otherwise be compiled. See *Specifying source files*
- COMPONENT_EXTRA_CLEAN: Paths, relative to the component build directory, of any files that are generated using custom make rules in the component.mk file and which need to be removed as part of make clean. See *Source Code Generation* for an example.
- COMPONENT_OWNBUILDTARGET & COMPONENT_OWNCLEANTARGET: These targets allow you to fully override the default build behaviour for the component. See *Fully Overriding The Component Makefile* for more details.
- COMPONENT_CONFIG_ONLY: If set, this flag indicates that the component produces no built output at all (ie COMPONENT_LIBRARY is not built), and most other component variables are ignored. This flag is used for IDF internal components which contain only KConfig.projbuild and/or Makefile.projbuild files to configure the project, but no source files.
- CFLAGS: Flags passed to the C compiler. A default set of CFLAGS is defined based on project settings. Component-specific additions can be made via CFLAGS +=. It is also possible (although not recommended) to override this variable completely for a component.
- CPPFLAGS: Flags passed to the C preprocessor (used for .c, .cpp and .S files). A default set of CPPFLAGS is defined based on project settings. Component-specific additions can be made via CPPFLAGS +=. It is also possible (although not recommended) to override this variable completely for a component.
- CXXFLAGS: Flags passed to the C++ compiler. A default set of CXXFLAGS is defined based on project settings. Component-specific additions can be made via CXXFLAGS +=. It is also possible (although not recommended) to override this variable completely for a component.

To apply compilation flags to a single source file, you can add a variable override as a target, ie:

```
apps/dhcpserver.o: CFLAGS += -Wno-unused-variable
```

This can be useful if there is upstream code that emits warnings.

Component Configuration

Each component can also have a Kconfig file, alongside `component.mk`. This contains contains configuration settings to add to the "make menuconfig" for this component.

These settings are found under the "Component Settings" menu when menuconfig is run.

To create a component KConfig file, it is easiest to start with one of the KConfig files distributed with ESP8266_RTOS_SDK.

For an example, see *Adding conditional configuration*.

Preprocessor Definitions

ESP8266_RTOS_SDK build systems adds the following C preprocessor definitions on the command line:

- `ESP_PLATFORM` — Can be used to detect that build happens within ESP8266_RTOS_SDK.
- `IDF_VER` — ESP8266_RTOS_SDK version, see *Preset Component Variables* for more details.

Build Process Internals

Top Level: Project Makefile

- "make" is always run from the project directory and the project makefile, typically named Makefile.
- The project makefile sets `PROJECT_NAME` and optionally customises other *optional project variables*
- The project makefile includes `$(IDF_PATH)/make/project.mk` which contains the project-level Make logic.
- `project.mk` fills in default project-level make variables and includes make variables from the project configuration. If the generated makefile containing project configuration is out of date, then it is regenerated (via targets in `project_config.mk`) and then the make process restarts from the top.
- `project.mk` builds a list of components to build, based on the default component directories or a custom list of components set in *optional project variables*.
- Each component can set some *optional project-wide component variables*. These are included via generated makefiles named `component_project_vars.mk` - there is one per component. These generated makefiles are included into `project.mk`. If any are missing or out of date, they are regenerated (via a recursive make call to the component makefile) and then the make process restarts from the top.
- *Makefile.projbuild* files from components are included into the make process, to add extra targets or configuration.
- By default, the project makefile also generates top-level build & clean targets for each component and sets up *app* and *clean* targets to invoke all of these sub-targets.
- In order to compile each component, a recursive make is performed for the component makefile.

To better understand the project make process, have a read through the `project.mk` file itself.

Second Level: Component Makefiles

- Each call to a component makefile goes via the `$(IDF_PATH)/make/component_wrapper.mk` wrapper makefile.

