Arduino-ESP32

Release 2.0.6

Espressif

Feb 08, 2023

CONTENTS:

1 Getting Started 3

2 Libraries 33

3 Guides 175

4 Tutorials 189

5 Advanced Utilities 215

6 Frequently Asked Questions 225

7 Troubleshooting 227

8 Contributions Guide 231

Here you will find all the relevant information about the project.

Note: This is a work in progress documentation and we will appreciate your help! We are looking for contributors!

CHAPTER

ONE

GETTING STARTED

1.1 About Arduino ESP32

Welcome to the Arduino ESP32 support documentation! Here you will find important information on how to use the project.

1.2 First Things First

Note: Before continuing, we must be clear that this project is supported by Espressif Systems and the community. Everyone is more than welcome to contribute back to this project.

ESP32 is a single 2.4 GHz Wi-Fi-and-Bluetooth SoC (System On a Chip) designed by Espressif Systems.

ESP32 is designed for mobile, wearable electronics, and Internet-of-Things (IoT) applications. It features all the state-of-the-art characteristics of low-power chips, including fine-grained clock gating, multiple power modes,and dynamic power scaling. For instance, in a low-power IoT sensor hub application scenario, ESP32 is woken-up periodically and only when a specified condition is detected. Low-duty cycle is used to minimize the amount of energy that the chip expends.

The output of the power amplifier is also adjustable, thus contributing to an optimal trade-off between communication range, data rate and power consumption.

The ESP32 series is available as a chip or module.

1.3 Supported SoC's

Here are the ESP32 series supported by the Arduino-ESP32 project:

SoC	Stable	Development	Datasheet
ESP32	Yes	Yes	ESP32
ESP32-S2	Yes	Yes	ESP32-S2
ESP32-C3	Yes	Yes	ESP32-C3
ESP32-S3	Yes	Yes	ESP32-S3

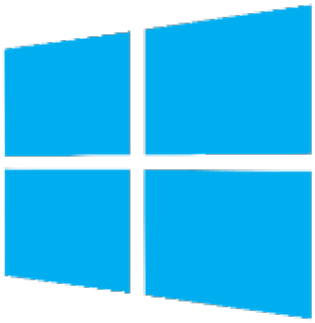
See Boards for more details about ESP32 development boards.

1.4 Arduino Core Reference

This documentation is built on the ESP32 and we are not going to cover the common Arduino API. To see the Arduino reference documentation, please consider reading the official documentation.

Arduino Official Documentation: Arduino Reference.

1.5 Supported Operating Systems

		
Windows	Linux	macOS

1.6 Supported IDEs

Here is the list of supported IDE for Arduino ESP32 support integration.

	
Arduino IDE	PlatformIO

See Installing Guides for more details on how to install the Arduino ESP32 support.

1.7 Support

This is an open project and it's supported by the community. Fell free to ask for help in one of the community channels.

1.8 Community

The Arduino community is huge! You can find a lot of useful content on the Internet. Here are some community channels where you may find information and ask for some help, if needed.

- ESP32 Forum: Official Espressif Forum.
- ESP32 Forum - Arduino: Official Espressif Forum for Arduino related discussions.
- ESP32 Forum - Hardware: Official Espressif Forum for Hardware related discussions.
- Gitter
- Espressif MCUs (Discord)
- ESP32 on Reddit

1.9 Issues Reporting

Before opening a new issue, please read this:

Be sure to search for a similar reported issue. This avoids duplicating or creating noise in the GitHub Issues reporting. We also have the troubleshooting guide to save your time on the most common issues reported by users.

For more details about creating new Issue, see the Issue Template.

If you have any new idea, see the Feature request Template.

1.10 First Steps

Here are the first steps to get the Arduino ESP32 support running.

To install Arduino-ESP32, please see the dedicated section on the Installation guide. We recommend you install it using the boards manager.

1.10.1 Installing

This guide will show how to install the Arduino-ESP32 support.

Before Installing

We recommend you install the support using your favorite IDE, but other options are available depending on your operating system. To install Arduino-ESP32 support, you can use one of the following options.

Installing using Arduino IDE

This is the way to install Arduino-ESP32 directly from the Arduino IDE.

Note: For overview of SoC's support, take a look on Supported Soc's table where you can find if the particular chip is under stable or development release.

- Stable release link:

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_
↪index.json
```

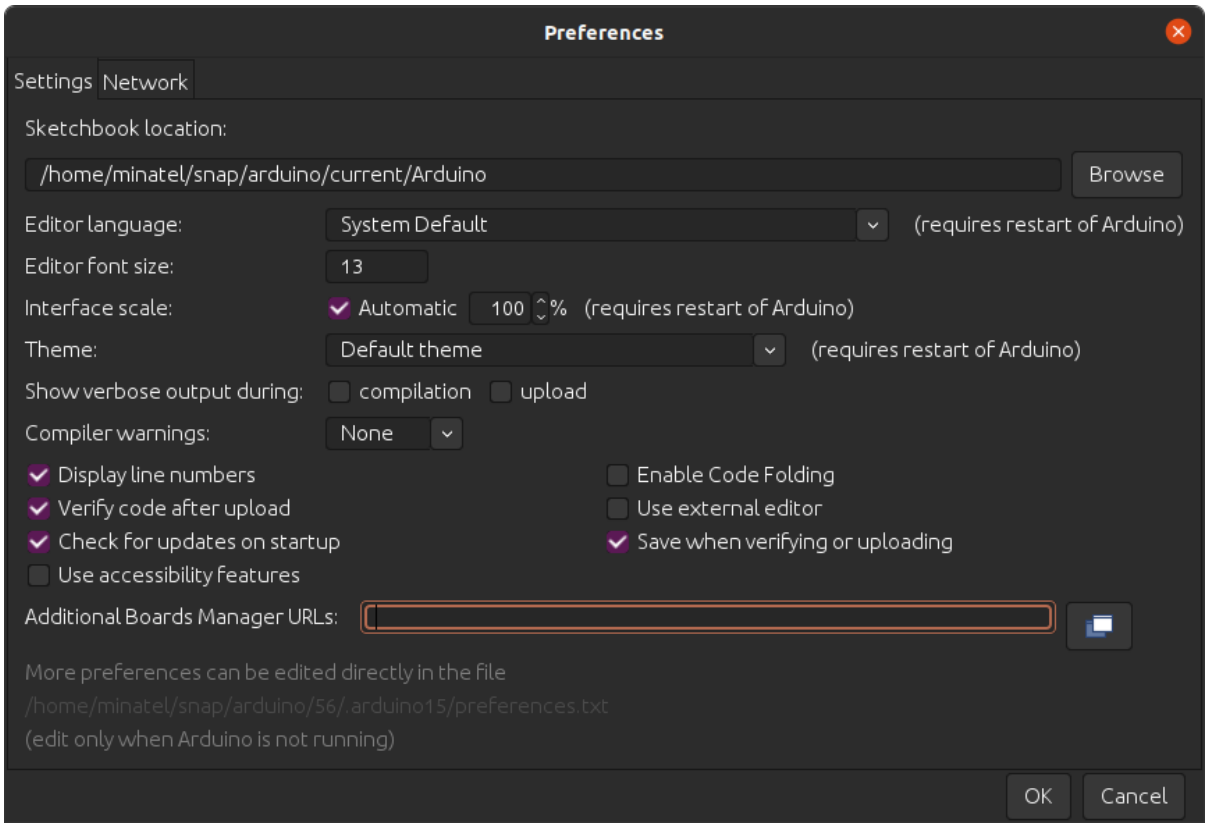
- Development release link:

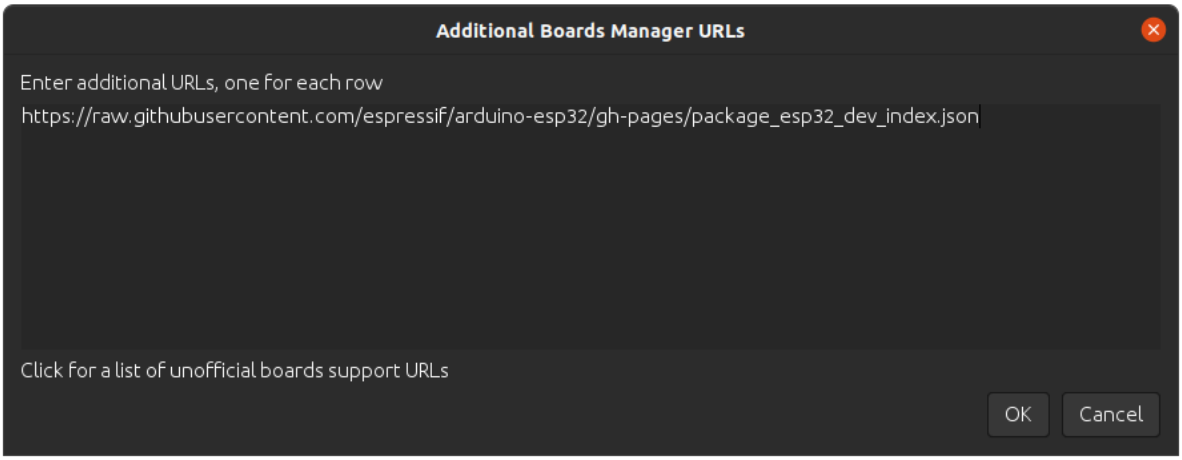
```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_
↪dev_index.json
```

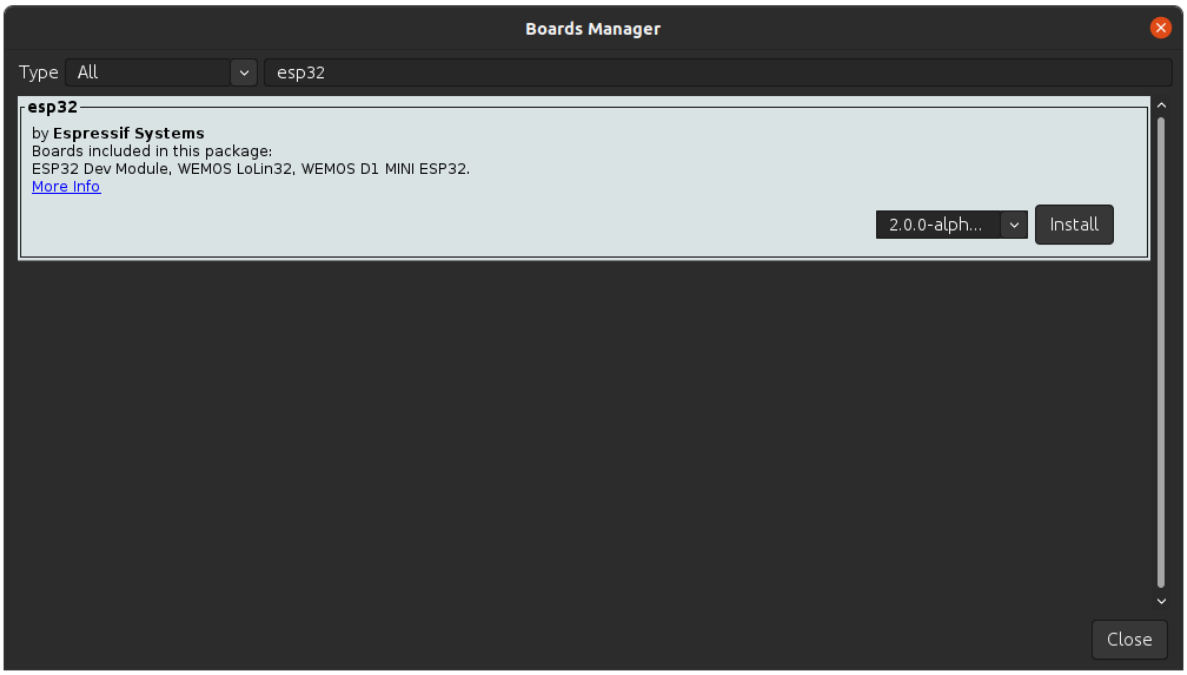
Note: Starting with the Arduino IDE version 1.6.4, Arduino allows installation of third-party platform packages using Boards Manager. We have packages available for Windows, macOS, and Linux.

To start the installation process using the Boards Managaer, follow these steps:

- Install the current upstream Arduino IDE at the 1.8 level or later. The current version is at the arduino.cc website.
- Start Arduino and open the Preferences window.
- Enter one of the release links above into *Additional Board Manager URLs* field. You can add multiple URLs, separating them with commas.
- Open Boards Manager from Tools > Board menu and install *esp32* platform (and do not forget to select your ESP32 board from Tools > Board menu after installation).
- Restart Arduino IDE.


Preferences
Settings
Network
Sketchbook location:
/home/minatel/snap/arduino/current/Arduino
Browse
Editor language:
System Default
(requires restart of Arduino)
Editor font size:
13
Interface scale:
Automatic
100
%
(requires restart of Arduino)
Theme:
Default theme
(requires restart of Arduino)
Show verbose output during:
compilation
upload
Compiler warnings:
None
Display line numbers
Verify code after upload
Check for updates on startup
Use accessibility features
Enable Code Folding
Use external editor
Save when verifying or uploading
Additional Boards Manager URLs:
More preferences can be edited directly in the file
/home/minatel/snap/arduino/56/.arduino15/preferences.txt
(edit only when Arduino is not running)
OK
Cancel


Additional Boards Manager URLs
Enter additional URLs, one for each row
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_dev_index.json
Click for a list of unofficial boards support URLs
OK
Cancel



Installing using PlatformIO

PlatformIO is a professional collaborative platform for embedded development. It has out-of-the-box support for ESP32 SoCs and allows working with Arduino ESP32 as well as ESP-IDF from Espressif without changing your development environment. PlatformIO includes lots of instruments for the most common development tasks such as debugging, unit testing, and static code analysis.

A detailed overview of the PlatformIO ecosystem and its philosophy can be found in the official documentation.

PlatformIO can be used in two flavors:

- PlatformIO IDE is a toolset for embedded C/C++ development available on Windows, macOS and Linux platforms
- PlatformIO Core (CLI) is a command-line tool that consists of a multi-platform build system, platform and library managers and other integration components. It can be used with a variety of code development environments and allows integration with cloud platforms and web services

To install PlatformIO, you can follow this Getting Started, provided at docs.platformio.org.

Using the stable code

Note: A detailed overview of supported development boards, examples and frameworks can be found on the official Espressif32 dev-platform page in the PlatformIO Registry.

The most reliable and easiest way to get started is to use the latest stable version of the ESP32 development platform that passed all tests/verifications and can be used in production.

Create a new project and select one of the available boards. You can change after by changing the platformio.ini file.

- For ESP32

```
[env:esp32dev]
platform = espressif32
board = esp32dev
framework = arduino
```

- For ESP32-S2 (ESP32-S2-Saola-1 board)

```
[env:esp32-s2-saola-1]
platform = espressif32
board = esp32-s2-saola-1
framework = arduino
```

- For ESP32-C3 (ESP32-C3-DevKitM-1 board)

```
[env:esp32-c3-devkitm-1]
platform = espressif32
board = esp32-c3-devkitm-1
framework = arduino
```

How to update to the latest code

To test the latest Arduino ESP32, you need to change your project *platformio.ini* accordingly. The following configuration uses the upstream version of the Espressif development platform and the latest Arduino core directly from the Espressif GitHub repository:

```
[env:esp32-c3-devkitm-1]
platform = https://github.com/platformio/platform-espressif32.git
board = esp32-c3-devkitm-1
framework = arduino
platform_packages =
    framework-arduinoespressif32 @ https://github.com/espressif/arduino-esp32#master
```

To get more information about PlatformIO, see the following links:

- PlatformIO Core (CLI)
- PlatformIO Home
- Tutorials and Examples
- Library Management

Windows (manual installation)

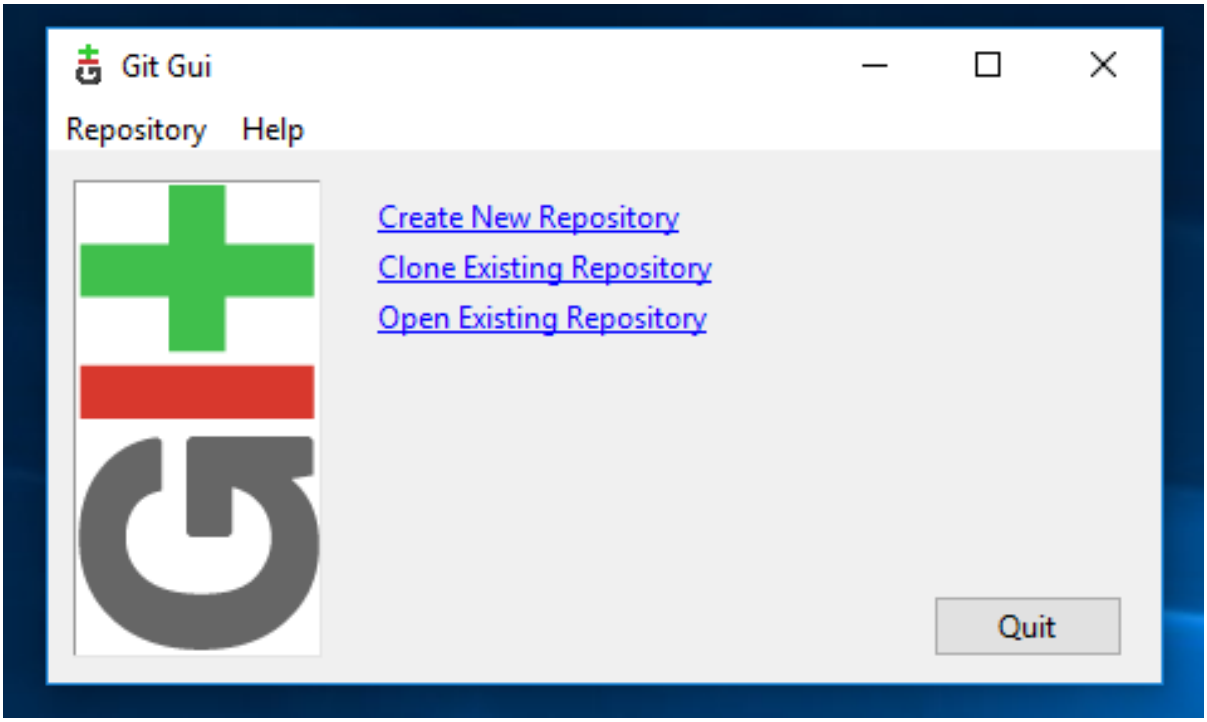
> **Warning:** Arduino ESP32 core v2.x.x cannot be used on Windows 8.x x86 (32 bits), Windows 7 or earlier. The Windows 32 bits OS is no longer supported by this toolchain.
>
> The Arduino ESP32 v1.0.6 still works on WIN32. You might want to install python 3.8.x because it is the latest release supported by Windows 7.

Steps to install Arduino ESP32 support on Windows:

Step 1

1. Download and install the latest Arduino IDE `Windows Installer` from [arduino.cc](https://www.arduino.cc/en/Main/Software)
2. Download and install Git from [git-scm.com](https://git-scm.com/download/win)
3. Start `Git GUI` and do the following steps:

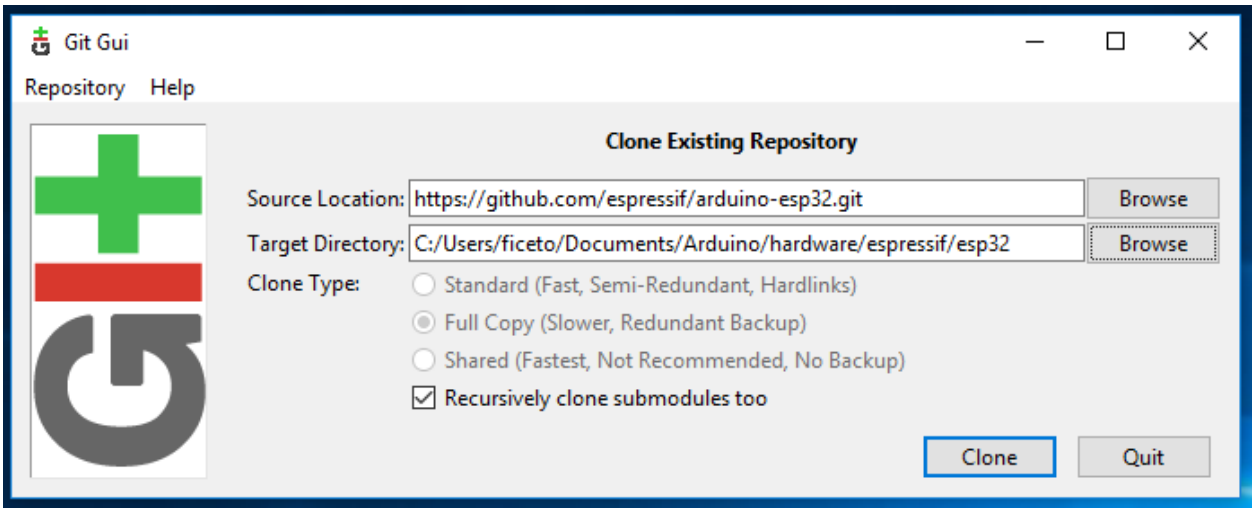
- Select `Clone Existing Repository`

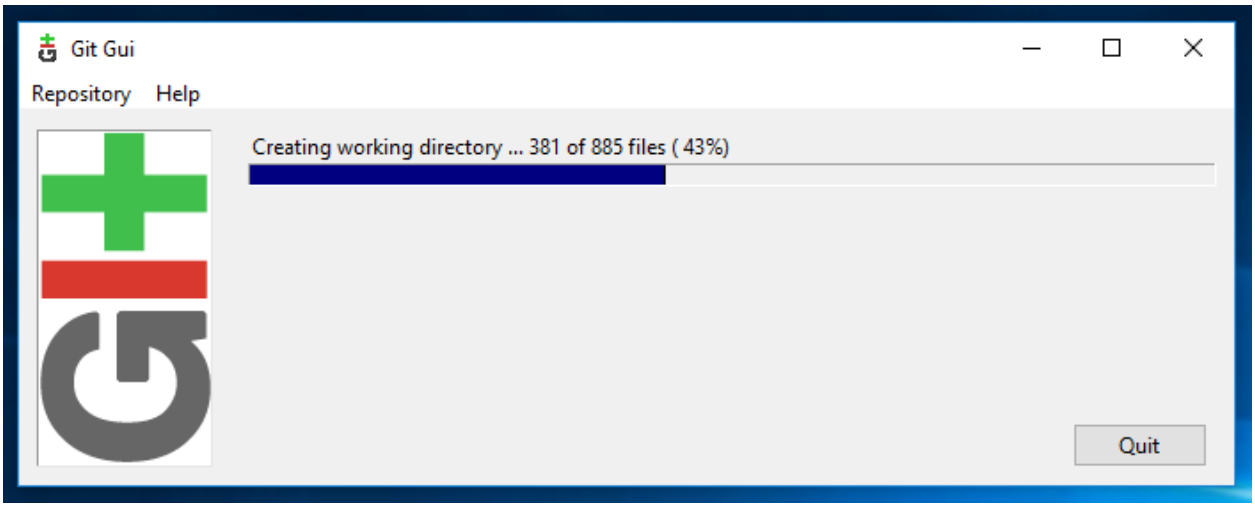


- **Select source and destination**
  - Sketchbook Directory: Usually `C:/Users/[YOUR_USER_NAME]/Documents/Arduino` and is listed underneath the "Sketchbook location" in Arduino preferences.
  - Source Location: `https://github.com/espressif/arduino-esp32.git`
  - Target Directory: `[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32`
  - Click `Clone` to start cloning the repository

Step 2

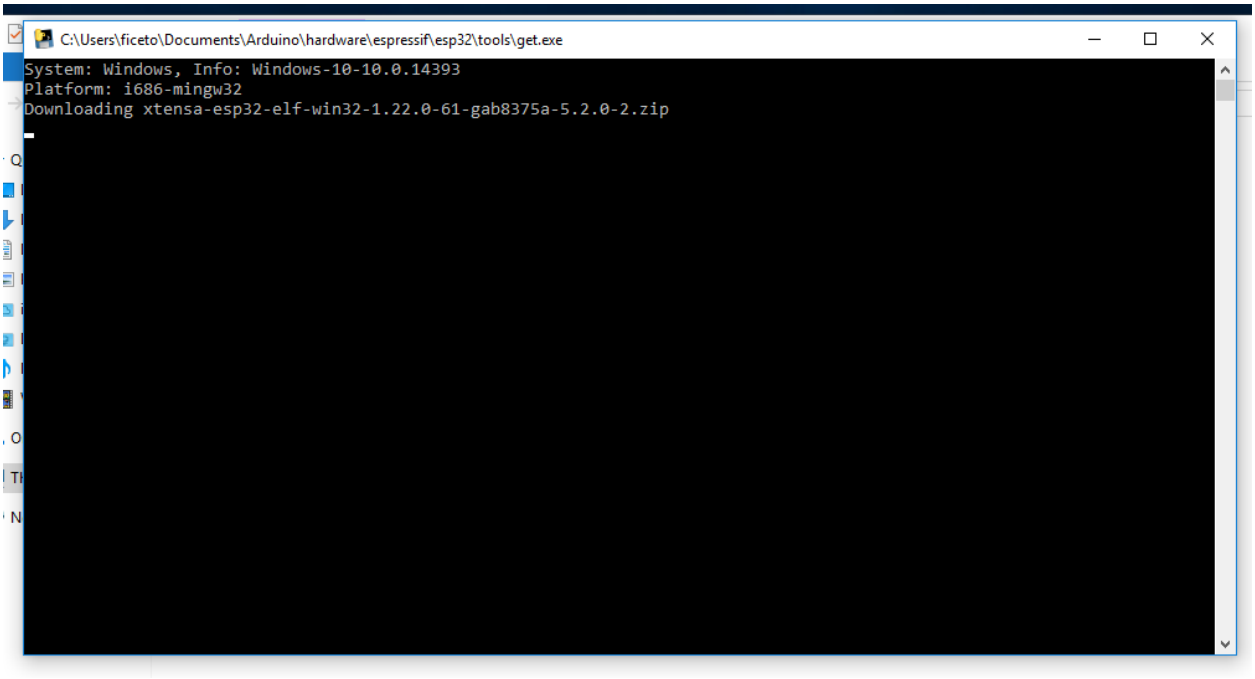
Step 3


Git Gui
Repository Help
Clone Existing Repository
Source Location: https://github.com/espressif/arduino-esp32.git
Browse
Target Directory: C:/Users/ficeto/Documents/Arduino/hardware/espressif/esp32
Browse
Clone Type:
Standard (Fast, Semi-Redundant, Hardlinks)
Full Copy (Slower, Redundant Backup)
Shared (Fastest, Not Recommended, No Backup)
Recursively clone submodules too
Clone
Quit


Git Gui
Repository Help
Creating working directory ... 381 of 885 files (43%)
Quit

- open a *Git Bash* session pointing to `[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32` and execute `` `git submodule update --init --recursive` ``
- Open `[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32/tools` and double-click `get.exe`

Step 4



- When `` `get.exe` `` finishes, you should see the following files in the directory

Step 5

1. Plug your ESP32 board and wait for the drivers to install (or install manually any that might be required)
2. Start Arduino IDE
3. Select your board in `Tools > Board` menu
4. Select the COM port that the board is attached to
5. Compile and upload (You might need to hold the boot button while uploading)

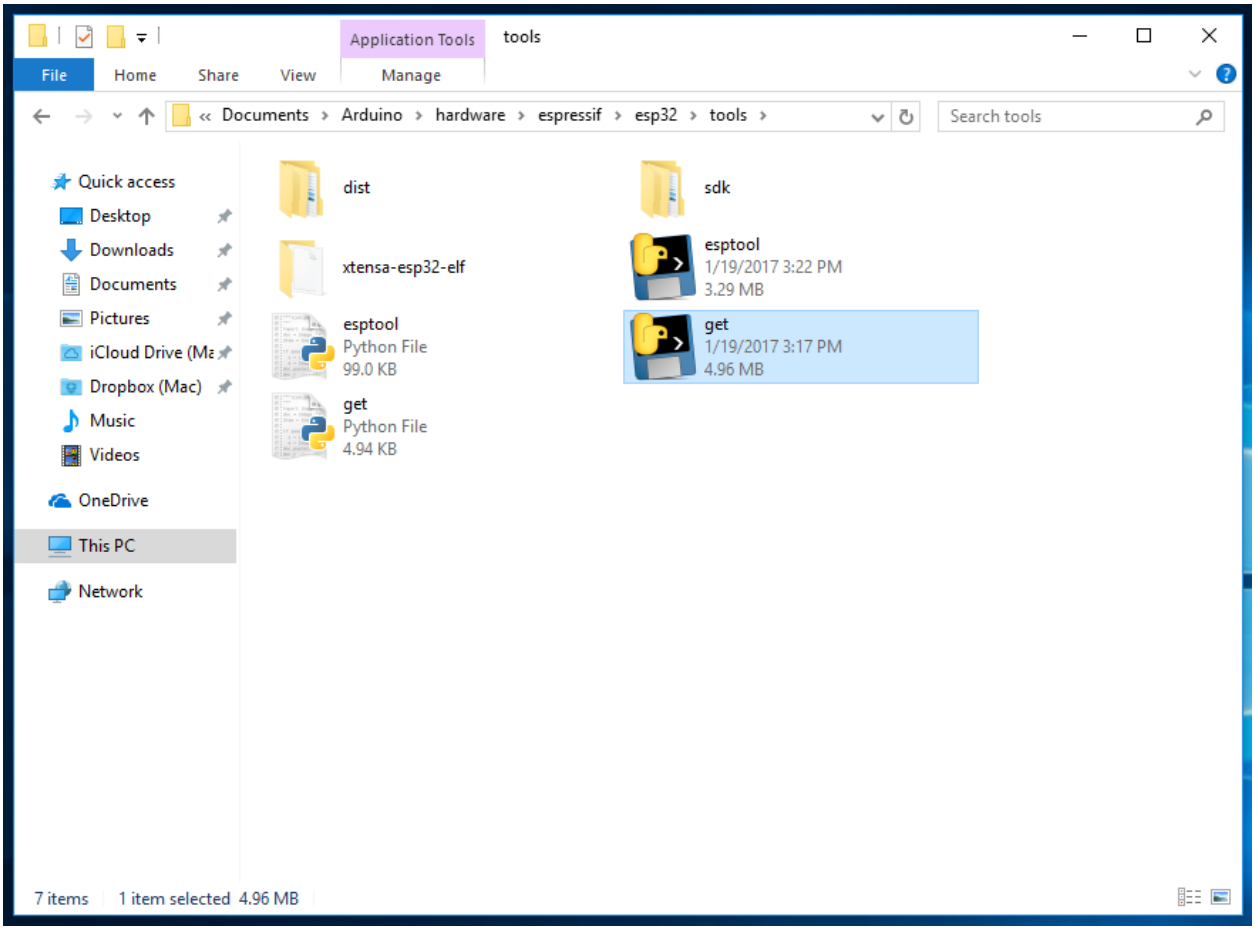
How to update to the latest code

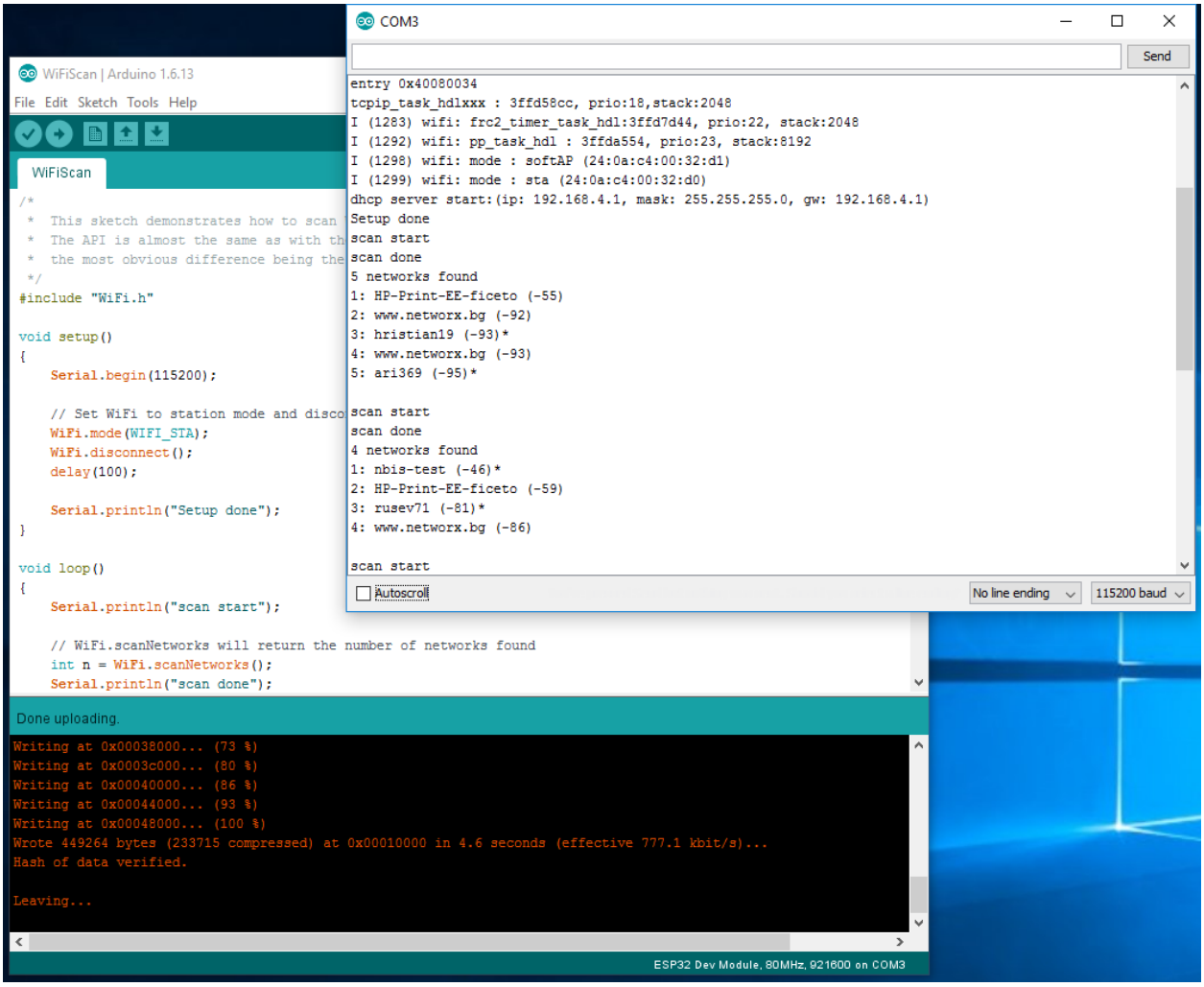
1. Start `Git GUI` and you should see the repository under `Open Recent Repository`. Click on it!

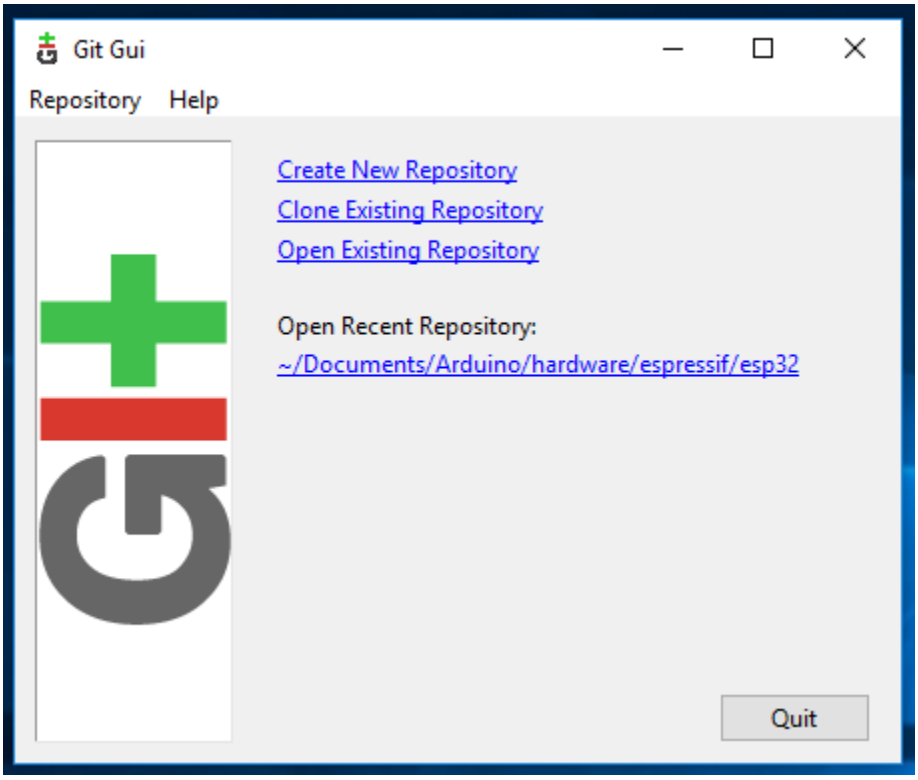
1. From menu `Remote` select `Fetch from > origin`

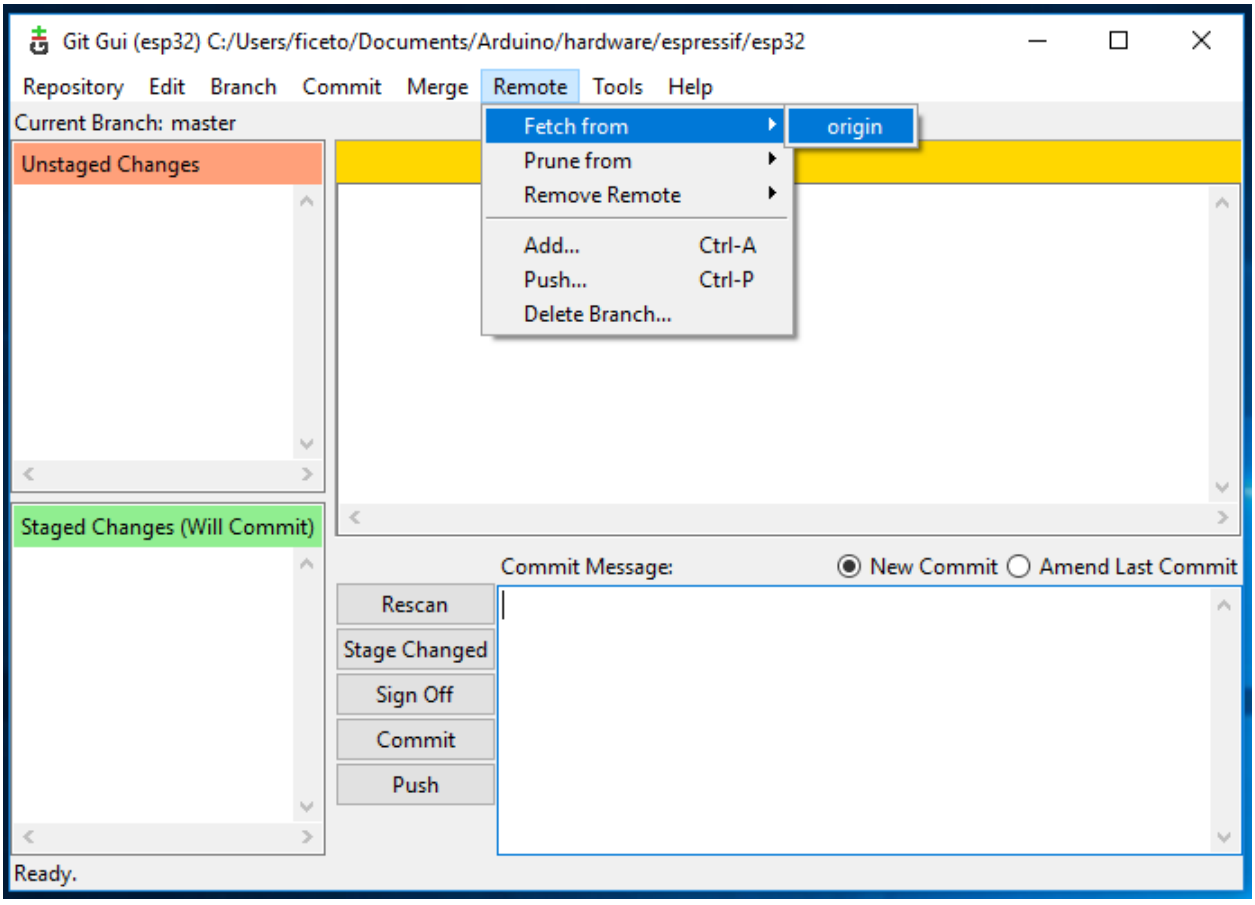
1. Wait for git to pull any changes and close `Git GUI`

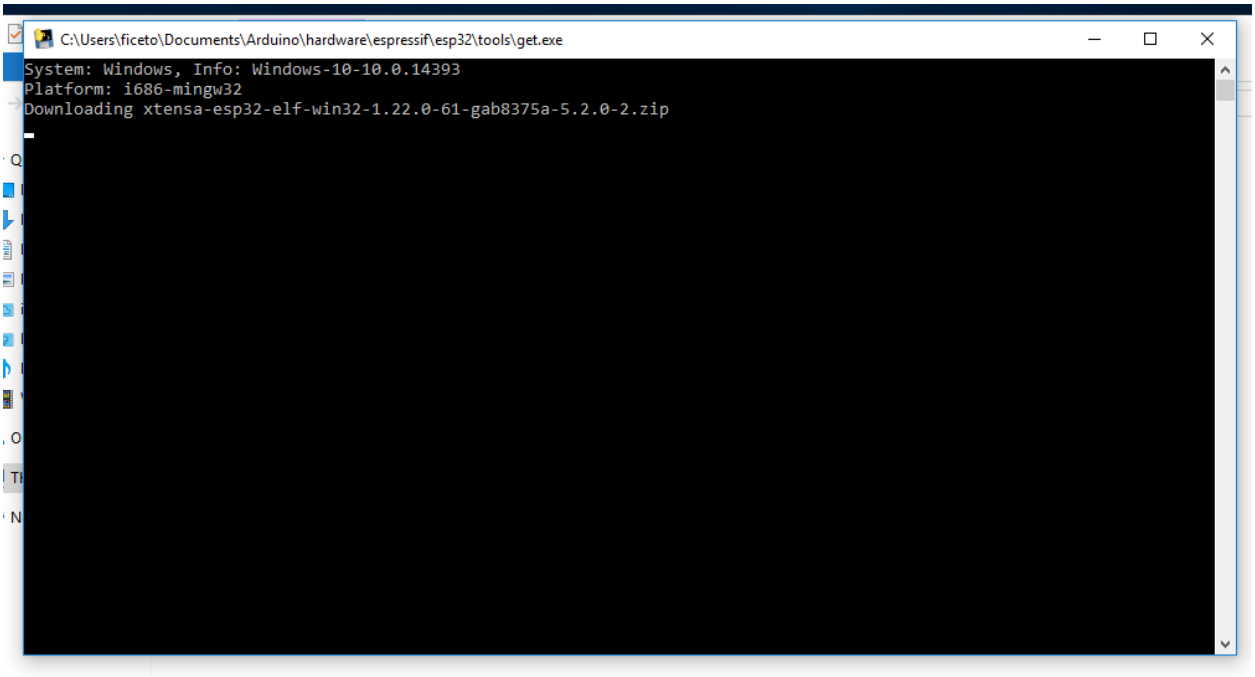
2. Open `[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32/tools` and double-click `get.exe`


Application Tools
tools
File
Home
Share
View
Manage
« Documents › Arduino › hardware › espressif › esp32 › tools ›
Search tools
Quick access
Desktop
Downloads
Documents
Pictures
iCloud Drive (Ma
Dropbox (Mac)
Music
Videos
OneDrive
This PC
Network
dist
xtensa-esp32-elf
esptool
Python File
99.0 KB
get
Python File
4.94 KB
sdk
esptool
1/19/2017 3:22 PM
3.29 MB
get
1/19/2017 3:17 PM
4.96 MB
7 items
1 item selected 4.96 MB


COM3
Send
entry 0x40080034
tcpip_task_hdlxxx : 3ffd58cc, prio:18,stack:2048
I (1283) wifi: frc2_timer_task_hdl:3ffd7d44, prio:22, stack:2048
I (1292) wifi: pp_task_hdl : 3ffda554, prio:23, stack:8192
I (1298) wifi: mode : softAP (24:0a:c4:00:32:d1)
I (1299) wifi: mode : sta (24:0a:c4:00:32:d0)
dhcp server start:(ip: 192.168.4.1, mask: 255.255.255.0, gw: 192.168.4.1)
Setup done
scan start
scan done
5 networks found
1: HP-Print-EE-ficeto (-55)
2: www.networx.bg (-92)
3: hristian19 (-93)*
4: www.networx.bg (-93)
5: ari369 (-95)*
scan start
scan done
4 networks found
1: nbis-test (-46)*
2: HP-Print-EE-ficeto (-59)
3: rusev71 (-81)*
4: www.networx.bg (-86)
scan start
Autoscroll
No line ending
115200 baud
WiFiScan | Arduino 1.6.13
File Edit Sketch Tools Help
WiFiScan
/*
* This sketch demonstrates how to scan
* The API is almost the same as with th
* the most obvious difference being the
*/
#include "WiFi.h"
void setup()
{
Serial.begin(115200);
// Set WiFi to station mode and disco
WiFi.mode(WIFI_STA);
WiFi.disconnect();
delay(100);
Serial.println("Setup done");
}
void loop()
{
Serial.println("scan start");
// WiFi.scanNetworks will return the number of networks found
int n = WiFi.scanNetworks();
Serial.println("scan done");
Done uploading.
Writing at 0x00038000... (73 %)
Writing at 0x0003c000... (80 %)
Writing at 0x00040000... (86 %)
Writing at 0x00044000... (93 %)
Writing at 0x00048000... (100 %)
Wrote 449264 bytes (233715 compressed) at 0x00010000 in 4.6 seconds (effective 777.1 kbit/s)...
Hash of data verified.
Leaving...
ESP32 Dev Module, 80MHz, 921600 on COM3


Git Gui
Repository
Help
Create New Repository
Clone Existing Repository
Open Existing Repository
Open Recent Repository:
~/Documents/Arduino/hardware/espressif/esp32
Quit


Git Gui (esp32) C:/Users/ficeto/Documents/Arduino/hardware/espressif/esp32
Repository
Edit
Branch
Commit
Merge
Remote
Tools
Help
Current Branch: master
Unstaged Changes
Staged Changes (Will Commit)
Fetch from
Prune from
Remove Remote
Add... Ctrl-A
Push... Ctrl-P
Delete Branch...
origin
Commit Message:
New Commit
Amend Last Commit
Rescan
Stage Changed
Sign Off
Commit
Push
Ready.



Linux

Debian/Ubuntu

- Install latest Arduino IDE from arduino.cc.
- Open Terminal and execute the following command (copy -> paste and hit enter):