- This component wrapper includes all component `Makefile.componentbuild` files, making any recipes, variables etc in these files available to every component.
- The `component_wrapper.mk` is called with the current directory set to the component build directory, and the `COMPONENT_MAKEFILE` variable is set to the absolute path to `component.mk`.
- `component_wrapper.mk` sets default values for all *component variables*, then includes the *component.mk* file which can override or modify these.
- If `COMPONENT_OWNBUILDTARGET` and `COMPONENT_OWNCLEANTARGET` are not defined, default build and clean targets are created for the component's source files and the prerequisite `COMPONENT_LIBRARY` static library file.
- The `component_project_vars.mk` file has its own target in `component_wrapper.mk`, which is evaluated from `project.mk` if this file needs to be rebuilt due to changes in the component makefile or the project configuration.

To better understand the component make process, have a read through the `component_wrapper.mk` file and some of the `component.mk` files included with ESP8266_RTOS_SDK.

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 ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK build system:

- Appending `V=1` to the make arguments (or setting it as an environment variable) will cause make to echo all commands executed, and also each directory as it is entered for a sub-make.
- Running `make -w` will cause make to echo each directory as it is entered for a sub-make - same as `V=1` but without also echoing all commands.
- Running `make --trace` (possibly in addition to one of the above arguments) will print out every target as it is built, and the dependency which caused it to be built.
- Running `make -p` prints a (very verbose) summary of every generated target in each makefile.

For more debugging tips and general make information, see the *GNU Make Manual*.

Warning On Undefined Variables

By default, the build process will print a warning if an undefined variable is referenced (like `$(DOES_NOT_EXIST)`). This can be useful to find errors in variable names.

If you don't want this behaviour, it can be disabled in menuconfig's top level menu under *SDK tool configuration*.

Note that this option doesn't trigger a warning if `ifdef` or `ifndef` are used in Makefiles.

Overriding Parts of the Project

Makefile.projbuild

For components that have build requirements that must be evaluated in the top-level project make pass, you can create a file called `Makefile.projbuild` in the component directory. This makefile is included when `project.mk` is evaluated.

For example, if your component needs to add to CFLAGS for the entire project (not just for its own source files) then you can set `CFLAGS +=` in Makefile.projbuild.

`Makefile.projbuild` files are used heavily inside ESP8266_RTOS_SDK, for defining project-wide build features such as `esptool.py` command line arguments and the `bootloader` "special app".

Note that `Makefile.projbuild` isn't necessary for the most common component uses - such as adding include directories to the project, or LDFLAGS to the final linking step. These values can be customised via the `component.mk` file itself. See *Optional Project-Wide Component Variables* for details.

Take care when setting variables or targets in this file. As the values are included into the top-level project makefile pass, they can influence or break functionality across all components!

KConfig.projbuild

This is an equivalent to `Makefile.projbuild` for *component configuration* KConfig files. If you want to include configuration options at the top-level of menuconfig, rather than inside the "Component Configuration" sub-menu, then these can be defined in the KConfig.projbuild file alongside the `component.mk` file.

Take care when adding configuration values in this file, as they will be included across the entire project configuration. Where possible, it's generally better to create a KConfig file for *component configuration*.

Makefile.componentbuild

For components that e.g. include tools to generate source files from other files, it is necessary to be able to add recipes, macros or variable definitions into the component build process of every components. This is done by having a `Makefile.componentbuild` in a component directory. This file gets included in `component_wrapper.mk`, before the `component.mk` of the component is included. As with the Makefile.projbuild, take care with these files: as they're included in each component build, a `Makefile.componentbuild` error may only show up when compiling an entirely different component.

Configuration-Only Components

Some special components which contain no source files, only `Kconfig.projbuild` and `Makefile.projbuild`, may set the flag `COMPONENT_CONFIG_ONLY` in the component.mk file. If this flag is set, most other component variables are ignored and no build step is run for the component.

Example Component Makefiles

Because the build environment tries to set reasonable defaults that will work most of the time, component.mk can be very small or even empty (see *Minimal Component Makefile*). However, overriding *component variables* is usually required for some functionality.

Here are some more advanced examples of `component.mk` makefiles:

Adding source directories

By default, sub-directories are ignored. If your project has sources in sub-directories instead of in the root of the component then you can tell that to the build system by setting COMPONENT_SRCDIRS:

```
COMPONENT_SRCDIRS := src1 src2
```

This will compile all source files in the src1/ and src2/ sub-directories instead.