```
sudo usermod -a -G dialout $USER && \
sudo apt-get install git && \
wget https://bootstrap.pypa.io/get-pip.py && \
sudo python3 get-pip.py && \
sudo pip3 install pyserial && \
mkdir -p ~/Arduino/hardware/espressif && \
cd ~/Arduino/hardware/espressif && \
git clone https://github.com/espressif/arduino-esp32.git esp32 && \
```

(continues on next page)

(continued from previous page)

```
cd esp32/tools && \
python3 get.py
```

- Restart Arduino IDE.
- If you have Arduino installed to ~/, modify the installation as follows, beginning at *mkdir -p ~/Arduino/hardware*:

```
cd ~/Arduino/hardware
mkdir -p espressif && \
cd espressif && \
git clone https://github.com/espressif/arduino-esp32.git esp32 && \
cd esp32/tools && \
python3 get.py
```

Fedora

- Install the latest Arduino IDE from arduino.cc.

Note: Command `$ sudo dnf -y install arduino` will most likely install an older release.

- Open Terminal and execute the following command (copy -> paste and hit enter):

```
sudo usermod -a -G dialout $USER && \
sudo dnf install git python3-pip python3-pyserial && \
mkdir -p ~/Arduino/hardware/espressif && \
cd ~/Arduino/hardware/espressif && \
git clone https://github.com/espressif/arduino-esp32.git esp32 && \
cd esp32/tools && \
python get.py
```

- Restart Arduino IDE.

openSUSE

- Install the latest Arduino IDE from arduino.cc.
- Open Terminal and execute the following command (copy -> paste and hit enter):

```
sudo usermod -a -G dialout $USER && \
if [ `python --version 2>&1 | grep '2.7' | wc -l` = "1" ]; then \
sudo zypper install git python-pip python-pyserial; \
else \
sudo zypper install git python3-pip python3-pyserial; \
fi && \
mkdir -p ~/Arduino/hardware/espressif && \
cd ~/Arduino/hardware/espressif && \
git clone https://github.com/espressif/arduino-esp32.git esp32 && \
cd esp32/tools && \
python get.py
```

- Restart Arduino IDE.

macOS

- Install the latest Arduino IDE from arduino.cc.
- Open Terminal and execute the following command (copy -> paste and hit enter):

```
mkdir -p ~/Documents/Arduino/hardware/espressif && \
cd ~/Documents/Arduino/hardware/espressif && \
git clone https://github.com/espressif/arduino-esp32.git esp32 && \
cd esp32/tools && \
python get.py
```

Where `~/Documents/Arduino` represents your sketch book location as per "Arduino" > "Preferences" > "Sketchbook location" (in the IDE once started). Adjust the command above accordingly.

- If you get the error below, install through the command line dev tools with *xcode-select –install* and try the command above again:

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

- Run the command:

```
xcode-select --install
```

- Try `python3` instead of `python` if you get the error: `IOError: [Errno socket error] [SSL: TLSV1_ALERT_PROTOCOL_VERSION] tlsv1 alert protocol version (_ssl.c:590)` when running `python get.py`
- If you get the following error when running `python get.py` urllib.error.URLError: <urlopen error SSL: CERTIFICATE_VERIFY_FAILED, go to Macintosh HD > Applications > Python3.6 folder (or any other python version), and run the following scripts: Install Certificates.command and Update Shell Profile.command
- Restart Arduino IDE.

1.10.2 Boards

Development Boards

You will need a development board or a custom board with the ESP32 (see Supported SoC's) to start playing. There is a bunch of different types and models widely available on the Internet. You need to choose one that covers all your requirements.

To help you on this selection, we point out some facts about choosing the proper boards to help you to save money and time.

One ESP32 to rule them all!

One important information that usually bring about some confusion is regarding the different models of the ESP32 SoC and modules.

The ESP32 is divided by family:

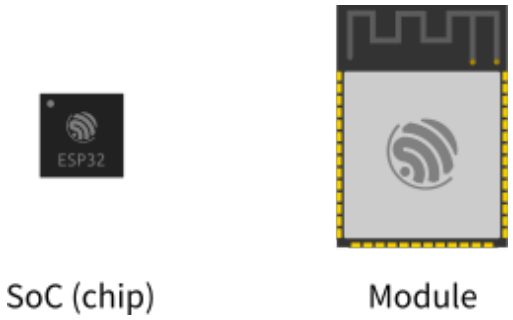
- **ESP32**
  - Wi-Fi and BLE
- **ESP32-S**

  - Wi-Fi only

- **ESP32-C**
  - Wi-Fi and BLE 5

For each family, we have SoC variants with some differentiation. The differences are more about the embedded flash and its size and the number of the cores (dual or single).

The modules use the SoC internally, including the external flash, PSRAM (in some models) and other essential electronic components. Essentially, all modules from the same family use the same SoC.



For example:

The SoC partnumber is the ESP32-D0WD-V3 and it's the same SoC used inside of the ESP32-WROVER (with PSRAM) and ESP32-WROOM modules. This means that the same characteristics are present in both modules' core.

For more detailed information regarding the SoC's and modules, see the Espressif Product Selector.

Now that you know that the module can be different but the heart is the same, you can choose your development board.

Before buying: Keep in mind that for some "must have" features when choosing the best board for your needs:

- **Embedded USB-to-Serial**
  - This is very convenient for programming and monitoring the logs with the terminal via USB.
- **Breadboard friendly**
  - If you are prototyping, this will be very useful to connect your board directly on the breadboard.
- **open-source/open-hardware**
  - Check if the schematics are available for download. This helps a lot on prototyping.
- **Support**
  - Some of the manufacturers offer a very good level of support, with examples and demo projects.

Espressif

ESP32-DevKitC-1

The *ESP32-DevKitC-1* development board is one of Espressif's official boards. This board is based on the ESP32-WROVER-E module, with the ESP32 as the core.

Specifications

- Wi-Fi 802.11 b/g/n (802.11n up to 150 Mbps)
- Bluetooth v4.2 BR/EDR and BLE specification
- Built around ESP32 series of SoCs
- Integrated 4 MB SPI flash
- Integrated 8 MB PSRAM
- **Peripherals**
  - SD card
  - UART
  - SPI
  - SDIO
  - I2C
  - LED PWM
  - Motor PWM
  - I2S
  - IR
  - Pulse Counter
  - GPIO
  - Capacitive Touch Sensor
  - ADC
  - DAC
  - Two-Wire Automotive Interface (TWAI®, compatible with ISO11898-1)
- Onboard PCB antenna or external antenna connector

Header Block

Note: Not all of the chip pins are exposed to the pin headers.

J1

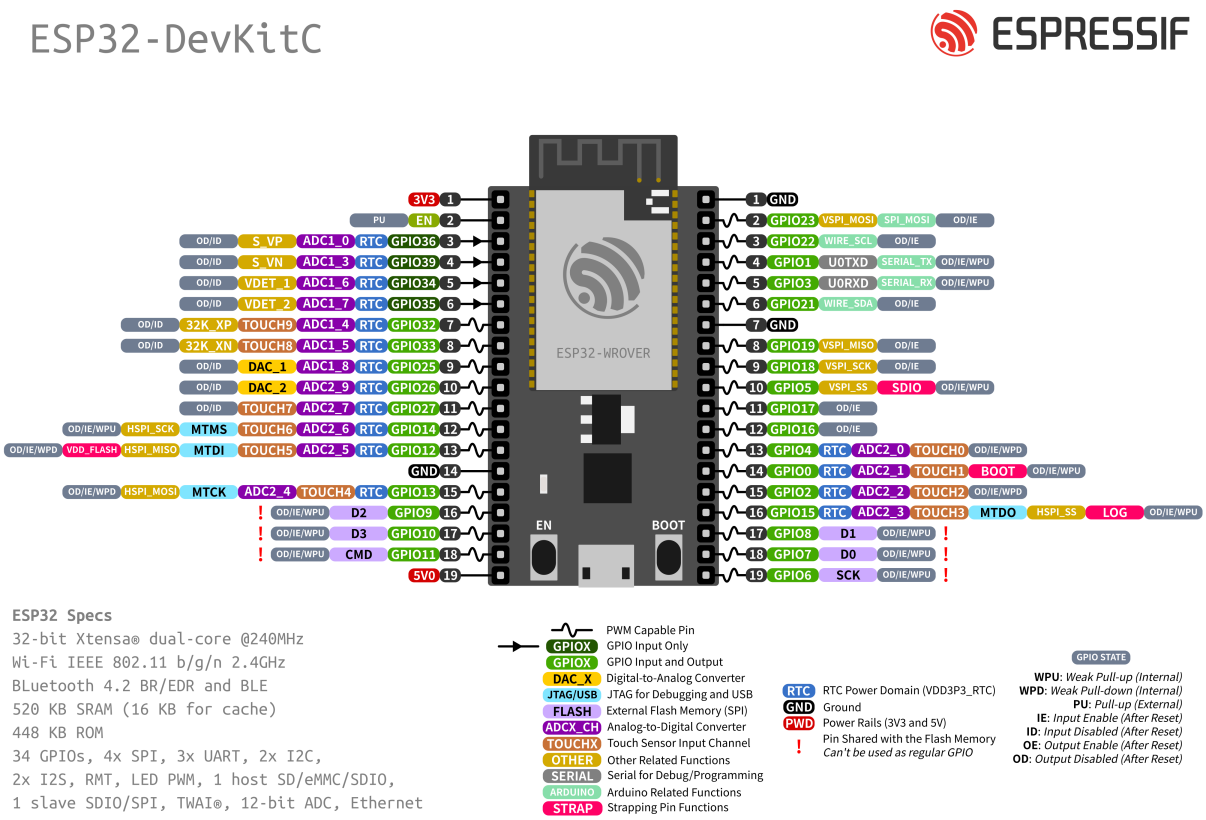
No.	Name	Type	Function
1	3V3	P	3.3 V power supply
2	EN	I	CHIP_PU, Reset
3	IO36	I	GPIO36, ADC1_CH0, S_VP
4	IO39	I	GPIO39, ADC1_CH3, S_VN
5	IO34	I	GPIO34, ADC1_CH6, VDET_1
6	IO35	I	GPIO35, ADC1_CH7, VDET_2
7	IO32	I/O	GPIO32, ADC1_CH4, TOUCH_CH9, XTAL_32K_P
8	IO33	I/O	GPIO33, ADC1_CH5, TOUCH_CH8, XTAL_32K_N
9	IO25	I/O	GPIO25, ADC1_CH8, DAC_1
10	IO26	I/O	GPIO26, ADC2_CH9, DAC_2
11	IO27	I/O	GPIO27, ADC2_CH7, TOUCH_CH7
12	IO14	I/O	GPIO14, ADC2_CH6, TOUCH_CH6, MTMS
13	IO12	I/O	GPIO12, ADC2_CH5, TOUCH_CH5, MTDI
14	GND	G	Ground
15	IO13	I/O	GPIO13, ADC2_CH4, TOUCH_CH4, MTCK
16	IO9	I/O	GPIO9, D2
17	IO10	I/O	GPIO10, D3
18	IO11	I/O	GPIO11, CMD
19	5V0	P	5 V power supply

J3

No.	Name	Type	Function
1	GND	G	Ground
2	IO23	I/O	GPIO23
3	IO22	I/O	GPIO22
4	IO1	I/O	GPIO1, U0TXD
5	IO3	I/O	GPIO3, U0RXD
6	IO21	I/O	GPIO21
7	GND	G	Ground
8	IO19	I/O	GPIO19
9	IO18	I/O	GPIO18
10	IO5	I/O	GPIO5
11	IO17	I/O	GPIO17
12	IO16	I/O	GPIO16
13	IO4	I/O	GPIO4, ADC2_CH0, TOUCH_CH0
14	IO0	I/O	GPIO0, ADC2_CH1, TOUCH_CH1, Boot
16	IO2	I/O	GPIO2, ADC2_CH2, TOUCH_CH2
17	IO15	I/O	GPIO15, ADC2_CH3, TOUCH_CH3, MTDO
17	IO8	I/O	GPIO8, D1
18	IO7	I/O	GPIO7, D0
19	IO6	I/O	GPIO6, SCK

P: Power supply; I: Input; O: Output; T: High impedance.

Pin Layout



Strapping Pins

Some of the GPIO's have important features during the booting process. Here is the list of the strapping pins on the ESP32.

GPIO	Default	Function	Pull-up	Pull-down
IO12	Pull-down	Voltage of Internal LDO (VDD_SDIO)	1V8	3V3
IO0	Pull-up	Booting Mode	SPI Boot	Download Boot
IO2	Pull-down	Booting Mode	Don't Care	Download Boot
IO15	Pull-up	Enabling/Disabling Log Print During Booting and Timing of SDIO Slave	U0TXD Active	U0TXD Silent
IO5	Pull-up	Timing of SDIO Slave	See ESP32	See ESP32

Be aware when choosing which pins to use.

Restricted Usage GPIO's

Some of the GPIO's are used for the external flash and PSRAM. These GPIO's cannot be used:

GPIO	Shared Function
IO6	External SPI Flash
IO7	External SPI Flash
IO8	External SPI Flash
IO9	External SPI Flash
IO10	External SPI Flash
IO11	External SPI Flash

Other GPIO's are *INPUT ONLY* and cannot be used as output pin:

GPIO	Function
IO34	GPIO34, ADC1_CH6, VDET_1
IO35	GPIO35, ADC1_CH7, VDET_2
IO36	GPIO36, ADC1_CH0, S_VP
IO39	GPIO39, ADC1_CH3, S_VN

Resources

- ESP32 (Datasheet)
- ESP32-WROVER-E (Datasheet)
- ESP32-DevKitC (Schematic)

ESP32-S2-Saola-1

The ESP32-S2-Saola-1 development board is one of Espressif's official boards. This board is based on the ESP32-S2-WROVER module, with the ESP32-S2 as the core.

Specifications

- Wi-Fi 802.11 b/g/n (802.11n up to 150 Mbps)
- Built around ESP32-S2 series of SoCs Xtensa® single-core
- Integrated 4 MB SPI flash
- Integrated 2 MB PSRAM
- **Peripherals**
  - 43 × programmable GPIOs
  - 2 × 13-bit SAR ADCs, up to 20 channels
  - 2 × 8-bit DAC
  - 14 × touch sensing IOs
  - 4 × SPI

  - 1 × I2S
  - 2 × I2C
  - 2 × UART
  - RMT (TX/RX)
  - LED PWM controller, up to 8 channels
  - 1 × full-speed USB OTG
  - 1 × temperature sensor
  - 1 × DVP 8/16 camera interface, implemented using the hardware resources of I2S
  - 1 × LCD interface (8-bit serial RGB/8080/6800), implemented using the hardware resources of SPI2
  - 1 × LCD interface (8/16/24-bit parallel), implemented using the hardware resources of I2S
  - 1 × TWAI® controller (compatible with ISO 11898-1)
- Onboard PCB antenna or external antenna connector

Header Block

Note: Not all of the chip pins are exposed to the pin headers.

J2

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

J3

No.	Name	Type	Function
1	GND	G	Ground
2	RST	I	CHIP_PU, Reset
3	IO46	I	GPIO46
4	IO45	I/O	GPIO45
5	IO44	I/O	GPIO44, U0RXD
6	IO43	I/O	GPIO43, U0TXD
7	IO42	I/O	GPIO42, MTMS
8	IO41	I/O	GPIO41, MTDI
9	IO40	I/O	GPIO40, MTDO
10	IO39	I/O	GPIO39, MTCK
11	IO38	I/O	GPIO38
12	IO37	I/O	GPIO37
13	IO36	I/O	GPIO36
14	IO35	I/O	GPIO35
16	IO34	I/O	GPIO34
17	IO33	I/O	GPIO33
17	IO26	I/O	GPIO26
18	IO21	I/O	GPIO21
19	IO20	I/O	GPIO20, ADC2_CH3, USB_D+
20	IO19	I/O	GPIO19, ADC2_CH3, USB_D-
21	IO18	I/O	GPIO18, ADC2_CH3, DAC_2

P: Power supply; I: Input; O: Output; T: High impedance.

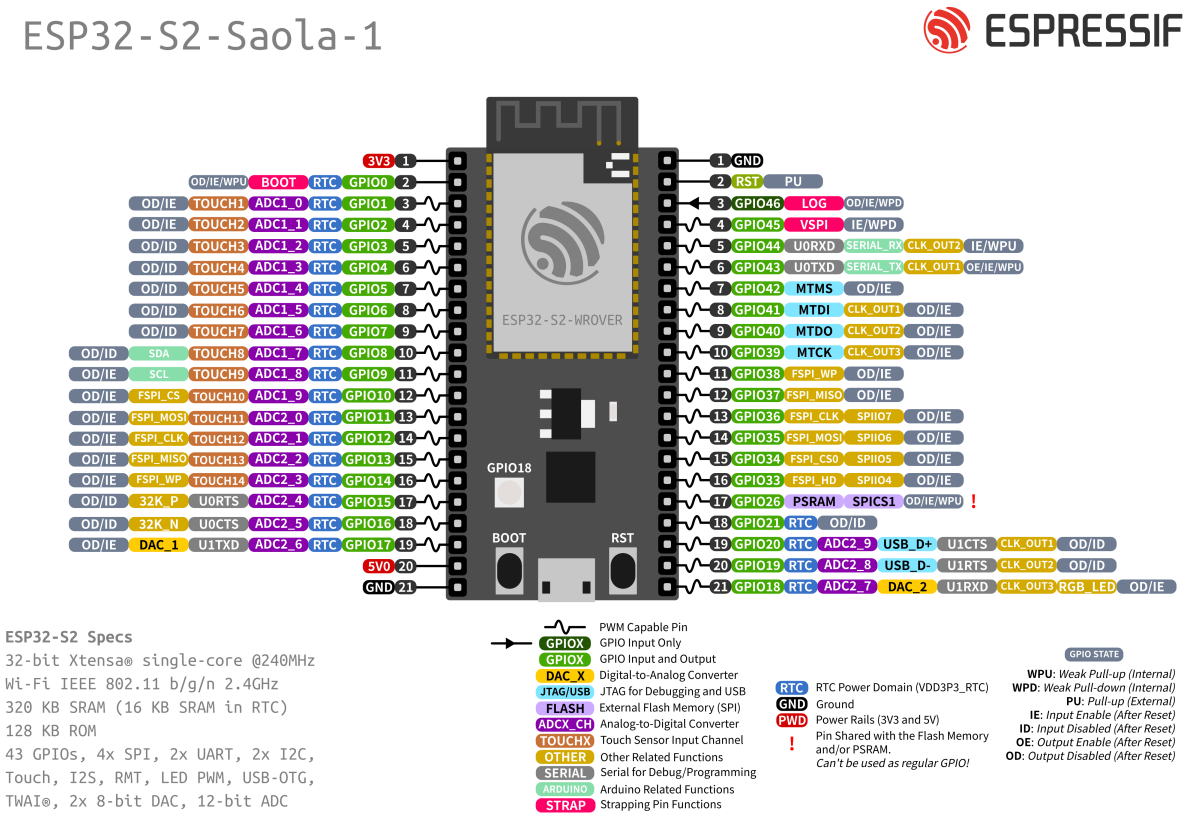
Pin Layout

Strapping Pins

Some of the GPIO's have important features during the booting process. Here is the list of the strapping pins on the ESP32-S2.

GPIO	Default	Function	Pull-up	Pull-down
IO45	Pull-down	Voltage of Internal LDO (VDD_SDIO)	1V8	3V3
IO0	Pull-up	Booting Mode	SPI Boot	Download Boot
IO46	Pull-down	Booting Mode	Don't Care	Download Boot
IO46	Pull-up	Enabling/Disabling Log Print During Booting and Timing of SDIO Slave	U0TXD Active	U0TXD Silent

For more detailed information, see the ESP32-S2 datasheet.



Restricted Usage GPIOS

Some of the GPIO's are used for the external flash and PSRAM. These GPIO's cannot be used:

GPIO	Shared Function
IO26	Connected to PSRAM

Other GPIO's are *INPUT ONLY* and cannot be used as output pin:

GPIO	Function
IO46	GPIO46

Resources

- ESP32-S2 (Datasheet)
- ESP32-S2-WROVER (Datasheet)
- ESP32-S2-Saola-1 (Schematics)

ESP32-C3-DevKitM-1

The ESP32-C3-DevKitM-1 development board is one of Espressif's official boards. This board is based on the ESP32-C3-MINI-1 module, with the ESP32-C3 as the core.

Specifications

- Small sized 2.4 GHz WiFi (802.11 b/g/n) and Bluetooth® 5 module
- Built around ESP32C3 series of SoCs, RISCV singlecore microprocessor
- 4 MB flash in chip package
- 15 available GPIOs (module)
- **Peripherals**
  - 22 × programmable GPIOs
  - Digital interfaces:
  - 3 × SPI
  - 2 × UART
  - 1 × I2C
  - 1 × I2S
  - Remote control peripheral, with 2 transmit channels and 2 receive channels
  - LED PWM controller, with up to 6 channels
  - Full-speed USB Serial/JTAG controller
  - General DMA controller (GDMA), with 3 transmit channels and 3 receive channels
  - 1 × TWAI® controller (compatible with ISO 11898-1)
  - **Analog interfaces:**
    - 2 × 12-bit SAR ADCs, up to 6 channels
    - 1 × temperature sensor
  - **Timers:**
    - 2 × 54-bit general-purpose timers
    - 3 × watchdog timers
    - 1 × 52-bit system timer
- Onboard PCB antenna or external antenna connector

Header Block

Note: Not all of the chip pins are exposed to the pin headers.

J1

No.	Name	Type[1]	Function
1	GND	G	Ground
2	3V3	P	3.3 V power supply
3	3V3	P	3.3 V power supply
4	IO2	I/O/T	GPIO2[2], ADC1_CH2, FSPIQ
5	IO3	I/O/T	GPIO3, ADC1_CH3
6	GND	G	Ground
7	RST	I	CHIP_PU
8	GND	G	Ground
9	IO0	I/O/T	GPIO0, ADC1_CH0, XTAL_32K_P
10	IO1	I/O/T	GPIO1, ADC1_CH1, XTAL_32K_N
11	IO10	I/O/T	GPIO10, FSPICS0
12	GND	G	Ground
13	5V	P	5 V power supply
14	5V	P	5 V power supply
15	GND	G	Ground

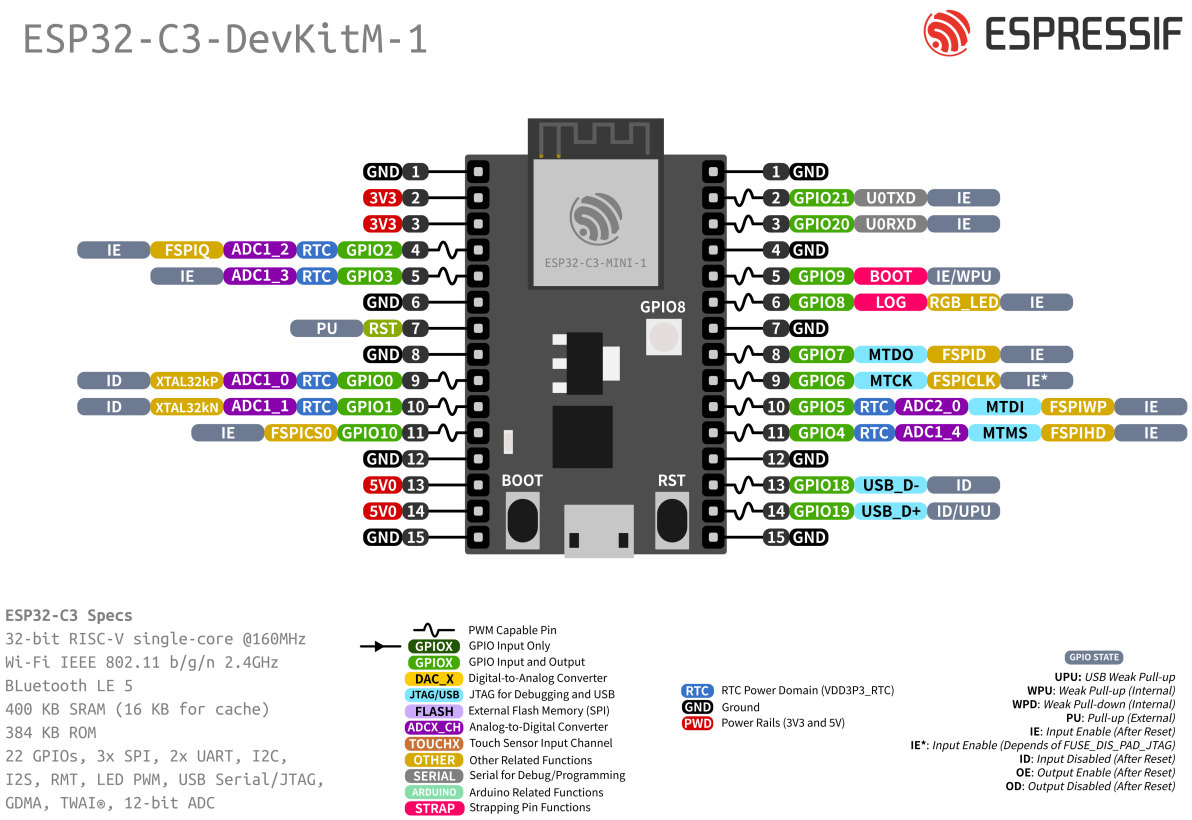
J3

No.	Name	Type[?]	Function
1	GND	G	Ground
2	TX	I/O/T	GPIO21, U0TXD
3	RX	I/O/T	GPIO20, U0RXD
4	GND	G	Ground
5	IO9	I/O/T	GPIO9[2]
6	IO8	I/O/T	GPIO8[2], RGB LED
7	GND	G	Ground
8	IO7	I/O/T	GPIO7, FSPID, MTDO
9	IO6	I/O/T	GPIO6, FSPICLK, MTCK
10	IO5	I/O/T	GPIO5, ADC2_CH0, FSPIWP, MTDI
11	IO4	I/O/T	GPIO4, ADC1_CH4, FSPIHD, MTMS
12	GND	G	Ground
13	IO18	I/O/T	GPIO18, USB_D-
14	IO19	I/O/T	GPIO19, USB_D+
15	GND	G	Ground

[1] P: Power supply; I: Input; O: Output; T: High impedance.

[2] GPIO2, GPIO8, and GPIO9 are strapping pins of the ESP32-C3FN4 chip. During the chip's system reset, the latches of the strapping pins sample the voltage level as strapping bits, and hold these bits until the chip is powered down or shut down.

Pin Layout



Strapping Pins

Some of the GPIO's have important features during the booting process. Here is the list of the strapping pins on the ESP32-C3.

GPIO	Default	Function	Pull-up	Pull-down
IO2	N/A	Booting Mode	See ESP32-C3	See ESP32-C3
IO9	Pull-up	Booting Mode	SPI Boot	Download Boot
IO8	N/A	Booting Mode	Don't Care	Download Boot
IO8	Pull-up	Enabling/Disabling Log Print	See ESP32-C3	See ESP32-C3

For more detailed information, see the ESP32-C3 datasheet.

Resources

- ESP32-C3 (Datasheet)
- ESP32-C3-MINI-1 (Datasheet)

Third Party

Add here the third party boards, listed by vendors.

Note: All the information must be provided by the vendor. If your favorite board is not here, consider creating an issue on GitHub and directly link/mention the vendor in the issue description.

LOLIN

-
-

Generic Vendor

Generic ESP32 Boards

Specifications

Add here the board/kit specifications.

Header Block

Header1

No.	Name	Type	Function
1	3V3	P	3.3 V power supply
2	IO0	I/O	GPIO0, Boot
3	5V0	P	5 V power supply
4	GND	G	Ground

Pin Layout

Add here the pin layout image (not required).

Datasheet

- ESP32 (Datasheet)
- ESP32-S2 (Datasheet)
- ESP32-C3 (Datasheet)
- ESP32-S3 (Datasheet)

Note: Create one file per board or one file with multiple boards. Do not add board information/description on this file.

Datasheet

- ESP32 (Datasheet)
- ESP32-S2 (Datasheet)
- ESP32-C3 (Datasheet)
- ESP32-S3 (Datasheet)

Resources

1.11 Examples

After installing the toolchain into your environment, you will be able to see all the dedicated examples for the ESP32. These examples are located in the examples menu or inside each library folder.

https://github.com/espressif/arduino-esp32/tree/master/libraries

1.12 Datasheet

- ESP32 (Datasheet)
- ESP32-S2 (Datasheet)
- ESP32-C3 (Datasheet)
- ESP32-S3 (Datasheet)

1.13 Resources

CHAPTER

TWO

LIBRARIES

Here is where the Libraries API's descriptions are located:

2.1 Supported Peripherals

Currently, the Arduino ESP32 supports the following peripherals with Arduino APIs.

Peripheral	ESP32	ESP32-S2	ESP32-C3	ESP32-S3		Comments
ADC	Yes	Yes	Yes	Yes		
Bluetooth	Yes	Not Supported	Not Supported	Not Supported		Bluetooth Classic
BLE	Yes	Not Supported	Yes	Yes		
DAC	Yes	Yes	Not Supported	Not Supported		
Ethernet	Yes	Not Supported	Not Supported	Not Supported		(*)
GPIO	Yes	Yes	Yes	Yes		
Hall Sensor	Yes	Not Supported	Not Supported	Not Supported		
I2C	Yes	Yes	Yes	Yes		
I2S	Yes	Yes	Yes	Yes		
LEDC	Yes	Yes	Yes	Yes		
Motor PWM	No	Not Supported	Not Supported	Not Supported		
Pulse Counter	No	No	No	No		
RMT	Yes	Yes	Yes	Yes		
SDIO	No	No	No	No		
SDMMC	Yes	Not Supported	Not Supported	Yes		
Timer	Yes	Yes	Yes	Yes		
Temp. Sensor	Not Supported	Yes	Yes	Yes		
Touch	Yes	Yes	Not Supported	Yes		
TWAI	No	No	No	No		
UART	Yes	Yes	Yes	Yes		
USB	Not Supported	Yes	Yes	Yes		ESP32-C3 only CDC/JTAG
Wi-Fi	Yes	Yes	Yes	Yes		

2.1.1 Notes

(*) SPI Ethernet is supported by all ESP32 families and RMII only for ESP32.

Note: Some peripherals are not available for all ESP32 families. To see more details about it, see the corresponding SoC at Product Selector page.

2.2 Datasheet

- ESP32 (Datasheet)
- ESP32-S2 (Datasheet)
- ESP32-C3 (Datasheet)
- ESP32-S3 (Datasheet)

2.3 APIs

The Arduino ESP32 offers some unique APIs, described in this section:

2.3.1 ADC

About

ADC (analog to digital converter) is a very common peripheral used to convert an analog signal such as voltage to a digital form so that it can be read and processed by a microcontroller.

ADCs are very useful in control and monitoring applications since most sensors (e.g., temperature, pressure, force) produce analogue output voltages.

Note: Each SoC or module has a different number of ADC's with a different number of channels and pins availible. Refer to datasheet of each board for more info.

Arduino-ESP32 ADC API

ADC common API

analogRead

This function is used to get the ADC raw value for a given pin/ADC channel.