Specifying source files

The standard component.mk logic adds all .S and .c files in the source directories as sources to be compiled unconditionally. It is possible to circumvent that logic and hard-code the objects to be compiled by manually setting the COMPONENT_OBJS variable to the name of the objects that need to be generated:

```
COMPONENT_OBJS := file1.o file2.o thing/filea.o thing/fileb.o anotherthing/main.o
COMPONENT_SRCDIRS := . thing anotherthing
```

Note that COMPONENT_SRCDIRS must be set as well.

Adding conditional configuration

The configuration system can be used to conditionally compile some files depending on the options selected in make menuconfig. For this, ESP8266_RTOS_SDK 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
```

(continues on next page)

(continued from previous page)

```
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: ESP8266_RTOS_SDK uses the default Make policy of judging a variable which is empty or contains only whitespace to be false while a variable with any non-whitespace in it is true.

(Note: Older versions of this document advised conditionally adding object file names to COMPONENT_OBJS. While this still is possible, this will only work when all object files for a component are named explicitely, and will not clean up deselected object files in a make clean pass.)

Source Code Generation

Some components will have a situation where a source file isn't supplied with the component itself but has to be generated from another file. Say our component has a header file that consists of the converted binary data of a BMP file, converted using a hypothetical tool called bmp2h. The header file is then included in as C source file called graphics_lib.c:

```
COMPONENT_EXTRA_CLEAN := logo.h

graphics_lib.o: logo.h

logo.h: $(COMPONENT_PATH)/logo.bmp
    bmp2h -i $^ -o $@
```

In this example, graphics_lib.o and logo.h will be generated in the current directory (the build directory) while logo.bmp comes with the component and resides under the component path. Because logo.h is a generated file, it needs to be cleaned when make clean is called which why it is added to the COMPONENT_EXTRA_CLEAN variable.

Cosmetic Improvements

Because logo.h is a generated file, it needs to be cleaned when make clean is called which why it is added to the COMPONENT_EXTRA_CLEAN variable.

Adding logo.h to the graphics_lib.o dependencies causes it to be generated before graphics_lib.c is compiled.

If a a source file in another component included `logo.h`, then this component's name would have to be added to the other component's `COMPONENT_DEPENDS` list to ensure that the components were built in-order.

Embedding Binary Data

Sometimes you have a file with some binary or text data that you'd like to make available to your component - but you don't want to reformat the file as C source.

You can set a variable COMPONENT_EMBED_FILES in component.mk, giving the names of the files to embed in this way:

```
COMPONENT_EMBED_FILES := server_root_cert.der
```

Or if the file is a string, you can use the variable COMPONENT_EMBED_TXTFILES. This will embed the contents of the text file as a null-terminated string:

```
COMPONENT_EMBED_TXTFILES := server_root_cert.pem
```

The file's contents will be added to the .rodata section in flash, and are available via symbol names as follows:

```
extern const uint8_t server_root_cert_pem_start[] asm("_binary_server_root_cert_pem_
↪start");
extern const uint8_t server_root_cert_pem_end[]   asm("_binary_server_root_cert_pem_
↪end");
```

The names are generated from the full name of the file, as given in COMPONENT_EMBED_FILES. Characters /, ., etc. are replaced with underscores. The _binary prefix in the symbol name is added by objcopy and is the same for both text and binary files.

For an example of using this technique, see protocols/https_mbedtls - 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 ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK defaults, it is possible to create a file `sdkconfig.defaults` in the project directory. This file will be used when running `make defconfig`, or creating a new config from scratch.

To override the name of this file, set the `SDKCONFIG_DEFAULTS` environment variable.