```
uint16_t analogRead(uint8_t pin);
```

- `pin` GPIO pin to read analog value

This function will return analog raw value.

analogReadMillivolts

This function is used to get ADC value for a given pin/ADC channel in millivolts.

```
uint32_t analogReadMilliVolts(uint8_t pin);
```

- `pin` GPIO pin to read analog value

This function will return analog value in millivolts.

analogReadResolution

This function is used to set the resolution of `analogRead` return value. Default is 12 bits (range from 0 to 4096) for all chips except ESP32S3 where default is 13 bits (range from 0 to 8192). When different resolution is set, the values read will be shifted to match the given resolution.

Range is 1 - 16 .The default value will be used, if this function is not used.

Note: For the ESP32, the resolution is between 9 to12 and it will change the ADC hardware resolution. Else value will be shifted.

```
void analogReadResolution(uint8_t bits);
```

- `bits` sets analog read resolution

analogSetClockDiv

This function is used to set the divider for the ADC clock.

Range is 1 - 255. Default value is 1.

```
void analogSetClockDiv(uint8_t clockDiv);
```

- `clockDiv` sets the divider for ADC clock.

analogSetAttenuation

This function is used to set the attenuation for all channels.

Input voltages can be attenuated before being input to the ADCs. There are 4 available attenuation options, the higher the attenuation is, the higher the measurable input voltage could be.

The measurable input voltage differs for each chip, see table below for detailed information.

ESP32

Attenuation	Measurable input voltage range
`ADC_ATTEN_DB_0`	100 mV ~ 950 mV
`ADC_ATTEN_DB_2_5`	100 mV ~ 1250 mV
`ADC_ATTEN_DB_6`	150 mV ~ 1750 mV
`ADC_ATTEN_DB_11`	150 mV ~ 2450 mV

ESP32-S2

Attenuation	Measurable input voltage range
ADC_ATTEN_DB_0	0 mV ~ 750 mV
ADC_ATTEN_DB_2_5	0 mV ~ 1050 mV
ADC_ATTEN_DB_6	0 mV ~ 1300 mV
ADC_ATTEN_DB_11	0 mV ~ 2500 mV

ESP32-C3

Attenuation	Measurable input voltage range
ADC_ATTEN_DB_0	0 mV ~ 750 mV
ADC_ATTEN_DB_2_5	0 mV ~ 1050 mV
ADC_ATTEN_DB_6	0 mV ~ 1300 mV
ADC_ATTEN_DB_11	0 mV ~ 2500 mV

ESP32-S3

Attenuation	Measurable input voltage range
ADC_ATTEN_DB_0	0 mV ~ 950 mV
ADC_ATTEN_DB_2_5	0 mV ~ 1250 mV
ADC_ATTEN_DB_6	0 mV ~ 1750 mV
ADC_ATTEN_DB_11	0 mV ~ 3100 mV

```
void analogSetAttenuation(adc_attenuation_t attenuation);
```

- `attenuation` sets the attenuation.

analogSetPinAttenuation

This function is used to set the attenuation for a specific pin/ADC channel. For more information refer to *analogSetAttenuation*.

```
void analogSetPinAttenuation(uint8_t pin, adc_attenuation_t attenuation);
```

- `pin` selects specific pin for attenuation settings.
- `attenuation` sets the attenuation.

adcAttachPin

This function is used to attach the pin to ADC (it will also clear any other analog mode that could be on)

```
bool adcAttachPin(uint8_t pin);
```

This function will return `true` if configuration is successful. Else returns `false`.

ADC API specific for ESP32 chip

analogSetWidth

This function is used to set the hardware sample bits and read resolution. Default is 12bit (0 - 4095). Range is 9 - 12.

```
void analogSetWidth(uint8_t bits);
```

analogSetVRefPin

This function is used to set pin to use for ADC calibration if the esp is not already calibrated (pins 25, 26 or 27).

```
void analogSetVRefPin(uint8_t pin);
```

- `pin` GPIO pin to set VRefPin for ADC calibration

hallRead

This function is used to get the ADC value of the HALL sensor conneted to pins 36(SVP) and 39(SVN).

```
int hallRead();
```

This function will return the hall sensor value.

Example Applications

Here is an example of how to use the ADC.

```
void setup() {
  // initialize serial communication at 115200 bits per second:
  Serial.begin(115200);

  //set the resolution to 12 bits (0-4096)
  analogReadResolution(12);
}

void loop() {
  // read the analog / millivolts value for pin 2:
  int analogValue = analogRead(2);
  int analogVolts = analogReadMilliVolts(2);

  // print out the values you read:
  Serial.printf("ADC analog value = %d\n",analogValue);
  Serial.printf("ADC millivolts value = %d\n",analogVolts);

  delay(100);  // delay in between reads for clear read from serial
}
```

Or you can run Arduino example 01.Basics -> AnalogReadSerial.

2.3.2 BLE

About

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Examples

To get started with BLE, you can try:

BLE Scan

```
/*
   Based on Neil Kolban example for IDF: https://github.com/nkolban/esp32-snippets/blob/
↪master/cpp_utils/tests/BLE%20Tests/SampleScan.cpp
   Ported to Arduino ESP32 by Evandro Copercini
*/

#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>

int scanTime = 5; //In seconds
BLEScan* pBLEScan;

class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice) {
      Serial.printf("Advertised Device: %s \n", advertisedDevice.toString().c_str());
    }
};

void setup() {
  Serial.begin(115200);
  Serial.println("Scanning...");

  BLEDevice::init("");
  pBLEScan = BLEDevice::getScan(); //create new scan
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
  pBLEScan->setActiveScan(true); //active scan uses more power, but get results faster
  pBLEScan->setInterval(100);
  pBLEScan->setWindow(99);  // less or equal setInterval value
}

void loop() {
  // put your main code here, to run repeatedly:
  BLEScanResults foundDevices = pBLEScan->start(scanTime, false);
  Serial.print("Devices found: ");
```

(continues on next page)

(continued from previous page)

```
  Serial.println(foundDevices.getCount());
  Serial.println("Scan done!");
  pBLEScan->clearResults();   // delete results fromBLEScan buffer to release memory
  delay(2000);
}
```

BLE UART

```
/*
    Video: https://www.youtube.com/watch?v=oCMOYS71NIU
    Based on Neil Kolban example for IDF: https://github.com/nkolban/esp32-snippets/blob/
↪master/cpp_utils/tests/BLE%20Tests/SampleNotify.cpp
    Ported to Arduino ESP32 by Evandro Copercini

   Create a BLE server that, once we receive a connection, will send periodic␣
↪notifications.
   The service advertises itself as: 6E400001-B5A3-F393-E0A9-E50E24DCCA9E
   Has a characteristic of: 6E400002-B5A3-F393-E0A9-E50E24DCCA9E - used for receiving␣
↪data with "WRITE"
   Has a characteristic of: 6E400003-B5A3-F393-E0A9-E50E24DCCA9E - used to send data␣
↪with  "NOTIFY"

   The design of creating the BLE server is:
   1. Create a BLE Server
   2. Create a BLE Service
   3. Create a BLE Characteristic on the Service
   4. Create a BLE Descriptor on the characteristic
   5. Start the service.
   6. Start advertising.

   In this example rxValue is the data received (only accessible inside that function).
   And txValue is the data to be sent, in this example just a byte incremented every␣
↪second.
*/
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>

BLEServer *pServer = NULL;
BLECharacteristic * pTxCharacteristic;
bool deviceConnected = false;
bool oldDeviceConnected = false;
uint8_t txValue = 0;

// See the following for generating UUIDs:
// https://www.uuidgenerator.net/

#define SERVICE_UUID           "6E400001-B5A3-F393-E0A9-E50E24DCCA9E" // UART service␣
↪UUID
```

(continues on next page)

(continued from previous page)

```
#define CHARACTERISTIC_UUID_RX "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_TX "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"


class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
      deviceConnected = true;
    };

    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
    }
};

class MyCallbacks: public BLECharacteristicCallbacks {
    void onWrite(BLECharacteristic *pCharacteristic) {
      std::string rxValue = pCharacteristic->getValue();

      if (rxValue.length() > 0) {
        Serial.println("*********");
        Serial.print("Received Value: ");
        for (int i = 0; i < rxValue.length(); i++)
          Serial.print(rxValue[i]);

        Serial.println();
        Serial.println("*********");
      }
    }
};


void setup() {
  Serial.begin(115200);

  // Create the BLE Device
  BLEDevice::init("UART Service");

  // Create the BLE Server
  pServer = BLEDevice::createServer();
  pServer->setCallbacks(new MyServerCallbacks());

  // Create the BLE Service
  BLEService *pService = pServer->createService(SERVICE_UUID);

  // Create a BLE Characteristic
  pTxCharacteristic = pService->createCharacteristic(
                                                      CHARACTERISTIC_
↪UUID_TX,
                                                      BLECharacteristic::PROPI
↪NOTIFY
                                                    );
```

(continues on next page)

(continued from previous page)

```
  pTxCharacteristic->addDescriptor(new BLE2902());

  BLECharacteristic * pRxCharacteristic = pService->createCharacteristic(
                                                                          ␣
↪CHARACTERISTIC_UUID_RX,
                                                                         BLECharacteristi
↪WRITE
                                                                       );

  pRxCharacteristic->setCallbacks(new MyCallbacks());

  // Start the service
  pService->start();

  // Start advertising
  pServer->getAdvertising()->start();
  Serial.println("Waiting a client connection to notify...");
}

void loop() {

    if (deviceConnected) {
        pTxCharacteristic->setValue(&txValue, 1);
        pTxCharacteristic->notify();
        txValue++;
                delay(10); // bluetooth stack will go into congestion, if too many␣
↪packets are sent
        }

    // disconnecting
    if (!deviceConnected && oldDeviceConnected) {
        delay(500); // give the bluetooth stack the chance to get things ready
        pServer->startAdvertising(); // restart advertising
        Serial.println("start advertising");
        oldDeviceConnected = deviceConnected;
    }
    // connecting
    if (deviceConnected && !oldDeviceConnected) {
                // do stuff here on connecting
        oldDeviceConnected = deviceConnected;
    }
}
```

Complete list of BLE examples.

2.3.3 Bluetooth

About

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Examples

To get started with Bluetooth, you can try:

Serial To Serial BT

```
//This example code is in the Public Domain (or CC0 licensed, at your option.)
//By Evandro Copercini - 2018
//
//This example creates a bridge between Serial and Classical Bluetooth (SPP)
//and also demonstrate that SerialBT have the same functionalities of a normal Serial

#include "BluetoothSerial.h"

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

#if !defined(CONFIG_BT_SPP_ENABLED)
#error Serial Bluetooth not available or not enabled. It is only available for the ESP32␣
↪chip.
#endif

BluetoothSerial SerialBT;

void setup() {
  Serial.begin(115200);
  SerialBT.begin("ESP32test"); //Bluetooth device name
  Serial.println("The device started, now you can pair it with bluetooth!");
}

void loop() {
  if (Serial.available()) {
    SerialBT.write(Serial.read());
  }
  if (SerialBT.available()) {
    Serial.write(SerialBT.read());
  }
  delay(20);
}
```

BT Classic Device Discovery

```
#include <BluetoothSerial.h>

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

#if !defined(CONFIG_BT_SPP_ENABLED)
#error Serial Bluetooth not available or not enabled. It is only available for the ESP32␣
↪chip.
#endif

BluetoothSerial SerialBT;


#define BT_DISCOVER_TIME        10000


static bool btScanAsync = true;
static bool btScanSync = true;


void btAdvertisedDeviceFound(BTAdvertisedDevice* pDevice) {
        Serial.printf("Found a device asynchronously: %s\n", pDevice->toString().c_
↪str());
}

void setup() {
  Serial.begin(115200);
  SerialBT.begin("ESP32test"); //Bluetooth device name
  Serial.println("The device started, now you can pair it with bluetooth!");


  if (btScanAsync) {
    Serial.print("Starting discoverAsync...");
    if (SerialBT.discoverAsync(btAdvertisedDeviceFound)) {
      Serial.println("Findings will be reported in \"btAdvertisedDeviceFound\"");
      delay(10000);
      Serial.print("Stopping discoverAsync... ");
      SerialBT.discoverAsyncStop();
      Serial.println("stopped");
    } else {
      Serial.println("Error on discoverAsync f.e. not workin after a \"connect\"");
    }
  }

  if (btScanSync) {
    Serial.println("Starting discover...");
    BTScanResults *pResults = SerialBT.discover(BT_DISCOVER_TIME);
    if (pResults)
      pResults->dump(&Serial);
```

(continues on next page)

(continued from previous page)

```
    else
      Serial.println("Error on BT Scan, no result!");
  }
}

void loop() {
  delay(100);
}
```

Complete list of Bluetooth examples.

2.3.4 DAC

About

DAC (digital to analog converter) is a very common peripheral used to convert a digital signal to an analog form.

ESP32 and ESP32-S2 have two 8-bit DAC channels. The DAC driver allows these channels to be set to arbitrary voltages.

DACs can be used for generating a specific (and dynamic) reference voltage for external sensors, controlling transistors, etc.

ESP32 SoC	DAC_1 pin	DAC_2 pin
ESP32	GPIO 25	GPIO 26
ESP32-S2	GPIO 17	GPIO 18

Arduino-ESP32 DAC API

dacWrite

This function is used to set the DAC value for a given pin/DAC channel.

```
void dacWrite(uint8_t pin, uint8_t value);
```

- `pin` GPIO pin.
- `value` to be set. Range is 0 - 255 (equals 0V - 3.3V).

dacDisable

This function is used to disable DAC output on a given pin/DAC channel.

```
void dacDisable(uint8_t pin);
```

- `pin` GPIO pin.

2.3.5 Deep Sleep

About

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Examples

To get started with Hall sensor, you can try:

ExternalWakeUp

```
/*
Deep Sleep with External Wake Up
=====================================
This code displays how to use deep sleep with
an external trigger as a wake up source and how
to store data in RTC memory to use it over reboots

This code is under Public Domain License.

Hardware Connections
======================
Push Button to GPIO 33 pulled down with a 10K Ohm
resistor

NOTE:
======
Only RTC IO can be used as a source for external wake
source. They are pins: 0,2,4,12-15,25-27,32-39.

Author:
Pranav Cherukupalli <cherukupallip@gmail.com>
*/

#define BUTTON_PIN_BITMASK 0x200000000 // 2^33 in hex

RTC_DATA_ATTR int bootCount = 0;

/*
Method to print the reason by which ESP32
has been awaken from sleep
*/
void print_wakeup_reason(){
  esp_sleep_wakeup_cause_t wakeup_reason;

  wakeup_reason = esp_sleep_get_wakeup_cause();
```

(continues on next page)

(continued from previous page)

```
  switch(wakeup_reason)
  {
    case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wakeup caused by external signal using␣
↪RTC_IO"); break;
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("Wakeup caused by external signal using␣
↪RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wakeup caused by timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wakeup caused by touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("Wakeup caused by ULP program"); break;
    default : Serial.printf("Wakeup was not caused by deep sleep: %d\n",wakeup_reason);␣
↪break;
  }
}

void setup(){
  Serial.begin(115200);
  delay(1000); //Take some time to open up the Serial Monitor

  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));

  //Print the wakeup reason for ESP32
  print_wakeup_reason();

  /*
  First we configure the wake up source
  We set our ESP32 to wake up for an external trigger.
  There are two types for ESP32, ext0 and ext1 .
  ext0 uses RTC_IO to wakeup thus requires RTC peripherals
  to be on while ext1 uses RTC Controller so doesnt need
  peripherals to be powered on.
  Note that using internal pullups/pulldowns also requires
  RTC peripherals to be turned on.
  */
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_33,1); //1 = High, 0 = Low

  //If you were to use ext1, you would use it like
  //esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIGH);

  //Go to sleep now
  Serial.println("Going to sleep now");
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}

void loop(){
  //This is not going to be called
}
```

Timer Wake Up

```
/*
Simple Deep Sleep with Timer Wake Up
=====================================
ESP32 offers a deep sleep mode for effective power
saving as power is an important factor for IoT
applications. In this mode CPUs, most of the RAM,
and all the digital peripherals which are clocked
from APB_CLK are powered off. The only parts of
the chip which can still be powered on are:
RTC controller, RTC peripherals ,and RTC memories

This code displays the most basic deep sleep with
a timer to wake it up and how to store data in
RTC memory to use it over reboots

This code is under Public Domain License.

Author:
Pranav Cherukupalli <cherukupallip@gmail.com>
*/

#define uS_TO_S_FACTOR 1000000ULL  /* Conversion factor for micro seconds to seconds */
#define TIME_TO_SLEEP  5        /* Time ESP32 will go to sleep (in seconds) */

RTC_DATA_ATTR int bootCount = 0;

/*
Method to print the reason by which ESP32
has been awaken from sleep
*/
void print_wakeup_reason(){
  esp_sleep_wakeup_cause_t wakeup_reason;

  wakeup_reason = esp_sleep_get_wakeup_cause();

  switch(wakeup_reason)
  {
    case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wakeup caused by external signal using␣
↪RTC_IO"); break;
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("Wakeup caused by external signal using␣
↪RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wakeup caused by timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wakeup caused by touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("Wakeup caused by ULP program"); break;
    default : Serial.printf("Wakeup was not caused by deep sleep: %d\n",wakeup_reason);␣
↪break;
  }
}

void setup(){
```

(continues on next page)

(continued from previous page)

```
  Serial.begin(115200);
  delay(1000); //Take some time to open up the Serial Monitor

  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));

  //Print the wakeup reason for ESP32
  print_wakeup_reason();

  /*
  First we configure the wake up source
  We set our ESP32 to wake up every 5 seconds
  */
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
  Serial.println("Setup ESP32 to sleep for every " + String(TIME_TO_SLEEP) +
  " Seconds");

  /*
  Next we decide what all peripherals to shut down/keep on
  By default, ESP32 will automatically power down the peripherals
  not needed by the wakeup source, but if you want to be a poweruser
  this is for you. Read in detail at the API docs
  http://esp-idf.readthedocs.io/en/latest/api-reference/system/deep_sleep.html
  Left the line commented as an example of how to configure peripherals.
  The line below turns off all RTC peripherals in deep sleep.
  */
  //esp_deep_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
  //Serial.println("Configured all RTC Peripherals to be powered down in sleep");

  /*
  Now that we have setup a wake cause and if needed setup the
  peripherals state in deep sleep, we can now start going to
  deep sleep.
  In the case that no wake up sources were provided but deep
  sleep was started, it will sleep forever unless hardware
  reset occurs.
  */
  Serial.println("Going to sleep now");
  Serial.flush();
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}

void loop(){
  //This is not going to be called
}
```

2.3.6 ESP-NOW

ESP-NOW is a fast, connectionless communication technology featuring a short packet transmission. ESP-NOW is ideal for smart lights, remote control devices, sensors and other applications.

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Examples

ESP-NOW Master

```
/**
   ESPNOW - Basic communication - Master
   Date: 26th September 2017
   Author: Arvind Ravulavaru <https://github.com/arvindr21>
   Purpose: ESPNow Communication between a Master ESP32 and a Slave ESP32
   Description: This sketch consists of the code for the Master module.
   Resources: (A bit outdated)
   a. https://espressif.com/sites/default/files/documentation/esp-now_user_guide_en.pdf
   b. http://www.esploradores.com/practica-6-conexion-esp-now/

   << This Device Master >>

   Flow: Master
   Step 1 : ESPNow Init on Master and set it in STA mode
   Step 2 : Start scanning for Slave ESP32 (we have added a prefix of `slave` to the SSID␣
↪of slave for an easy setup)
   Step 3 : Once found, add Slave as peer
   Step 4 : Register for send callback
   Step 5 : Start Transmitting data from Master to Slave

   Flow: Slave
   Step 1 : ESPNow Init on Slave
   Step 2 : Update the SSID of Slave with a prefix of `slave`
   Step 3 : Set Slave in AP mode
   Step 4 : Register for receive callback and wait for data
   Step 5 : Once data arrives, print it in the serial monitor

   Note: Master and Slave have been defined to easily understand the setup.
         Based on the ESPNOW API, there is no concept of Master and Slave.
         Any devices can act as master or salve.
*/

#include <esp_now.h>
#include <WiFi.h>
#include <esp_wifi.h> // only for esp_wifi_set_channel()

// Global copy of slave
esp_now_peer_info_t slave;
```

(continues on next page)

(continued from previous page)

```
#define CHANNEL 1
#define PRINTSCANRESULTS 0
#define DELETEBEFOREPAIR 0

// Init ESP Now with fallback
void InitESPNow() {
  WiFi.disconnect();
  if (esp_now_init() == ESP_OK) {
    Serial.println("ESPNow Init Success");
  }
  else {
    Serial.println("ESPNow Init Failed");
    // Retry InitESPNow, add a counte and then restart?
    // InitESPNow();
    // or Simply Restart
    ESP.restart();
  }
}

// Scan for slaves in AP mode
void ScanForSlave() {
  int16_t scanResults = WiFi.scanNetworks(false, false, false, 300, CHANNEL); // Scan␣
↪only on one channel
  // reset on each scan
  bool slaveFound = 0;
  memset(&slave, 0, sizeof(slave));

  Serial.println("");
  if (scanResults == 0) {
    Serial.println("No WiFi devices in AP Mode found");
  } else {
    Serial.print("Found "); Serial.print(scanResults); Serial.println(" devices ");
    for (int i = 0; i < scanResults; ++i) {
      // Print SSID and RSSI for each device found
      String SSID = WiFi.SSID(i);
      int32_t RSSI = WiFi.RSSI(i);
      String BSSIDstr = WiFi.BSSIDstr(i);

      if (PRINTSCANRESULTS) {
        Serial.print(i + 1);
        Serial.print(": ");
        Serial.print(SSID);
        Serial.print(" (");
        Serial.print(RSSI);
        Serial.print(")");
        Serial.println("");
      }
      delay(10);
      // Check if the current device starts with `Slave`
      if (SSID.indexOf("Slave") == 0) {
        // SSID of interest
        Serial.println("Found a Slave.");
```

(continues on next page)

(continued from previous page)

```
        Serial.print(i + 1); Serial.print(": "); Serial.print(SSID); Serial.print(" [");␣
↪Serial.print(BSSIDstr); Serial.print("]"); Serial.print(" ("); Serial.print(RSSI);␣
↪Serial.print(")"); Serial.println("");
        // Get BSSID => Mac Address of the Slave
        int mac[6];
        if ( 6 == sscanf(BSSIDstr.c_str(), "%x:%x:%x:%x:%x:%x",  &mac[0], &mac[1], &
↪mac[2], &mac[3], &mac[4], &mac[5] ) ) {
          for (int ii = 0; ii < 6; ++ii ) {
            slave.peer_addr[ii] = (uint8_t) mac[ii];
          }
        }

        slave.channel = CHANNEL; // pick a channel
        slave.encrypt = 0; // no encryption

        slaveFound = 1;
        // we are planning to have only one slave in this example;
        // Hence, break after we find one, to be a bit efficient
        break;
      }
    }
  }

  if (slaveFound) {
    Serial.println("Slave Found, processing..");
  } else {
    Serial.println("Slave Not Found, trying again.");
  }

  // clean up ram
  WiFi.scanDelete();
}

// Check if the slave is already paired with the master.
// If not, pair the slave with master
bool manageSlave() {
  if (slave.channel == CHANNEL) {
    if (DELETEBEFOREPAIR) {
      deletePeer();
    }

    Serial.print("Slave Status: ");
    // check if the peer exists
    bool exists = esp_now_is_peer_exist(slave.peer_addr);
    if ( exists) {
      // Slave already paired.
      Serial.println("Already Paired");
      return true;
    } else {
      // Slave not paired, attempt pair
      esp_err_t addStatus = esp_now_add_peer(&slave);
      if (addStatus == ESP_OK) {
```

(continues on next page)

(continued from previous page)

```
        // Pair success
        Serial.println("Pair success");
        return true;
      } else if (addStatus == ESP_ERR_ESPNOW_NOT_INIT) {
        // How did we get so far!!
        Serial.println("ESPNOW Not Init");
        return false;
      } else if (addStatus == ESP_ERR_ESPNOW_ARG) {
        Serial.println("Invalid Argument");
        return false;
      } else if (addStatus == ESP_ERR_ESPNOW_FULL) {
        Serial.println("Peer list full");
        return false;
      } else if (addStatus == ESP_ERR_ESPNOW_NO_MEM) {
        Serial.println("Out of memory");
        return false;
      } else if (addStatus == ESP_ERR_ESPNOW_EXIST) {
        Serial.println("Peer Exists");
        return true;
      } else {
        Serial.println("Not sure what happened");
        return false;
      }
    }
  } else {
    // No slave found to process
    Serial.println("No Slave found to process");
    return false;
  }
}

void deletePeer() {
  esp_err_t delStatus = esp_now_del_peer(slave.peer_addr);
  Serial.print("Slave Delete Status: ");
  if (delStatus == ESP_OK) {
    // Delete success
    Serial.println("Success");
  } else if (delStatus == ESP_ERR_ESPNOW_NOT_INIT) {
    // How did we get so far!!
    Serial.println("ESPNOW Not Init");
  } else if (delStatus == ESP_ERR_ESPNOW_ARG) {
    Serial.println("Invalid Argument");
  } else if (delStatus == ESP_ERR_ESPNOW_NOT_FOUND) {
    Serial.println("Peer not found.");
  } else {
    Serial.println("Not sure what happened");
  }
}

uint8_t data = 0;
// send data
void sendData() {
```

(continues on next page)

(continued from previous page)

```
  data++;
  const uint8_t *peer_addr = slave.peer_addr;
  Serial.print("Sending: "); Serial.println(data);
  esp_err_t result = esp_now_send(peer_addr, &data, sizeof(data));
  Serial.print("Send Status: ");
  if (result == ESP_OK) {
    Serial.println("Success");
  } else if (result == ESP_ERR_ESPNOW_NOT_INIT) {
    // How did we get so far!!
    Serial.println("ESPNOW not Init.");
  } else if (result == ESP_ERR_ESPNOW_ARG) {
    Serial.println("Invalid Argument");
  } else if (result == ESP_ERR_ESPNOW_INTERNAL) {
    Serial.println("Internal Error");
  } else if (result == ESP_ERR_ESPNOW_NO_MEM) {
    Serial.println("ESP_ERR_ESPNOW_NO_MEM");
  } else if (result == ESP_ERR_ESPNOW_NOT_FOUND) {
    Serial.println("Peer not found.");
  } else {
    Serial.println("Not sure what happened");
  }
}

// callback when data is sent from Master to Slave
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  char macStr[18];
  snprintf(macStr, sizeof(macStr), "%02x:%02x:%02x:%02x:%02x:%02x",
           mac_addr[0], mac_addr[1], mac_addr[2], mac_addr[3], mac_addr[4], mac_addr[5]);
  Serial.print("Last Packet Sent to: "); Serial.println(macStr);
  Serial.print("Last Packet Send Status: "); Serial.println(status == ESP_NOW_SEND_
↪SUCCESS ? "Delivery Success" : "Delivery Fail");
}

void setup() {
  Serial.begin(115200);
  //Set device in STA mode to begin with
  WiFi.mode(WIFI_STA);
  esp_wifi_set_channel(CHANNEL, WIFI_SECOND_CHAN_NONE);
  Serial.println("ESPNow/Basic/Master Example");
  // This is the mac address of the Master in Station Mode
  Serial.print("STA MAC: "); Serial.println(WiFi.macAddress());
  Serial.print("STA CHANNEL "); Serial.println(WiFi.channel());
  // Init ESPNow with a fallback logic
  InitESPNow();
  // Once ESPNow is successfully Init, we will register for Send CB to
  // get the status of Trasnmitted packet
  esp_now_register_send_cb(OnDataSent);
}

void loop() {
  // In the loop we scan for slave
  ScanForSlave();
```

(continues on next page)

(continued from previous page)

```
  // If Slave is found, it would be populate in `slave` variable
  // We will check if `slave` is defined and then we proceed further
  if (slave.channel == CHANNEL) { // check if slave channel is defined
    // `slave` is defined
    // Add slave as peer if it has not been added already
    bool isPaired = manageSlave();
    if (isPaired) {
      // pair success or already paired
      // Send data to device
      sendData();
    } else {
      // slave pair failed
      Serial.println("Slave pair failed!");
    }
  }
  else {
    // No slave found to process
  }

  // wait for 3seconds to run the logic again
  delay(3000);
}
```

ESP-NOW Slave

```
/**
   ESPNOW - Basic communication - Slave
   Date: 26th September 2017
   Author: Arvind Ravulavaru <https://github.com/arvindr21>
   Purpose: ESPNow Communication between a Master ESP32 and a Slave ESP32
   Description: This sketch consists of the code for the Slave module.
   Resources: (A bit outdated)
   a. https://espressif.com/sites/default/files/documentation/esp-now_user_guide_en.pdf
   b. http://www.esploradores.com/practica-6-conexion-esp-now/

   << This Device Slave >>

   Flow: Master
   Step 1 : ESPNow Init on Master and set it in STA mode
   Step 2 : Start scanning for Slave ESP32 (we have added a prefix of `slave` to the SSID␣
↪of slave for an easy setup)
   Step 3 : Once found, add Slave as peer
   Step 4 : Register for send callback
   Step 5 : Start Transmitting data from Master to Slave

   Flow: Slave
   Step 1 : ESPNow Init on Slave
   Step 2 : Update the SSID of Slave with a prefix of `slave`
   Step 3 : Set Slave in AP mode
   Step 4 : Register for receive callback and wait for data
```

(continues on next page)

(continued from previous page)