Save flash arguments

There're some scenarios that we want to flash the target board without IDF. For this case we want to save the built binaries, esptool.py and esptool write_flash arguments. It's simple to write a script to save binaries and esptool.py. We can use command `make print_flash_cmd`, it will print the flash arguments:

```
--flash_mode qio --flash_freq 40m --flash_size 2MB 0x0000 bootloader/bootloader.bin␣
↪0x10000 ssc.bin 0x8000 partitions_singleapp.bin
```

Then use flash arguments as the arguemnts for esptool write_flash arguments:

```
python esptool.py --chip esp8266 --port /dev/ttyUSB0 --baud 921600 --before default_
↪reset --after hard_reset write_flash -z --flash_mode qio --flash_freq 40m --flash_
↪size detect 0 bootloader/bootloader.bin 0x10000 example_app.bin 0x8000 partitions_
↪singleapp.bin
```

3.1.3 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.

3.2 Partition Tables

3.2.1 Overview

A single ESP8266's flash can contain multiple apps, as well as many different kinds of data (calibration data, filesystems, parameter storage, etc). For this reason a partition table is flashed to offset 0x8000 in the flash.

Partition table length is 0xC00 bytes (maximum 95 partition table entries). An MD5 checksum is appended after the table data.

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"
- "Two OTA app"

If you *make partition_table* then it will print a summary of the partition table.

3.2.2 Built-in Partition Tables

Here is the summary printed for the "Single factory app, no OTA" configuration:

```
# Espressif ESP8266 Partition Table
# Name,   Type, SubType, Offset,  Size
nvs,      data, nvs,     0x9000,  0x6000
phy_init, data, phy,     0xf000,  0x1000
factory,  app,  factory, 0x10000, 0xF0000
```

- 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 "Two OTA definitions" configuration:

```
# Espressif ESP8266 Partition Table
# Name,   Type, SubType, Offset,   Size
nvs,      data, nvs,     0x9000,   0x4000
otadata,  data, ota,     0xd000,   0x2000
phy_init, data, phy,     0xf000,   0x1000
ota_0,    0,    ota_0,   0x10000,  0xF0000
ota_1,    0,    ota_1,   0x110000, 0xF0000
```

- There are now two app partition definitions, ota_0 at 0x10000 and ota_1 at 0x110000
- 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 ota_0 app.

3.2.3 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
ota_0,    app,  ota_0,   0x10000,  0xF0000
ota_1,    app,  ota_1,   0x110000, 0xF0000
```

- 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 ESP8266. 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 ESP8266_RTOS_SDK 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.

ESP8266_RTOS_SDK 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 ota_0 (0x10), ota_1 (0x11) . . . ota_15 (0x1F) or test (0x20).

- ota_0 (0x10) is the default app partition. The bootloader will execute the ota_0 app unless there it sees another partition of type data/ota, in which case it reads this partition to determine which OTA image to boot.
- ota_0 (0x10) . . . ota_15 (0x1F) are the OTA app slots. If using OTA, an application should have at least two OTA application slots (ota_0 & ota_1).

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 **ESP_PHY_INIT_DATA_IN_PARTITION** option. You will also need to flash your devices with phy init data as the ESP8266_RTOS_SDK 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 ESP8266_RTOS_SDK uses.

Offset & Size

Please note that the app partition must fall in only one integrated partition of 1M. Otherwise, the application crashes.

The starting address of firmware is configured to 0x10000 by default. If you want to change the starting address of firmware, please:

- Configure the value in *menu -> partition table -> select "Custom partition table CSV" -> (0x10000) Factory app partition offset*;
- Configure the ota_1 offset in the CSV file of partition table to the value , and ota_2 offset to the mirror value (ota_2 = ota_1 + 0x100000).
  - Please enter an aligned offset. Otherwise, the tool will return errors.
  - Don't leave it blank, because, in this case, the tool will automatically align the app partition, which may cause app partition overlaps. That said, the app partition falls in more than one integrated partitions of 1M.

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).

3.2.4 Generating Binary Partition Table

The partition table which is flashed to the ESP8266 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.)

3.2.5 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.

3.3 System Tasks

This document explains the ESP8266 RTOS SDK internal system tasks.

3.3.1 Overview

The main tasks and their attributes are as following:

Names	stack size	Priority
uiT	3584(C)	14
IDLE	768	0
Tmr	2048(C)	2
ppT	2048(C)	13
pmT	1024	11
rtT	2048	12
tiT	2048(C)	8
esp_event_loop_task	2048(C)	10

Note: (C) means it is configurable by "menuconfig".

3.3.2 Tasks Introduction

uiT

This task initializes the system, including peripherals, file system, user entry function and so on. This task will delete itself and free the resources after calling *app_main*.

IDLE

This task is freeRTOS internal idle callback task, it is created when starting the freeRTOS. Its hook function is *vApplicationIdleHook*. The system's function of *sleep* and function of feeding *task watch dog* are called in the *vApplicationIdleHook*.

Tmr

This task is the processor of freeRTOS internal software timer.

ppT

This task is to process Wi-Fi hardware driver and stack. It posts messages from the logic link layer to the upper layer TCP/IP stack after transforming them into ethernet packets.

pmT

The task is for system power management. It will check if the system can sleep right now, and if it is, it will start preparing for system sleep.

rtT

The task is the processor of high priority hardware timer. It mainly process Wi-Fi real time events. It is suggested that functions based on this component should not be called in application, because it may block other low layer Wi-Fi functions.

tiT

The task is the main task of TCP-IP stack(LwIP) , it is to deal with TCP-IP packets.

esp_event_loop_task

The task processes system events, for example, Wi-Fi and TCP-IP stack events.

3.3.3 Suggestions

In general, the priority of user task should NOT be higher than the system real timer task's priority (12). So it is suggested that keep your user tasks' priorities less than 12. If you want to speed up the TCP/UDP throughput, you can try to set the priority of send/receive task to be higher than the "tiT" task's priority (8).

3.4 PWM & Sniffer Co-exists

3.4.1 1. Overview

Without hardware PWM, ESP8266 has to use the hardware timer to simulate the PWM. We are using the Wi-Fi internal timer to drive the PWM, so there may be resource competition issue when using PWM and sniffer/SmartConfig at the same time.

3.4.2 2. Root Cause

To ensure the high precision of the PWM, the hardware Timer1 will trigger the interrupt AHEAD_TICKS1(6us by default) earlier. And in the interrupt, it will poll to wait for AHEAD_TICKS1(6us by default). After handling the GPIO invert in one channel, the system will check the remaining time (T1) to the next channel invert.

If the T1 < AHEAD_TICKS2(8us by default), the system will not exit the interrupt, but poll to wait till timeout, and then invert the GPIO in the next channel; then the system will repeat these steps until all channels inverted.

So theoretically, the max time that PWM may occupy the CPU is 6 + 8 * n, n means the channel count. For example, if there are 3 channels, then PWM may take 30us at most.

In this case, PWM will affect the Wi-Fi sniffer/SmartConfig function, especially for the capture of the LDPC packets, or HT40 packets which require the CPU to handle them in time, otherwise those packets will loss.

3.4.3 3. Issue that may happen

If your application used both PWM and sniffer/SmartConfig, the sniffer/SmartConfig may take a long time to connect to an AP. You can stop the PWM and try it again. If the sniffer/SmartConfig becomes much faster, then it is the PWM that affect the sniffer/SmartConfig. In this case, you should adjust the frequency, duty cycle and phase of the PWM.

3.4.4 4. Suggestion

When using the PWM and SmartConfig at the same time, please note:

1. The PWM's frequency cannot be too high, 2KHz at most.
2. Revise the PWM's duty cycle and phase, make the time intervals (Tn) between each channel inverting be equal to 0 or be larger than 50us (Tn = 0, or Tn > 50).

3.5 FOTA from an Old SDK to the New ESP8266 RTOS SDK (IDF Style)

FOTA: firmware over the air, herein it means the firmware upgrading through Wi-Fi. Since the ESP8266 RTOS SDK V3.0, we refactored the SDK to be the ESP-IDF style. This document introduces the FOTA from a non-OS (or an old RTOS that earlier than V3.0) firmware to the new IDF style RTOS firmware. Please note that users need to make modifications to the application, because the new APIs are not compatible with the old SDKs', due to underlying changes in the architecture.

3.5.1 SDK Partition Map

Here are the partition maps of the old SDK and the new IDF style RTOS SDK:

1. The Old ESP8266 SDK

Boot/4KB	APP1	APP2	System Parameter/16KB

2. The New ESP8266 SDK (IDF Style)

Boot/16KB	Partition Table/4KB	NVS	APP1	APP2

In the new IDF style ESP8266 RTOS SDK SDK, each partition's base address is configurable in menuconfig, except boot's.

3.5.2 Firmware Compatibility

To implement FOTA from an old SDK firmware to the new one, users need to download all necessary partitions of the new firmware (including new boot, new partition table, and new application), into the old one's APP partition.

Then the new bootloader will unpack the packed new firmware, and copy each partition data to the target partition address.

When FOTA completing, the partition map may look like the following graph (what will it be is based on your actual partition table):

old SDK Boot/4KB	new Boot/16KB	new Partition Table/4KB | new NVS	new APP1	new APP2	System Parameter/16KB

In this case, there are about 40KB(4KB + 16KB + 4KB + 16KB) flash size cannot be used by users.

FOTA by Single Firmware URL

FOTA by Multi Firmware URLs

3.5.3 Workflow

Herein we provide an example of the FOTA.

Step 1: Connect to AP

Connect your host PC and the ESP8266 to the same AP.

Step 2: Configure and Build

Here, we use the system/ota/native_ota/1MB_flash/new_to_new_with_old if flash is 1MB or system/ota/native_ota/2+MB_flash/new_to_new_with_old if flash is 2MB or larger.

Open a new terminal on your PC, set the following configurations, and then compile the example:

1. Enter the target directory