```
   Step 5 : Once data arrives, print it in the serial monitor

   Note: Master and Slave have been defined to easily understand the setup.
         Based on the ESPNOW API, there is no concept of Master and Slave.
         Any devices can act as master or salve.
*/

#include <esp_now.h>
#include <WiFi.h>

#define CHANNEL 1

// Init ESP Now with fallback
void InitESPNow() {
  WiFi.disconnect();
  if (esp_now_init() == ESP_OK) {
    Serial.println("ESPNow Init Success");
  }
  else {
    Serial.println("ESPNow Init Failed");
    // Retry InitESPNow, add a counte and then restart?
    // InitESPNow();
    // or Simply Restart
    ESP.restart();
  }
}

// config AP SSID
void configDeviceAP() {
  const char *SSID = "Slave_1";
  bool result = WiFi.softAP(SSID, "Slave_1_Password", CHANNEL, 0);
  if (!result) {
    Serial.println("AP Config failed.");
  } else {
    Serial.println("AP Config Success. Broadcasting with AP: " + String(SSID));
    Serial.print("AP CHANNEL "); Serial.println(WiFi.channel());
  }
}

void setup() {
  Serial.begin(115200);
  Serial.println("ESPNow/Basic/Slave Example");
  //Set device in AP mode to begin with
  WiFi.mode(WIFI_AP);
  // configure device AP mode
  configDeviceAP();
  // This is the mac address of the Slave in AP Mode
  Serial.print("AP MAC: "); Serial.println(WiFi.softAPmacAddress());
  // Init ESPNow with a fallback logic
  InitESPNow();
  // Once ESPNow is successfully Init, we will register for recv CB to
  // get recv packer info.
```

(continues on next page)

(continued from previous page)

```
  esp_now_register_recv_cb(OnDataRecv);
}

// callback when data is recv from Master
void OnDataRecv(const uint8_t *mac_addr, const uint8_t *data, int data_len) {
  char macStr[18];
  snprintf(macStr, sizeof(macStr), "%02x:%02x:%02x:%02x:%02x:%02x",
           mac_addr[0], mac_addr[1], mac_addr[2], mac_addr[3], mac_addr[4], mac_addr[5]);
  Serial.print("Last Packet Recv from: "); Serial.println(macStr);
  Serial.print("Last Packet Recv Data: "); Serial.println(*data);
  Serial.println("");
}

void loop() {
  // Chill
}
```

Resources

- ESP-NOW (User Guide)

2.3.7 Ethernet

About

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Examples

To get started with Ethernet, you can try:

LAN8720

```
/*
    This sketch shows the Ethernet event usage

*/

#include <ETH.h>

static bool eth_connected = false;

void WiFiEvent(WiFiEvent_t event)
{
  switch (event) {
```

(continues on next page)

(continued from previous page)

```
    case ARDUINO_EVENT_ETH_START:
      Serial.println("ETH Started");
      //set eth hostname here
      ETH.setHostname("esp32-ethernet");
      break;
    case ARDUINO_EVENT_ETH_CONNECTED:
      Serial.println("ETH Connected");
      break;
    case ARDUINO_EVENT_ETH_GOT_IP:
      Serial.print("ETH MAC: ");
      Serial.print(ETH.macAddress());
      Serial.print(", IPv4: ");
      Serial.print(ETH.localIP());
      if (ETH.fullDuplex()) {
        Serial.print(", FULL_DUPLEX");
      }
      Serial.print(", ");
      Serial.print(ETH.linkSpeed());
      Serial.println("Mbps");
      eth_connected = true;
      break;
    case ARDUINO_EVENT_ETH_DISCONNECTED:
      Serial.println("ETH Disconnected");
      eth_connected = false;
      break;
    case ARDUINO_EVENT_ETH_STOP:
      Serial.println("ETH Stopped");
      eth_connected = false;
      break;
    default:
      break;
  }
}

void testClient(const char * host, uint16_t port)
{
  Serial.print("\nconnecting to ");
  Serial.println(host);

  WiFiClient client;
  if (!client.connect(host, port)) {
    Serial.println("connection failed");
    return;
  }
  client.printf("GET / HTTP/1.1\r\nHost: %s\r\n\r\n", host);
  while (client.connected() && !client.available());
  while (client.available()) {
    Serial.write(client.read());
  }

  Serial.println("closing connection\n");
  client.stop();
```

(continues on next page)

(continued from previous page)

```
}

void setup()
{
  Serial.begin(115200);
  WiFi.onEvent(WiFiEvent);
  ETH.begin();
}


void loop()
{
  if (eth_connected) {
    testClient("google.com", 80);
  }
  delay(10000);
}
```

TLK110

```
/*
    This sketch shows the Ethernet event usage

*/

#include <ETH.h>

#define ETH_ADDR        31
#define ETH_POWER_PIN   17
#define ETH_MDC_PIN     23
#define ETH_MDIO_PIN    18
#define ETH_TYPE        ETH_PHY_TLK110

static bool eth_connected = false;

void WiFiEvent(WiFiEvent_t event)
{
  switch (event) {
    case ARDUINO_EVENT_ETH_START:
      Serial.println("ETH Started");
      //set eth hostname here
      ETH.setHostname("esp32-ethernet");
      break;
    case ARDUINO_EVENT_ETH_CONNECTED:
      Serial.println("ETH Connected");
      break;
    case ARDUINO_EVENT_ETH_GOT_IP:
      Serial.print("ETH MAC: ");
      Serial.print(ETH.macAddress());
      Serial.print(", IPv4: ");
```

(continues on next page)

(continued from previous page)

```
      Serial.print(ETH.localIP());
      if (ETH.fullDuplex()) {
        Serial.print(", FULL_DUPLEX");
      }
      Serial.print(", ");
      Serial.print(ETH.linkSpeed());
      Serial.println("Mbps");
      eth_connected = true;
      break;
    case ARDUINO_EVENT_ETH_DISCONNECTED:
      Serial.println("ETH Disconnected");
      eth_connected = false;
      break;
    case ARDUINO_EVENT_ETH_STOP:
      Serial.println("ETH Stopped");
      eth_connected = false;
      break;
    default:
      break;
  }
}

void testClient(const char * host, uint16_t port)
{
  Serial.print("\nconnecting to ");
  Serial.println(host);

  WiFiClient client;
  if (!client.connect(host, port)) {
    Serial.println("connection failed");
    return;
  }
  client.printf("GET / HTTP/1.1\r\nHost: %s\r\n\r\n", host);
  while (client.connected() && !client.available());
  while (client.available()) {
    Serial.write(client.read());
  }

  Serial.println("closing connection\n");
  client.stop();
}

void setup()
{
  Serial.begin(115200);
  WiFi.onEvent(WiFiEvent);
  ETH.begin(ETH_ADDR, ETH_POWER_PIN, ETH_MDC_PIN, ETH_MDIO_PIN, ETH_TYPE);
}


void loop()
{
```

(continues on next page)

(continued from previous page)

```
  if (eth_connected) {
    testClient("google.com", 80);
  }
  delay(10000);
}
```

Complete list of Ethernet examples.

2.3.8 GPIO

About

One of the most used and versatile peripheral in a microcontroller is the GPIO. The GPIO is commonly used to write and read the pin state.

GPIO stands to General Purpose Input Output, and is responsible to control or read the state of a specific pin in the digital world. For example, this peripheral is widely used to create the LED blinking or to read a simple button.

Note: There are some GPIOs with special restrictions, and not all GPIOs are accessible through the developemnt board. For more information about it, see the corresponding board pin layout information.

GPIOs Modes

There are two different modes in the GPIO configuration:

- **Input Mode**

In this mode, the GPIO will receive the digital state from a specific device. This device could be a button or a switch.

- **Output Mode**

For the output mode, the GPIO will change the GPIO digital state to a specific device. You can drive an LED for example.

GPIO API

Here is the common functions used for the GPIO peripheral.

pinMode

The `pinMode` function is used to define the GPIO operation mode for a specific pin.

```
void pinMode(uint8_t pin, uint8_t mode);
```

- `pin` defines the GPIO pin number.
- `mode` sets operation mode.

The following modes are supported for the basic *input* and *output*:

- **INPUT** sets the GPIO as input without pullup or pulldown (high impedance).

- **OUTPUT** sets the GPIO as output/read mode.
- **INPUT_PULLDOWN** sets the GPIO as input with the internal pulldown.
- **INPUT_PULLUP** sets the GPIO as input with the internal pullup.

Internal Pullup and Pulldown

The ESP32 SoC families supports the internal pullup and pulldown throught a 45kR resistor, that can be enabled when configuring the GPIO mode as `INPUT` mode. If the pullup or pulldown mode is not defined, the pin will stay in the high impedance mode.

digitalWrite

The function `digitalWrite` sets the state of the selected GPIO to `HIGH` or `LOW`. This function is only used if the `pinMode` was configured as `OUTPUT`.

```
void digitalWrite(uint8_t pin, uint8_t val);
```

- `pin` defines the GPIO pin number.
- `val` set the output digital state to `HIGH` or `LOW`.

digitalRead

To read the state of a given pin configured as `INPUT`, the function `digitalRead` is used.

```
int digitalRead(uint8_t pin);
```

- `pin` select GPIO

This function will return the logical state of the selected pin as `HIGH` or `LOW`.

Interrupts

The GPIO peripheral on the ESP32 supports interruptions.

attachInterrupt

The function `attachInterruptArg` is used to attach the interrupt to the defined pin.

```
attachInterrupt(uint8_t pin, voidFuncPtr handler, int mode);
```

- `pin` defines the GPIO pin number.
- `handler` set the handler function.
- `mode` set the interrupt mode.

Here are the supported interrupt modes:

- **DISABLED**
- **RISING**

- **FALLING**
- **CHANGE**
- **ONLOW**
- **ONHIGH**
- **ONLOW_WE**
- **ONHIGH_WE**

attachInterruptArg

The function `attachInterruptArg` is used to attach the interrupt to the defined pin using arguments.

```
attachInterruptArg(uint8_t pin, voidFuncPtrArg handler, void * arg, int mode);
```

- `pin` defines the GPIO pin number.
- `handler` set the handler function.
- `arg` pointer to the interrupt arguments.
- `mode` set the interrupt mode.

detachInterrupt

To detach the interruption from a specific pin, use the `detachInterrupt` function giving the GPIO to be detached.

```
detachInterrupt(uint8_t pin);
```

- `pin` defines the GPIO pin number.

Example Code

GPIO Input and Output Modes

```
#define LED    12
#define BUTTON 2

uint8_t stateLED = 0;

  void setup() {
      pinMode(LED, OUTPUT);
      pinMode(BUTTON,INPUT_PULLUP);
  }

  void loop() {

     if(!digitalRead(BUTTON)){
       stateLED = stateLED^1;
      digitalWrite(LED,stateLED);
    }
  }
```

GPIO Interrupt

```
#include <Arduino.h>

struct Button {
    const uint8_t PIN;
    uint32_t numberKeyPresses;
    bool pressed;
};

Button button1 = {23, 0, false};
Button button2 = {18, 0, false};

void ARDUINO_ISR_ATTR isr(void* arg) {
    Button* s = static_cast<Button*>(arg);
    s->numberKeyPresses += 1;
    s->pressed = true;
}

void ARDUINO_ISR_ATTR isr() {
    button2.numberKeyPresses += 1;
    button2.pressed = true;
}

void setup() {
    Serial.begin(115200);
    pinMode(button1.PIN, INPUT_PULLUP);
    attachInterruptArg(button1.PIN, isr, &button1, FALLING);
    pinMode(button2.PIN, INPUT_PULLUP);
    attachInterrupt(button2.PIN, isr, FALLING);
}

void loop() {
    if (button1.pressed) {
        Serial.printf("Button 1 has been pressed %u times\n", button1.numberKeyPresses);
        button1.pressed = false;
    }
    if (button2.pressed) {
        Serial.printf("Button 2 has been pressed %u times\n", button2.numberKeyPresses);
        button2.pressed = false;
    }
    static uint32_t lastMillis = 0;
    if (millis() - lastMillis > 10000) {
      lastMillis = millis();
      detachInterrupt(button1.PIN);
    }
}
```

2.3.9 Hall Sensor

About

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Example

To get started with Hall sensor, you can try:

Hall Sensor

```
//Simple sketch to access the internal hall effect detector on the esp32.
//values can be quite low.
//Brian Degger / @sctv
int val = 0;
void setup() {
  Serial.begin(9600);
    }

void loop() {
  // put your main code here, to run repeatedly:
  val = hallRead();
  // print the results to the serial monitor:
  //Serial.print("sensor = ");
  Serial.println(val);//to graph
}
```

2.3.10 I2C

About

I2C (Inter-Integrated Circuit) / TWI (Two-wire Interface) is a widely used serial communication to connect devices in a short distance. This is one of the most common peripherals used to connect sensors, EEPROMs, RTC, ADC, DAC, displays, OLED, and many other devices and microcontrollers.

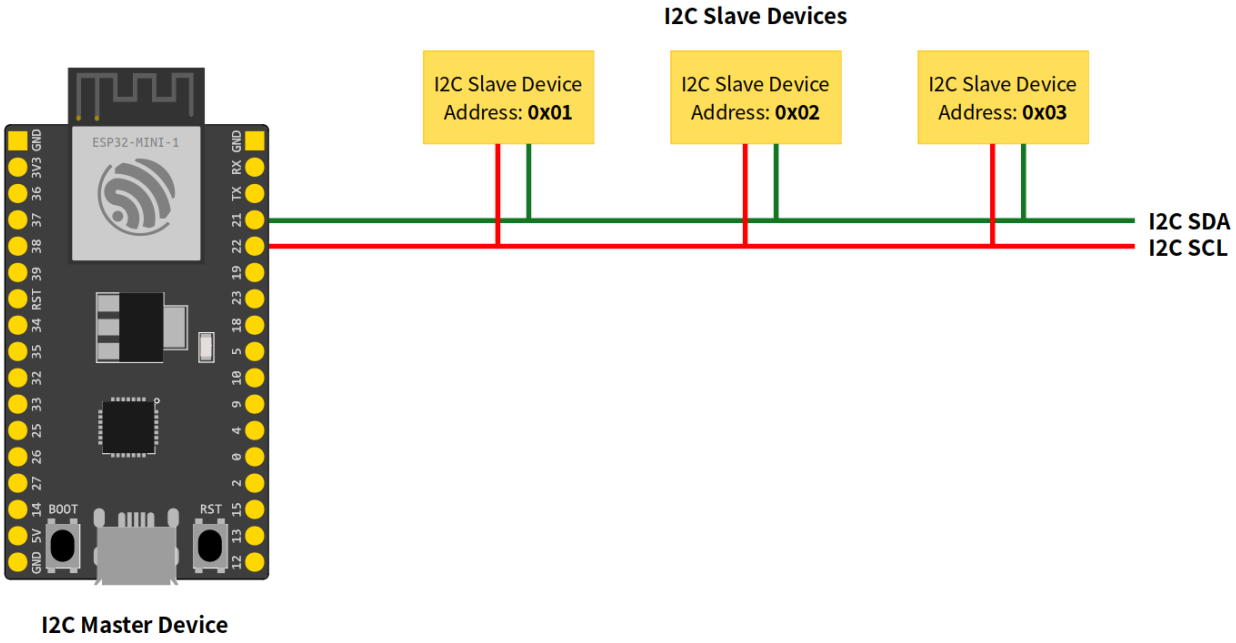
This serial communication is considered as a low-speed bus, and multiple devices can be connected on the same two-wires bus, each with a unique 7-bits address (up to 128 devices). These two wires are called SDA (serial data line) and SCL (serial clock line).

Note: The SDA and SCL lines require pull-up resistors. See the device datasheet for more details about the resistors' values and the operating voltage.

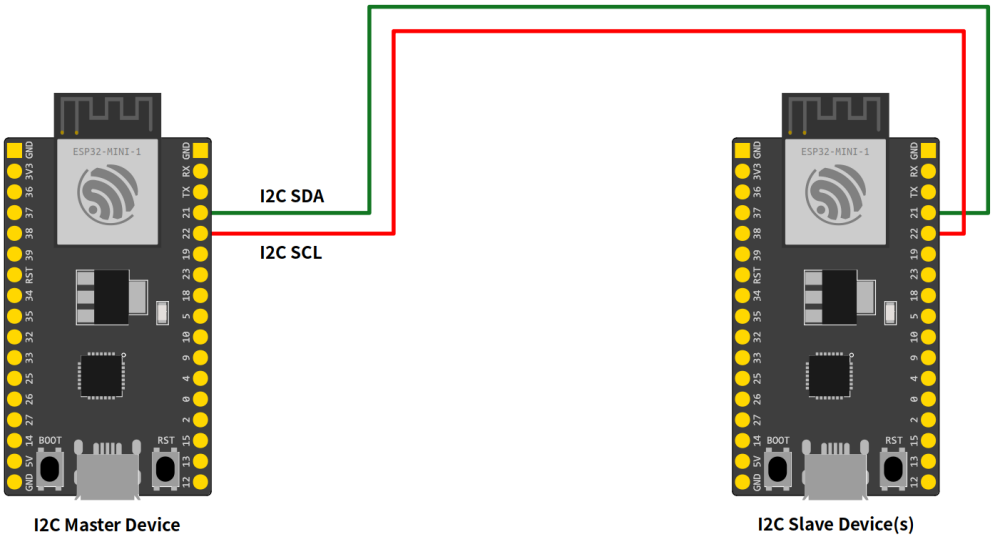
I2C Modes

The I2C can be used in two different modes:

- **I2C Master Mode**
  - In this mode, the ESP32 generates the clock signal and initiates the communication with the slave device.



- **I2C Slave Mode**
  - The slave mode, the clock is generated by the master device and responds to the master if the destination address is the same as the destination.



Arduino-ESP32 I2C API

The ESP32 I2C library is based on the Arduino Wire Library and implements a few more APIs, described in this documentation.

I2C Common API

Here are the common functions used for master and slave modes.

begin

This function is used to start the peripheral using the default configuration.

```
bool begin();
```

This function will return `true` if the peripheral was initialized correctly.

setPins

This function is used to define the `SDA` and `SCL` pins.

Note: Call this function before `begin` to change the pins from the default ones.

```
bool setPins(int sdaPin, int sclPin);
```

- `sdaPin` sets the GPIO to be used as the I2C peripheral data line.
- `sclPin` sets the GPIO to be used as the I2C peripheral clock line.

The default pins may vary from board to board. On the *Generic ESP32* the default I2C pins are:

- `sdaPin` **GPIO21**
- `sclPin` **GPIO22**

This function will return `true` if the peripheral was configured correctly.

setClock

Use this function to set the bus clock. The default value will be used if this function is not used.

```
bool setClock(uint32_t frequency);
```

- `frequency` sets the bus frequency clock.

This function will return `true` if the clock was configured correctly.

getClock

Use this function to get the bus clock.

```
uint32_t getClock();
```

This function will return the current frequency configuration.

setTimeOut

Set the bus timeout given in milliseconds. The default value is 50ms.

```
void setTimeOut(uint16_t timeOutMillis);
```

- `timeOutMillis` sets the timeout in ms.

getTimeOut

Get the bus timeout in milliseconds.

```
uint16_t getTimeOut();
```

This function will return the current timeout configuration.

write

This function writes data to the buffer.

```
size_t write(uint8_t);
```

or

```
size_t write(const uint8_t *, size_t);
```

The return will be the size of the data added to the buffer.

end

This function will finish the communication and release all the allocated resources. After calling `end` you need to use `begin` again in order to initialize the I2C driver again.

```
bool end();
```

I2C Master Mode

This mode is used to initiate communication to the slave.

Basic Usage

To start using I2C master mode on the Arduino, the first step is to include the `Wire.h` header to the sketch.

```
#include "Wire.h"
```

Now, we can start the peripheral configuration by calling `begin` function.

```
Wire.begin();
```

By using `begin` without any arguments, all the settings will be done by using the default values. To set the values by your own, see the function description. This function is described here: *i2c begin*

After calling `begin`, we can start the transmission by calling `beginTransmission` and passing the I2C slave address:

```
Wire.beginTransmission(I2C_DEV_ADDR);
```

To write some bytes to the slave, use the `write` function.

```
Wire.write(x);
```

You can pass different data types using `write` function. This function is described here: *i2c write*

Note: The `write` function does not write directly to the slave device but adds to the I2C buffer. To do so, you need to use the `endTransmission` function to send the buffered bytes to the slave device.

```
Wire.endTransmission(true);
```

After calling `endTransmission`, the data stored in the I2C buffer will be transmitted to the slave device.

Now you can request a reading from the slave device. The `requestFrom` will ask for a readout to the selected device by giving the address and the size.

```
Wire.requestFrom(I2C_DEV_ADDR, SIZE);
```

and the `readBytes` will read it.

```
Wire.readBytes(temp, error);
```

I2C Master APIs

Here are the I2C master APIs. These function are intended to be used only for master mode.

begin

In master mode, the `begin` function can be used by passing the **pins** and **bus frequency**. Use this function only for the master mode.

```
bool begin(int sdaPin, int sclPin, uint32_t frequency)
```

Alternatively, you can use the `begin` function without any argument to use all default values.

This function will return `true` if the peripheral was initialized correctly.

beginTransmission

This function is used to star a communication process with the slave device. Call this function by passing the slave `address` before writing the message to the buffer.

```
void beginTransmission(uint16_t address)
```

endTransmission

After writing to the buffer using *i2c write*, use the function `endTransmission` to send the message to the slave device address defined on the `beginTransmission` function.