```
cd $IDF_PATH/examples/system/ota
```

2. Enable the OTA compatibility function

```
Component config --->
    ESP8266-specific --->
        [*] (**Expected**)ESP8266 update from old SDK by OTA
```

3. ESP8285(ESP8266 + 1MB flash) configuration:

Configure the flash size according to your actual development board's flash.

```
Serial flasher config  --->
    Flash size (x MB)  ---> real flash size
```

4. Configure example's parameters

```
Example Configuration --->
    (myssid) WiFi SSID
    (mypassword) WiFi Password
    (192.168.0.3) HTTP Server IP
    (8070)HTTP Server Port
    (/hello_world.ota.bin) HTTP GET Filename
```

- WiFi SSID: Wi-Fi SSID of router
- WiFi Password: Wi-Fi password of router
- HTTP Server IP: It may be the PC's IP address
- HTTP Server Port: HTTP server port
- HTTP GET Filename: Using "ota.ota.bin" which is the target firmware of the example

5. Select connecting to the original AP

If users want to connect to the original AP of old SDK, then configurate as following:

```
Example Configuration  --->
    [*] Connect to the original AP
```

5. Build the project

Input following command to start building:

```
make ota
```

After compiling, the final firmware "ota.v2_to_v3.ota.bin" will be generated. Then users can download and update to this new firmware when running an old SDK OTA application.

- Note: The finally firmware's name mentioned above will be as "xxx.v2_to_v3.ota.bin", "xxx" is the name of your project.

4. Start HTTP Server

```
cd build
python -m SimpleHTTPServer 8070
```

3.5.4 Note

- It will take a lot of time for the new bootloader unpacking the firmware at the first time, please wait a while.
- **The terminal will print some log that shows the progress:**
  - log "I (281) boot: Start unpacking V3 firmware ...", it means that bootloader starts unpacking.
  - log "Pack V3 firmware successfully and start to reboot", it means that bootloader unpacked firmware successfully.

- This "unpacking workflow" will only be executed when it is an old SDK firmware that upgrade to the new SDK firmware, for example, V2.0 upgrade to V3.1. After that, the FOTA in later versions (for example, V3.1 upgrade to later) will be the normal FOTA workflow.

3.5.5 Inheritance Data

Users can perfer to the source code system/ota/native_ota/2+MB_flash/new_to_new_with_old/main/ota_example_main.c to check how to load original AP's information.

See structure **old_sysconf** in the file of esp8266/include/internal/esp_system_internal.h for the organization of this information.

3.6 Factory Test

3.6.1 1. Overview

The document introduces how to develop, compile, download and run the factory test firmware.

The factory test software development kit is also an example of the SDK, and it is located at examples/system/factory-test.

3.6.2 2. Development

Users can use ready-to-use applications directly, or can also add custom application code into the factory test software development kit.

More details of adding customer components, please refer to *Documentation for the GNU Make based build system*.

Users can just develop the factory test application as normal examples of the SDK.

2.1 Application code

Just like other applications, the entry function of factory test application is `app_main`. It should be added into the source code file of users. For example, users can add the `app_main` into `main.c` of the above sample project.

Users can refer to the source code in file /examples/system/factory-test/main/main.c to build custom project.

2.2 Linking address

The SDK's partition only supports two applications that named as `ota_0` and `ota_1`.

In this case, we link the factory test firmware to the partition of `ota_1`. So, please do not flash the factory test firmware into the partition of `ota_0`.

3.6.3 3. Compile

To make the bootloader run the `ota_1(factory test firmware)`, please enable the `GPIO triggers boot from test app partition` and set the `correct` GPIO of your development board in menuconfig:

```
Bootloader config  --->
    [*] GPIO triggers boot from test app partition
    (12)  Number of the GPIO input to boot TEST partition
```

Using the partition table file which has two "OTA" definitions partition:

```
Partition Table  --->
    Partition Table (Factory app, two OTA definitions)  --->
        (X) Factory app, two OTA definitions
```

Enable the console which is used for human-computer interaction:

```
Component config  --->
    Virtual file system  --->
        [*] Using espressif VFS
```

Enable pthread for this function:

```
Component config  --->
    PThreads  --->
        [*] Enable pthread
```

Then call command `make app2` in the terminal to compile the firmware which is able to run at `ota_1` partition. The `Make System` will start compiling bootloader, partition table file, factory test firmware and so on one by one.

3.6.4 3.1 Special Commands

1. `make app2`: only compile factory test firmware which is able to run at `ota_1`, `with` bootloader, partition table file and so on
2. `make app2-flash`: flash(download) only the factory test firmware which is able to run at `ota_1`, `without` bootloader, partition table file and so on
3. `make app2-flash-all`: flash(download) the factory test firmware which is able to run at `ota_1`, `with` bootloader, partition table file and so on

3.6.5 4. Download

Input command `make app2-flash-all` in the terminal to download bootloader, partition table file and factory test firmware which is located at `ota_1` one by one.

If users only want to download factory test firmware, please use command `make app2-flash` instead.

3.6.6 5. Run

Please hold the `correct` GPIO, which is configured in the menuconfig in Section 3 `Compile`, to be low level and power on. Input command `make monitor` in the terminal, and then logs will appear like following:

```
ets Jan  8 2013,rst cause:1, boot mode:(3,6)

load 0x40100000, len 7872, room 16
0x40100000: _stext at ??:?

tail 0
```

(continues on next page)

(continued from previous page)

```
chksum 0xf1
load 0x3ffe8408, len 24, room 8
tail 0
chksum 0x78
load 0x3ffe8420, len 3604, room 8
tail 12
chksum 0x1b
I (64) boot: ESP-IDF v3.2-dev-354-gba1f90cd-dirty 2nd stage bootloader
I (64) boot: compile time 13:56:17
I (72) qio_mode: Enabling default flash chip QIO
I (73) boot: SPI Speed      : 40MHz
I (80) boot: SPI Mode       : QIO
I (86) boot: SPI Flash Size : 2MB
I (92) boot: Partition Table:
I (98) boot: ## Label            Usage          Type ST Offset   Length
I (109) boot:  0 nvs              WiFi data        01 02 00009000 00004000
I (120) boot:  1 otadata          OTA data         01 00 0000d000 00002000
I (132) boot:  2 phy_init         RF data          01 01 0000f000 00001000
I (144) boot:  3 ota_0            OTA app          00 10 00010000 000f0000
I (155) boot:  4 ota_1            OTA app          00 11 00110000 000f0000
I (167) boot: End of partition table
I (173) boot: No factory image, trying OTA 0
I (5180) boot: Detect a boot condition of the test firmware
I (5180) esp_image: segment 0: paddr=0x00110010 vaddr=0x40210010 size=0x37b18␣
↪(228120) map
I (5263) esp_image: segment 1: paddr=0x00147b30 vaddr=0x3ffe8000 size=0x00718 ( ␣
↪1816) load
I (5264) esp_image: segment 2: paddr=0x00148250 vaddr=0x3ffe8718 size=0x0019c (  ␣
↪412) load
I (5275) esp_image: segment 3: paddr=0x001483f4 vaddr=0x40100000 size=0x084b0 (␣
↪33968) load
0x40100000: _stext at ??:?