```
uint8_t endTransmission(bool sendStop);
```

- `sendStop` enables **(true)** or disables **(false)** the stop signal *(only used in master mode)*.

Calling the this function without `sendStop` is equivalent to `sendStop = true`.

```
uint8_t endTransmission(void);
```

This function will return the error code.

requestFrom

To read from the slave device, use the `requestFrom` function.

```
uint8_t requestFrom(uint16_t address, uint8_t size, bool sendStop)
```

- `address` set the device address.
- `size` define the size to be requested.
- `sendStop` enables (true) or disables (false) the stop signal.

This function will return the number of bytes read from the device.

Example Application - WireMaster.ino

Here is an example of how to use the I2C in Master Mode.

```
#include "Wire.h"

#define I2C_DEV_ADDR 0x55

uint32_t i = 0;

void setup() {
  Serial.begin(115200);
  Serial.setDebugOutput(true);
  Wire.begin();
}

void loop() {
  delay(5000);

  //Write message to the slave
  Wire.beginTransmission(I2C_DEV_ADDR);
  Wire.printf("Hello World! %u", i++);
  uint8_t error = Wire.endTransmission(true);
  Serial.printf("endTransmission: %u\n", error);

  //Read 16 bytes from the slave
  uint8_t bytesReceived = Wire.requestFrom(I2C_DEV_ADDR, 16);
  Serial.printf("requestFrom: %u\n", bytesReceived);
  if((bool)bytesReceived){ //If received more than zero bytes
    uint8_t temp[bytesReceived];
    Wire.readBytes(temp, bytesReceived);
    log_print_buf(temp, bytesReceived);
  }
}
```

I2C Slave Mode

This mode is used to accept communication from the master.

Basic Usage

To start using I2C as slave mode on the Arduino, the first step is to include the `Wire.h` header to the scketch.

```
#include "Wire.h"
```

Before calling `begin` we must create two callback functions to handle the communication with the master device.

```
Wire.onReceive(onReceive);
```

and

```
Wire.onRequest(onRequest);
```

The `onReceive` will handle the request from the master device uppon a slave read request and the `onRequest` will handle the answer to the master.

Now, we can start the peripheral configuration by calling `begin` function with the device address.

```
Wire.begin((uint8_t)I2C_DEV_ADDR);
```

By using `begin` without any arguments, all the settings will be done by using the default values. To set the values by your own, see the function description. This function is described here: *i2c begin*

For ESP32 only!

Use the function `slaveWrite` in order to pre-write to the slave response buffer. This is used only for the ESP32 in order to add the slave capability on the chip and keep compatability with Arduino.

```
Wire.slaveWrite((uint8_t *)message, strlen(message));
```

I2C Slave APIs

Here are the I2C slave APIs. These function are intended to be used only for slave mode.

begin

In slave mode, the `begin` function must be used by passing the **slave address**. You can also define the **pins** and the **bus frequency**.

```
bool Wire.begin(uint8_t addr, int sdaPin, int sclPin, uint32_t frequency)
```

This function will return `true` if the peripheral was initialized correctly.

onReceive

The `onReceive` function is used to define the callback for the data received from the master.

```
void onReceive( void (*)(int) );
```

onRequest

The `onRequest` function is used to define the callback for the data to be send to the master.

```
void onRequest( void (*)(void) );
```

slaveWrite

The `slaveWrite` function writes on the slave response buffer before receiving the response message. This function is only used for adding the slave compatability for the ESP32.

> **Warning:** This function is only required for the ESP32. You **don't** need to use for ESP32-S2 and ESP32-C3.

```
size_t slaveWrite(const uint8_t *, size_t);
```

Example Application - WireSlave.ino

Here is an example of how to use the I2C in Slave Mode.

```
#include "Wire.h"

#define I2C_DEV_ADDR 0x55

uint32_t i = 0;

void onRequest(){
  Wire.print(i++);
  Wire.print(" Packets.");
  Serial.println("onRequest");
}

void onReceive(int len){
  Serial.printf("onReceive[%d]: ", len);
  while(Wire.available()){
    Serial.write(Wire.read());
  }
  Serial.println();
}

void setup() {
  Serial.begin(115200);
  Serial.setDebugOutput(true);
  Wire.onReceive(onReceive);
  Wire.onRequest(onRequest);
  Wire.begin((uint8_t)I2C_DEV_ADDR);

#if CONFIG_IDF_TARGET_ESP32
  char message[64];
  snprintf(message, 64, "%u Packets.", i++);
  Wire.slaveWrite((uint8_t *)message, strlen(message));
#endif
}

void loop() {

}
```

2.3.11 I2S

About

I2S - Inter-IC Sound, correctly written I^2S pronounced "eye-squared-ess", alternative notation is IIS. I^2S is an electrical serial bus interface standard used for connecting digital audio devices together.

It is used to communicate PCM (Pulse-Code Modulation) audio data between integrated circuits in an electronic device. The I^2S bus separates clock and serial data signals, resulting in simpler receivers than those required for asynchronous communications systems that need to recover the clock from the data stream.

Despite the similar name, I^2S is unrelated and incompatible with the bidirectional I^2C (IIC) bus.

The I^2S bus consists of at least three lines:

Note: All lines can be attached to almost any pin and this change can occur even during operation.

- **Bit clock line**
  - Officially "continuous serial clock (SCK)". Typically written "bit clock (BCLK)".
  - In this library function parameter `sckPin` or constant `PIN_I2S_SCK`.
- **Word clock line**
  - Officially "word select (WS)". Typically called "left-right clock (LRCLK)" or "frame sync (FS)".
  - 0 = Left channel, 1 = Right channel
  - In this library function parameter `fsPin` or constant `PIN_I2S_FS`.
- **Data line**
  - Officially "serial data (SD)", but can be called SDATA, SDIN, SDOUT, DACDAT, ADCDAT, etc.
  - Unlike Arduino I2S with single data pin switching between input and output, in ESP core driver use separate data line for input and output.
  - For backward compatibility, the shared data pin is `sdPin` or constant `PIN_I2S_SD` when using simplex mode.
  - When using in duplex mode, there are two data lines:
    - Output data line is called `outSdPin` for function parameter, or constant `PIN_I2S_SD_OUT`
    - Input data line is called `inSdPin` for function parameter, or constant `PIN_I2S_SD_IN`

I2S Modes

The I2S can be set up in three groups of modes:

- Master (default) or Slave.
- Simplex (default) or Duplex.
- Operation modes (Philips standard, ADC/DAC, PDM)
  - Most of them are dual-channel, some can be single channel

Note: Officially supported operation mode is only `I2S_PHILIPS_MODE`. Other modes are implemented, but we cannot guarantee flawless execution and behavior.

Master / Slave Mode

In **Master mode** (default) the device is generating clock signal `sckPin` and word select signal on `fsPin`.

In **Slave mode** the device listens on attached pins for the clock signal and word select - i.e. unless externally driven the pins will remain LOW.

How to enter either mode is described in the function section.

Operation Modes

Setting the operation mode is done with function `begin` (see API section)

- **`I2S_PHILIPS_MODE`**
  - Currently the only official* `PIN_I2S_SCK`
- `PIN_I2S_FS`
- `PIN_I2S_SD`
- `PIN_I2S_SD_OUT` only need to send one channel data but the data will be copied for another channel automatically, then both channels will transmit same data.
- **`ADC_DAC_MODE`** The output will be an analog signal on pins `25` (L or R?) and `26` (L or R?). Input will be received on pin `_inSdPin`. The data are sampled in 12 bits and stored in a 16 bits, with the 4 most significant bits set to zero.
- **`PDM_STEREO_MODE`** Pulse-density-modulation is similar to PWM, but instead, the pulses have constant width. The signal is modulated with the number of ones or zeroes in sequence.
- **`PDM_MONO_MODE`** Single-channel version of PDM mode described above.

Simplex / Duplex Mode

The **Simplex** mode is the default after driver initialization. Simplex mode uses the shared data pin `sdPin` or constant `PIN_I2S_SD` for both output and input, but can only read or write. This is the same behavior as in original Arduino library.

The **Duplex** mode uses two separate data pins:

- Output pin `outSdPin` for function parameter, or constant `PIN_I2S_SD_OUT`
- Input pin `inSdPin` for function parameter, or constant `PIN_I2S_SD_IN`

In this mode, the driver is able to read and write simultaneously on each line and is suitable for applications like walkie-talkie or phone.

Switching between these modes is performed simply by calling setDuplex() or setSimplex() (see APi section for details and more functions).

Arduino-ESP32 I2S API

The ESP32 I2S library is based on the Arduino I2S Library and implements a few more APIs, described in this documentation.

Initialization and deinitialization

Before initialization, choose which pins you want to use. In DAC mode you can use only pins *25* and *26* for the output.

begin (Master Mode)

Before usage choose which pins you want to use. In DAC mode you can use only pins 25 and 26 as output.

```
int begin(int mode, int sampleRate, int bitsPerSample)
```

Parameters:

- [in] `mode` one of above mentioned operation mode, for example `I2S_PHILIPS_MODE`.
- [in] `sampleRate` is the sampling rate in Hz. Currently officially supported value is only 16000 - other than this value will print warning, but continue to operate, however the resulting audio quality may suffer and the app may crash.
- [in] `bitsPerSample` is the number of bits in a channel sample.

Currently, the supported value is only 16 - other than this value will print a warning, but continues to operate, however, the resulting audio quality may suffer and the application may crash.

For `ADC_DAC_MODE` the only possible value will remain 16.

This function will return `true` on success or `fail` in case of failure.

When failed, an error message will be printed if subscribed.

begin (Slave Mode)

Performs initialization before use - creates buffers, task handling underlying driver messages, configuring and starting the driver operation.

This version initializes I2S in SLAVE mode (see previous entry for MASTER mode).

```
int begin(int mode, int bitsPerSample)
```

Parameters:

- [in] `mode` one of above mentioned modes for example `I2S_PHILIPS_MODE`.
- [in] `bitsPerSample` is the umber of bits in a channel sample. Currently, the only supported value is only 16 - other than this value will print warning, but continue to operate, however the resulting audio quality may suffer and the app may crash.

For `ADC_DAC_MODE` the only possible value will remain 16.

This function will return `true` on success or `fail` in case of failure.

When failed, an error message will be printed if subscribed.

end

Performs safe deinitialization - free buffers, destroy task, end driver operation, etc.

```
void end()
```

Pin setup

Pins can be changed in two ways- 1st constants, 2nd functions.

Note: Shared data pin can be equal to any other data pin, but must not be equal to clock pin nor frame sync pin! Input and Output pins must not be equal, but one of them can be equal to shared data pin!

```
sckPin != fsPin != outSdPin != inSdPin
```

```
sckPin != fsPin != sdPin
```

By default, the pin numbers are defined in constants in the header file. You can redefine any of those constants before including `I2S.h`. This way the driver will use these new default values and you will not need to specify pins in your code. The constants and their default values are:

- `PIN_I2S_SCK` 14
- `PIN_I2S_FS` 25
- `PIN_I2S_SD` 26
- `PIN_I2S_SD_OUT` 26
- `PIN_I2S_SD_IN` 35

The second option to change pins is using the following functions. These functions can be called on either on initialized or uninitialized object.

If called on the initialized object (after calling `begin`) the pins will change during operation. If called on the uninitialized object (before calling `begin`, or after calling `end`) the new pin setup will be used on next initialization.

setSckPin

Set and apply clock pin.

```
int setSckPin(int sckPin)
```

This function will return `true` on success or `fail` in case of failure.

setFsPin

Set and apply frame sync pin.

```
int setFsPin(int fsPin)
```

This function will return `true` on success or `fail` in case of failure.

setDataPin

Set and apply shared data pin used in simplex mode.

```
int setDataPin(int sdPin)
```

This function will return `true` on success or `fail` in case of failure.

setDataInPin

Set and apply data input pin.

```
int setDataInPin(int inSdPin)
```

This function will return `true` on success or `fail` in case of failure.

setDataOutPin

Set and apply data output pin.

```
int setDataOutPin(int outSdPin)
```

This function will return `true` on success or `fail` in case of failure.

setAllPins

Set all pins using given values in parameters. This is simply a wrapper of four functions mentioned above.

```
int setAllPins(int sckPin, int fsPin, int sdPin, int outSdPin, int inSdPin)
```

Set all pins to default i.e. take values from constants mentioned above. This simply calls the the function with the following constants.

- `PIN_I2S_SCK` 14
- `PIN_I2S_FS` 25
- `PIN_I2S_SD` 26
- `PIN_I2S_SD_OUT` 26
- `PIN_I2S_SD_IN` 35

```
int setAllPins()
```

getSckPin

Get the current value of the clock pin.

```
int getSckPin()
```

getFsPin

Get the current value of frame sync pin.

```
int getFsPin()
```

getDataPin

Get the current value of shared data pin.

```
int getDataPin()
```

getDataInPin

Get the current value of data input pin.

```
int getDataInPin()
```

getDataOutPin

Get the current value of data output pin.

```
int getDataOutPin()
```

onTransmit

Register the function to be called on each successful transmit event.

```
void onTransmit(void(*)(void))
```

onReceive

Register the function to be called on each successful receives event.

```
void onReceive(void(*)(void))
```

setBufferSize

Set the size of buffer.

```
int setBufferSize(int bufferSize)
```

This function can be called on both the initialized or uninitialized driver.

If called on initialized, it will change internal values for buffer size and re-initialize driver with new value. If called on uninitialized, it will only change the internal values which will be used for next initialization.

Parameter `bufferSize` must be in range from 8 to 1024 and the unit is sample words. The default value is 128.

Example: 16 bit sample, dual channel, buffer size for input:

```
128 = 2B sample * 2 channels * 128 buffer size * buffer count (default 2) =
1024B
```

And more `1024B for output buffer in total of `2kB` used.

This function always assumes dual-channel, keeping the same size even for MONO modes.

This function will return `true` on success or `fail` in case of failure.

When failed, an error message will be printed.

getBufferSize

Get current buffer sizes in sample words (see description for `setBufferSize`).

```
int getBufferSize()
```

Duplex vs Simplex

Original Arduino I2S library supports only *simplex* mode (only transmit or only receive at a time). For compatibility, we kept this behavior, but ESP natively supports *duplex* mode (receive and transmit simultaneously on separate pins). By default this library is initialized in simplex mode as it would in Arduino, switching input and output on `sdPin` (constant `PIN_I2S_SD` default pin 26).

setDuplex

Switch to duplex mode and use separate pins:

```
int setDuplex()
```

input: inSdPin (constant PIN_I2S_SD_IN, default 35) output: outSdPin (constant PIN_I2S_SD, default 26)

setSimplex

(Default mode)

Switch to simplex mode using shared data pin sdPin (constant PIN_I2S_SD, default 26).

```
int setSimplex()
```

isDuplex

Returns 1 if current mode is duplex, 0 if current mode is simplex (default).

```
int isDuplex()
```

Data stream

available

Returns number of **bytes** ready to read.

```
int available()
```

read

Read `size` bytes from internal buffer if possible.

```
int read(void* buffer, size_t size)
```

This function is non-blocking, i.e. if the requested number of bytes is not available, it will return as much as possible without waiting.

Hint: use `available()` before calling this function.

Parameters:

[out] `void* buffer` buffer into which will be copied data read from internal buffer. WARNING: this buffer must be allocated before use!

[in] `size_t size` number of bytes required to be read.

Returns number of successfully bytes read. Returns `false` ` in case of reading error.

Read one sample.

```
int read()
```

peek

Read one sample from the internal buffer and returns it.

```
int peek()
```

Repeated peeks will be returned in the same sample until `read` is called.

flush

Force write internal buffer to driver.

```
void flush()
```

write

Write a single byte.

```
size_t write(uint8_t)
```

Single-sample writes are blocking - waiting until there is free space in the internal buffer to be written into.

Returns number of successfully written bytes, in this case, 1. Returns 0 on error.

Write single sample.

```
size_t write(int32_t)
```

Single-sample writes are blocking - waiting until there is free space in the internal buffer to be written into.

Returns number of successfully written bytes. Returns 0 on error.

Expected return number is `bitsPerSample/8`.

Write buffer of supplied size;

```
size_t write(const void *buffer, size_t size)
```

Parameters:

[in] `const void *buffer` buffer to be written [in] `size_t size` size of buffer in bytes

Returns number of successfully written bytes. Returns 0 in case of error. The expected return number is equal to `size`.

write

This is a wrapper of the previous function performing typecast from *uint8_t*` to `void*`.

```
size_t write(const uint8_t *buffer, size_t size)
```

availableForWrite

Returns number of bytes available for write.

```
int availableForWrite()
```

write_blocking

Core function implementing blocking write, i.e. waits until all requested data are written.

```
size_t write_blocking(const void *buffer, size_t size)
```

WARNING: If too many bytes are requested, this can cause WatchDog Trigger Reset!

Returns number of successfully written bytes. Returns 0 on error.

write_nonblocking

Core function implementing non-blocking write, i.e. writes as much as possible and exits.

```
size_t write_nonblocking(const void *buffer, size_t size)
```

Returns number of successfully written bytes. Returns 0 on error.

Sample code

```
#include <I2S.h>
const int buff_size = 128;
int available, read;
uint8_t buffer[buff_size];

I2S.begin(I2S_PHILIPS_MODE, 16000, 16);
I2S.read(); // Switch the driver in simplex mode to receive
available = I2S.available();
if(available < buff_size){
  read = I2S.read(buffer, available);
}else{
  read = I2S.read(buffer, buff_size);
}
I2S.write(buffer, read);
I2S.end();
```

2.3.12 ESP Insights

About

ESP Insights is a remote diagnostics solution that allows users to remotely monitor the health of ESP devices in the field.

Developers normally prefer debugging issues by physically probing them using gdb or observing the logs. This surely helps debug issues, but there are often cases wherein issues are seen only in specific environments under specific conditions. Even things like casings and placement of the product can affect the behaviour. A few examples are

- Wi-Fi disconnections for a smart switch concealed in a wall.
- Smart speakers crashing during some specific usage pattern.
- Appliance frequently rebooting due to power supply issues.

Additional information about ESP Insights can be found here.

ESP Insights Agent API

Insights.begin

This initializes the ESP Insights agent.

```
bool begin(const char *auth_key, const char *node_id = NULL, uint32_t log_type =␣
↪0xFFFFFFFF, bool alloc_ext_ram = false);
```

- `auth_key` Auth key generated using Insights dashboard
- `log_type` Type of logs to be captured (value can be a mask of ESP_DIAG_LOG_TYPE_ERROR, ESP_DIAG_LOG_TYPE_WARNING and ESP_DIAG_LOG_TYPE_EVENT)

This function will return

1. true : On success
2. false in case of failure

Insights.send

Read insights data from buffers and send it to the cloud. Call to this function is asynchronous, it may take some time to send the data.

```
bool sendData()
```

This function will return

1. true : On success
2. false in case of failure

Insights.end

Deinitialize ESP Insights.

```
void end();
```

Insights.disable

Disable ESP Insights.

```
void disable();
```

ESP Insights Metrics API

metrics object of *Insights* class expose API's for using metrics.

Insights.metrics.addX

Register a metric of type X, where X is one of: Bool, Int, Uint, Float, String, IPv4 or MAC

```
bool addX(const char *tag, const char *key, const char *label, const char *path);
```

- `tag` : Tag of metrics
- `key` : Unique key for the metrics
- `label` : Label for the metrics
- `path` : Hierarchical path for key, must be separated by '.' for more than one level

This function will return

1. true : On success
2. false in case of failure

Insights.metrics.remove

Unregister a diagnostics metrics

```
bool remove(const char *key);
```

- `key` : Key for the metrics

This function will return

1. true : On success
2. false in case of failure

Insights.metrics.removeAll

Unregister all previously registered metrics

```
bool removeAll();
```

This function will return

1. true : On success
2. false in case of failure

Insights.metrics.setX

Add metrics of type X to storage, where X is one of: Bool, Int, Uint, Float, String, IPv4 or MAC

```
bool setX(const char *key, const void val);
```

- `key` : Key of metrics
- `val` : Value of metrics

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

Insights.metrics.dumpHeap

Dumps the heap metrics and prints them to the console. This API collects and reports metrics value at any give point in time.

```
bool dumpHeap();
```

This function will return

1. true : On success
2. false in case of failure

Insights.metrics.dumpWiFi

Dumps the wifi metrics and prints them to the console. This API can be used to collect wifi metrics at any given point in time.

```
bool dumpWiFi();
```

This function will return

1. true : On success
2. false in case of failure

Insights.metrics.setHeapPeriod

Reset the periodic interval By default, heap metrics are collected every 30 seconds, this function can be used to change the interval. If the interval is set to 0, heap metrics collection disabled.

```
void setHeapPeriod(uint32_t period);
```

- `period` : Period interval in seconds

Insights.metrics.setWiFiPeriod

Reset the periodic interval By default, wifi metrics are collected every 30 seconds, this function can be used to change the interval. If the interval is set to 0, wifi metrics collection disabled.