I (5299) boot: Loaded app from partition at offset 0x110000
I (5340) system_api: Base MAC address is not set, read default base MAC address from␣
↪BLK0 of EFUSE
I (5340) system_api: Base MAC address is not set, read default base MAC address from␣
↪BLK0 of EFUSE
I (5530) phy_init: phy ver: 1055_12
I (5530) reset_reason: RTC reset 1 wakeup 0 store 0, reason is 1
I (5530) factory-test: SDK factory test firmware version:v3.2-dev-354-gba1f90cd-dirty
```

Then users can input test commands to start factory testing.

3.6.7 6. Test Commands

1. `rftest_init`:

   ```
   parameters: no

   function: initialize RF to prepare for test
   ```

2. `tx_contin_en <parameter 1>`:

```
parameter 1: value 1 means that chip transmits packets continuously with 92% duty␣
↪cycle,
             value 0 means that "iqview" test mode

function: set test mode
```

3. esp_tx <parameter 1> <parameter 2> <parameter 3>:

```
parameter 1: transmit channel which ranges from 1 to 14
parameter 2: transmit rate which ranges from 0 to 23
parameter 2: transmit power attenuation which ranges from -127 to 127, unit is 0.
↪25dB

function: start transmitting Wi-Fi packets

note 1: command "wifitxout" is the same as "esp_tx"
note 2: the function can be stopped by command "cmdstop"
```

4. esp_rx <parameter 1> <parameter 2>:

```
parameter 1: transmit channel which ranges from 1 to 14
parameter 2: transmit rate which ranges from 0 to 23

function: start receiving Wi-Fi packets

note 1: the function can be stopped by command "cmdstop"
```

5. wifiscwout <parameter 1> <parameter 2> <parameter 3>:

```
parameter 1: enable signal, value 1 means enable, value 0 means disable
parameter 2: transmit channel which ranges from 1 to 14
parameter 3: transmit power attenuation which ranges from -127 to 127, unit is 0.
↪25dB

function: start transmitting single carrier Wi-Fi packets

note 1: the function can be stopped by command "cmdstop"
```

6. cmdstop:

```
parameters: no

function: stop transmitting or receiving Wi-Fi packets

note 1: command "CmdStop" is the same as "cmdstop"
```

CHAPTER 4

General Notes

Adding this content here is to improve the user's development efficiency and avoid stepping into known problems.

4.1 1. Bootloader

V3.1 updated the bootloader to initialize SPI flash I/O mode and clock. So if you are using the V3.0 bootloader, and now upgrade to the new SDK, please disable the following configuration in the menuconfig:

```
"Bootloader config  --->
    [ ] Bootloader init SPI flash"
```

4.2 2. OTA

We split the native OTA example into several sub-examples to let custemors to choose which application matches the scenario they really want. examples/system/ota/native_ota.

4.3 3. 802.11n only AP

For better compatibility, the SDK is in bg mode by default. And application can set it to be bgn mode for reconnecting when it fails to connect some 11n only APs, refer to the examples/wifi/simple_wifi.

4.4 4. JTAG I/O

In some cases, if enable JTAG I/O (default options), it will cost some more current so that the hardware will cost more power. So if users don't use Jtag or these GPIOs directly and want to save more power, please enable this option in the menuconfig:

```
"Bootloader config  --->
    [ ] Bootloader disable JTAG I/O"
```

- genindex

Index

Symbols

A

B

C

E

F

G

H

I

M

P

R

S

T

U

V

W