```
void setHeapPeriod(uint32_t period);
```

- `period` : Period interval in seconds

ESP Insights Variables API

variables object of *Insights* class expose API's for using variables.

Insights.variables.addX

Register a variable of type X, where X is one of: Bool, Int, Uint, Float, String, IPv4 or MAC

```
bool addX(const char *tag, const char *key, const char *label, const char *path);
```

- `tag` : Tag of variable
- `key` : Unique key for the variable
- `label` : Label for the variable
- `path` : Hierarchical path for key, must be separated by '.' for more than one level

This function will return

1. true : On success
2. false in case of failure

Insights.variables.remove

Unregister a diagnostics variable

```
bool remove(const char *key);
```

- `key` : Key for the variable

This function will return

1. true : On success
2. false in case of failure

Insights.variables.removeAll

Unregister all previously registered variables

```
bool unregisterAll();
```

This function will return

1. true : On success
2. false in case of failure

Insights.variables.setX

Add variable of type X to storage, where X is one of: Bool, Int, Uint, Float, String, IPv4 or MAC

```
bool setX(const char *key, const void val);
```

- `key` : Key of metrics
- `val` : Value of metrics

This function will return

1. true : On success
2. false in case of failure

Example

To get started with Insights, you can try:

```
#include "Insights.h"
#include "WiFi.h"

const char insights_auth_key[] = "<ENTER YOUR AUTH KEY>";

#define WIFI_SSID       "<ENTER YOUR SSID>"
#define WIFI_PASSPHRASE "<ENTER YOUR PASSWORD>"

void setup()
{
    Serial.begin(115200);
    WiFi.mode(WIFI_STA);
    WiFi.begin(WIFI_SSID, WIFI_PASSPHRASE);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("WiFi connected");
    if(!Insights.begin(insights_auth_key)){
        return;
    }
    Serial.println("=======================================");
```

(continues on next page)

(continued from previous page)

```
    Serial.printf("ESP Insights enabled Node ID %s\n", Insights.nodeID());
    Serial.println("=========================================");
}

void loop()
{
    delay(1000);
}
```

2.3.13 LED Control (LEDC)

About

The LED control (LEDC) peripheral is primarly designed to control the intensity of LEDs, although it can also be used to generate PWM signals for other purposes.

ESP32 SoCs has from 6 to 16 channels (variates on socs, see table below) which can generate independent waveforms, that can be used for example to drive RGB LED devices.

ESP32 SoC	Number of LEDC channels
ESP32	16
ESP32-S2	8
ESP32-C3	6
ESP32-S3	8

Arduino-ESP32 LEDC API

ledcSetup

This function is used to setup the LEDC channel frequency and resolution.

```
uint32_t ledcSetup(uint8_t channel, uint32_t freq, uint8_t resolution_bits);
```

- `channel` select LEDC channel to config.
- `freq` select frequency of pwm.
- `resolution_bits` select resolution for ledc channel.
  - range is 1-14 bits (1-20 bits for ESP32).

This function will return `frequency` configured for LEDC channel. If `0` is returned, error occurs and ledc channel was not configured.

ledcWrite

This function is used to set duty for the LEDC channel.

```
void ledcWrite(uint8_t chan, uint32_t duty);
```

- `chan` select the LEDC channel for writing duty.
- `duty` select duty to be set for selected channel.

ledcRead

This function is used to get configured duty for the LEDC channel.

```
uint32_t ledcRead(uint8_t chan);
```

- `chan` select LEDC channel to read the configured duty.

This function will return `duty` set for selected LEDC channel.

ledcReadFreq

This function is used to get configured frequency for the LEDC channel.

```
uint32_t ledcReadFreq(uint8_t chan);
```

- `chan` select the LEDC channel to read the configured frequency.

This function will return `frequency` configured for selected LEDC channel.

ledcWriteTone

This function is used to setup the LEDC channel to 50 % PWM tone on selected frequency.

```
uint32_t ledcWriteTone(uint8_t chan, uint32_t freq);
```

- `chan` select LEDC channel.
- `freq` select frequency of pwm signal.

This function will return `frequency` set for channel. If `0` is returned, error occurs and ledc cahnnel was not configured.

ledcWriteNote

This function is used to setup the LEDC channel to specific note.

```
uint32_t ledcWriteNote(uint8_t chan, note_t note, uint8_t octave);
```

- `chan` select LEDC channel.
- `note` select note to be set.

NOTE_C	NOTE_Cs	NOTE_D	NOTE_Eb	NOTE_E	NOTE_F
NOTE_Fs	NOTE_G	NOTE_Gs	NOTE_A	NOTE_Bb	NOTE_B

- `octave` select octave for note.

This function will return `frequency` configured for the LEDC channel according to note and octave inputs. If `0` is returned, error occurs and the LEDC channel was not configured.

ledcAttachPin

This function is used to attach the pin to the LEDC channel.

```
void ledcAttachPin(uint8_t pin, uint8_t chan);
```

- `pin` select GPIO pin.
- `chan` select LEDC channel.

ledcDetachPin

This function is used to detach the pin from LEDC.

```
void ledcDetachPin(uint8_t pin);
```

- `pin` select GPIO pin.

ledcChangeFrequency

This function is used to set frequency for the LEDC channel.

```
uint32_t ledcChangeFrequency(uint8_t chan, uint32_t freq, uint8_t bit_num);
```

- `channel` select LEDC channel.
- `freq` select frequency of pwm.
- `bit_num` select resolution for LEDC channel.
  - range is 1-14 bits (1-20 bits for ESP32).

This function will return `frequency` configured for the LEDC channel. If `0` is returned, error occurs and the LEDC channel frequency was not set.

analogWrite

This function is used to write an analog value (PWM wave) on the pin. It is compatible with Arduinos analogWrite function.

```
void analogWrite(uint8_t pin, int value);
```

- `pin` select the GPIO pin.
- `value` select the duty cycle of pwm. * range is from 0 (always off) to 255 (always on).

analogWriteResolution

This function is used to set resolution for all analogWrite channels.

```
void analogWriteResolution(uint8_t bits);
```

- `bits` select resolution for analog channels.

analogWriteFrequency

This function is used to set frequency for all analogWrite channels.

```
void analogWriteFrequency(uint32_t freq);
```

- `freq` select frequency of pwm.

Example Applications

LEDC software fade example:

```
/*
 LEDC Software Fade

 This example shows how to software fade LED
 using the ledcWrite function.

 Code adapted from original Arduino Fade example:
 https://www.arduino.cc/en/Tutorial/Fade

 This example code is in the public domain.
 */

// use first channel of 16 channels (started from zero)
#define LEDC_CHANNEL_0     0

// use 12 bit precission for LEDC timer
#define LEDC_TIMER_12_BIT  12

// use 5000 Hz as a LEDC base frequency
#define LEDC_BASE_FREQ     5000

// fade LED PIN (replace with LED_BUILTIN constant for built-in LED)
#define LED_PIN            5

int brightness = 0;    // how bright the LED is
int fadeAmount = 5;    // how many points to fade the LED by

// Arduino like analogWrite
// value has to be between 0 and valueMax
void ledcAnalogWrite(uint8_t channel, uint32_t value, uint32_t valueMax = 255) {
  // calculate duty, 4095 from 2 ^ 12 - 1
```

(continues on next page)

(continued from previous page)

```
  uint32_t duty = (4095 / valueMax) * min(value, valueMax);

  // write duty to LEDC
  ledcWrite(channel, duty);
}

void setup() {
  // Setup timer and attach timer to a led pin
  ledcSetup(LEDC_CHANNEL_0, LEDC_BASE_FREQ, LEDC_TIMER_12_BIT);
  ledcAttachPin(LED_PIN, LEDC_CHANNEL_0);
}

void loop() {
  // set the brightness on LEDC channel 0
  ledcAnalogWrite(LEDC_CHANNEL_0, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness <= 0 || brightness >= 255) {
    fadeAmount = -fadeAmount;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

LEDC Write RGB example:

```
/*
  ledcWrite_RGB.ino
  Runs through the full 255 color spectrum for an rgb led
  Demonstrate ledcWrite functionality for driving leds with PWM on ESP32

  This example code is in the public domain.

  Some basic modifications were made by vseven, mostly commenting.
 */

// Set up the rgb led names
uint8_t ledR = 2;
uint8_t ledG = 4;
uint8_t ledB = 5;

uint8_t ledArray[3] = {1, 2, 3}; // three led channels

const boolean invert = true; // set true if common anode, false if common cathode

uint8_t color = 0;          // a value from 0 to 255 representing the hue
uint32_t R, G, B;           // the Red Green and Blue color components
uint8_t brightness = 255;  // 255 is maximum brightness, but can be changed.  Might need␣
↪256 for common anode to fully turn off.
```

(continues on next page)

(continued from previous page)

```
// the setup routine runs once when you press reset:
void setup()
{
  Serial.begin(115200);
  delay(10);

  ledcAttachPin(ledR, 1); // assign RGB led pins to channels
  ledcAttachPin(ledG, 2);
  ledcAttachPin(ledB, 3);

  // Initialize channels
  // channels 0-15, resolution 1-16 bits, freq limits depend on resolution
  // ledcSetup(uint8_t channel, uint32_t freq, uint8_t resolution_bits);
  ledcSetup(1, 12000, 8); // 12 kHz PWM, 8-bit resolution
  ledcSetup(2, 12000, 8);
  ledcSetup(3, 12000, 8);
}

// void loop runs over and over again
void loop()
{
  Serial.println("Send all LEDs a 255 and wait 2 seconds.");
  // If your RGB LED turns off instead of on here you should check if the LED is common␣
↪anode or cathode.
  // If it doesn't fully turn off and is common anode try using 256.
  ledcWrite(1, 255);
  ledcWrite(2, 255);
  ledcWrite(3, 255);
  delay(2000);
  Serial.println("Send all LEDs a 0 and wait 2 seconds.");
  ledcWrite(1, 0);
  ledcWrite(2, 0);
  ledcWrite(3, 0);
  delay(2000);

  Serial.println("Starting color fade loop.");

 for (color = 0; color < 255; color++) { // Slew through the color spectrum

  hueToRGB(color, brightness);  // call function to convert hue to RGB

  // write the RGB values to the pins
  ledcWrite(1, R); // write red component to channel 1, etc.
  ledcWrite(2, G);
  ledcWrite(3, B);

  delay(100); // full cycle of rgb over 256 colors takes 26 seconds
 }

}
```

(continues on next page)

(continued from previous page)

```
// Courtesy http://www.instructables.com/id/How-to-Use-an-RGB-LED/?ALLSTEPS
// function to convert a color to its Red, Green, and Blue components.

void hueToRGB(uint8_t hue, uint8_t brightness)
{
    uint16_t scaledHue = (hue * 6);
    uint8_t segment = scaledHue / 256; // segment 0 to 5 around the
                                            // color wheel
    uint16_t segmentOffset =
      scaledHue - (segment * 256); // position within the segment

    uint8_t complement = 0;
    uint16_t prev = (brightness * ( 255 -  segmentOffset)) / 256;
    uint16_t next = (brightness *  segmentOffset) / 256;

    if(invert)
    {
      brightness = 255 - brightness;
      complement = 255;
      prev = 255 - prev;
      next = 255 - next;
    }

    switch(segment ) {
    case 0:      // red
        R = brightness;
        G = next;
        B = complement;
    break;
    case 1:     // yellow
        R = prev;
        G = brightness;
        B = complement;
    break;
    case 2:     // green
        R = complement;
        G = brightness;
        B = next;
    break;
    case 3:    // cyan
        R = complement;
        G = prev;
        B = brightness;
    break;
    case 4:    // blue
        R = next;
        G = complement;
        B = brightness;
    break;
   case 5:      // magenta
    default:
        R = brightness;
```

(continues on next page)

(continued from previous page)

```
        G = complement;
        B = prev;
    break;
    }
}
```

2.3.14 Preferences

About

The Preferences library is unique to arduino-esp32. It should be considered as the replacement for the Arduino EEPROM library.

It uses a portion of the on-board non-volatile memory (NVS) of the ESP32 to store data. This data is retained across restarts and loss of power events to the system.

Preferences works best for storing many small values, rather than a few large values. If large amounts of data are to be stored, consider using a file system library such as LitteFS.

The Preferences library is usable by all ESP32 variants.

Header File

```
#include <Preferences.h>
```

Overview

Library methods are provided to:

- create a namespace;
- open and close a namespace;
- store and retrieve data within a namespace for supported data types;
- determine if a key value has been initialized;
- delete a `key-value` pair;
- delete all `key-value` pairs in a namespace;
- determine data types stored against a key;
- determine the number of key entries in the namespace.

Preferences directly supports the following data types:

Table 1: **Table 1 — Preferences Data Types**

Preferences Type	Data Type	Size (bytes)
Bool	bool	1
Char	int8_t	1
UChar	uint8_t	1
Short	int16_t	2
UShort	uint16_t	2
Int	int32_t	4
UInt	uint32_t	4
Long	int32_t	4
ULong	uint32_t	4
Long64	int64_t	8
ULong64	uint64_t	8
Float	float_t	8
Double	double_t	8
String	const char*	variable
	String	
Bytes	uint8_t	variable

String values can be stored and retrieved either as an Arduino String or as a null terminated `char` array (c-string).

Bytes type is used for storing and retrieving an arbitrary number of bytes in a namespace.

Arduino-esp32 Preferences API

`begin`

Open non-volatile storage with a given namespace name from an NVS partition.

```
bool begin(const char * name, bool readOnly=false, const char* partition_
↪label=NULL)
```

Parameters

- **`name` (Required)**
  - Namespace name. Maximum length is 15 characters.
- **`readOnly` (Optional)**
  - `false` will open the namespace in read-write mode.
  - `true` will open the namespace in read-only mode.
  - if omitted, the namespace is opened in read-write mode.
- **`partition_label` (Optional)**
  - name of the NVS partition in which to open the namespace.
  - if omitted, the namespace is opened in the "`nvs`" partition.

Returns

- `true` if the namespace was opened successfully; `false` otherwise.

Notes

- If the namespace does not exist within the partition, it is first created.
- Attempting to write a key value to a namespace open in read-only mode will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

end

Close the currently opened namespace.

```
void end()
```

Parameters

- None

Returns

- Nothing

Note

- After closing a namespace, methods used to access it will fail.

clear

Delete all keys and values from the currently opened namespace.

```
bool clear()
```

Parameters

- None

Returns

- `true` if all keys and values were deleted; `false` otherwise.

Note

- the namespace name still exists afterward.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

remove

Delete a key-value pair from the currently open namespace.

```
bool remove(const char * key)
```

Parameters

- **key (Required)**
  - the name of the key to be deleted.

Returns

- `true` if key-value pair was deleted; `false` otherwise.

Note

- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

putChar, putUChar

Store a value against a given key in the currently open namespace.

```
size_t putChar(const char* key, int8_t value)
size_t putUChar(const char* key, uint8_t value)
```

Parameters

- **key (Required)**
  - if the key does not exist in the currently opened namespace it is first created.
- **value (Required)**
  - must match the data type of the method.

Returns

- 1 (the number of bytes stored for these data types) if the call is successful; 0 otherwise.

Notes

- Attempting to store a value without a namespace being open in read-write mode will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

putShort, putUShort

Store a value against a given key in the currently open namespace.

```
size_t putShort(const char* key, int16_t value)
size_t putUShort(const char* key, uint16_t value)
```

Parameters

- **key (Required)**
  - if the key does not exist in the currently opened namespace it is first created.
- **value (Required)**
  - must match the data type of the method.

Returns

- 2 (the number of bytes stored for these data types) if the call is successful; 0 otherwise.

Notes

- Attempting to store a value without a namespace being open in read-write mode will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

putInt, putUInt

putLong, putULong

Store a value against a given key in the currently open namespace.

```
size_t putInt(const char* key, int32_t value)
size_t putUInt(const char* key, uint32_t value)
size_t putLong(const char* key, int32_t value)
size_t putULong(const char* key, uint32_t value)
```

Parameters

- **key (Required)**
  - if the key does not exist in the currently opened namespace it is first created.
- **value (Required)**
  - must match the data type of the method.

Returns

- `4` (the number of bytes stored for these data types) if the call is successful; `0` otherwise.

Notes

- Attempting to store a value without a namespace being open in read-write mode will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

putLong64, putULong64

putFloat, putDouble

Store a value against a given key in the currently open namespace.

```
size_t putLong64(const char* key, int64_t value)
size_t putULong64(const char* key, uint64_t value)
size_t putFloat(const char* key, float_t value)
size_t putDouble(const char* key, double_t value)
```

Parameters

- **key (Required)**
  - if the key does not exist in the currently opened namespace it is first created.
- **value (Required)**
  - must match the data type of the method.

Returns

- `8` (the number of bytes stored for these data types) if the call is successful; `0` otherwise.

Notes

- Attempting to store a value without a namespace being open in read-write mode will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

putBool

Store a value against a given key in the currently open namespace.

```
size_t putBool(const char* key, bool value)
```

Parameters

- **key (Required)**
  - if the key does not exist in the currently opened namespace it is first created.
- **value (Required)**
  - must match the data type of the method.

Returns

- `true` if successful; `false` otherwise.

Notes

- Attempting to store a value without a namespace being open in read-write mode will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

putString

Store a variable length value against a given key in the currently open namespace.

```
size_t putString(const char* key, const char* value);
size_t putString(const char* key, String value);
```

Parameters

- **key (Required)**
  - if the key does not exist in the currently opened namespace it is first created.
- **value (Required)**
  - if `const char*`, a null-terminated (c-string) character array.
  - if `String`, a valid Arduino String type.

Returns

- if successful: the number of bytes stored; `0` otherwise.

Notes

- Attempting to store a value without a namespace being open in read-write mode will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

putBytes

Store a variable number of bytes against a given key in the currently open namespace.

```
size_t putBytes(const char* key, const void* value, size_t len);
```

Parameters

- **key (Required)**
  - if the key does not exist in the currently opened namespace it is first created.
- **value (Required)**
  - pointer to an array or buffer containing the bytes to be stored.
- **len (Required)**
  - the number of bytes from `value` to be stored.

Returns

- if successful: the number of bytes stored; `0` otherwise.

Notes

- Attempting to store a value without a namespace being open in read-write mode will fail.
- This method operates on the bytes used by the underlying data type, not the number of elements of a given data type. The data type of `value` is not retained by the Preferences library afterward.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

getChar, getUChar

Retrieve a value stored against a given key in the currently open namespace.

```
int8_t getChar(const char* key, int8_t defaultValue = 0)
uint8_t getUChar(const char* key, uint8_t defaultValue = 0)
```

Parameters

- `key` (Required)
- **`defaultValue` (Optional)**
  - must match the data type of the method if provided.

Returns

- the value stored against `key` if the call is successful.
- `defaultValue`, if it is provided; `0` otherwise.

Notes

- Attempting to retrieve a key without a namespace being available will fail.
- Attempting to retrieve value from a non existant key will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

getShort, getUShort

Retrieve a value stored against a given key in the currently open namespace.

```
int16_t getShort(const char* key, int16_t defaultValue = 0)
uint16_t getUShort(const char* key, uint16_t defaultValue = 0)
```

Except for the data type returned, behaves exactly like `getChar`.

getInt, getUInt

Retrieve a value stored against a given key in the currently open namespace.

```
int32_t getInt(const char* key, int32_t defaultValue = 0)
uint32_t getUInt(const char* key, uint32_t defaultValue = 0)
```

Except for the data type returned, behaves exactly like `getChar`.

getLong, getULong

Retrieve a value stored against a given key in the currently open namespace.

```
int32_t getLong(const char* key, int32_t defaultValue = 0)
uint32_t getULong(const char* key, uint32_t defaultValue = 0)
```

Except for the data type returned, behaves exactly like `getChar`.

getLong64, getULong64

Retrieve a value stored against a given key in the currently open namespace.

```
int64_t getLong64(const char* key, int64_t defaultValue = 0)
uint64_t getULong64(const char* key, uint64_t defaultValue = 0)
```

Except for the data type returned, behaves exactly like `getChar`.

getFloat

Retrieve a value stored against a given key in the currently open namespace.

```
float_t getFloat(const char* key, float_t defaultValue = NAN)
```

Except for the data type returned and the value of `defaultValue`, behaves exactly like `getChar`.

getDouble

Retrieve a value stored against a given key in the currently open namespace.

```
double_t getDouble(const char* key, double_t defaultValue = NAN)
```

Except for the data type returned and the value of `defaultValue`, behaves exactly like `getChar`.

getBool

Retrieve a value stored against a given key in the currently open namespace.

```
uint8_t getUChar(const char* key, uint8_t defaultValue = 0);
```

Except for the data type returned, behaves exactly like `getChar`.

getString

Copy a string of `char` stored against a given key in the currently open namespace to a buffer.

```
size_t getString(const char* key, char* value, size_t len);
```

Parameters

- `key` (Required)
- **`value` (Required)**
  - a buffer of a size large enough to hold `len` bytes
- **`len` (Required)**
  - the number of type `char`` to be written to the buffer pointed to by `value`

Returns

- if successful; the number of bytes equal to `len` is written to the buffer pointed to by `value`, and the method returns `1`.
- if the method fails, nothing is written to the buffer pointed to by `value` and the method returns `0`.

Notes

- `len` must equal the number of bytes stored against the key or the call will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

getString

Retrieve an Arduino String value stored against a given key in the currently open namespace.

```
String getString(const char* key, String defaultValue = String());
```

Parameters

- `key` (Required)
- `defaultValue` (Optional)

Returns

- the value stored against `key` if the call if successful
- if the method fails: it returns `defaultValue`, if provided; "" (an empty String) otherwise.

Notes

- `defaultValue` must be of type `String`.

getBytes

Copy a series of bytes stored against a given key in the currently open namespace to a buffer.

```
size_t getBytes(const char* key, void * buf, size_t len);
```

Parameters

- `key` (Required)
- **`buf` (Required)**
  - a buffer of a size large enough to hold `len` bytes.
- **`len` (Required)**
  - the number of bytes to be written to the buffer pointed to by `buf`

Returns

- if successful, the number of bytes equal to `len` is written to buffer `buf`, and the method returns 1.
- if the method fails, nothing is written to the buffer and the method returns `0`.

Notes

- `len` must equal the number of bytes stored against the key or the call will fail.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

getBytesLength

Get the number of bytes stored in the value against a key of type `Bytes` in the currently open namespace.

```
size_t getBytesLength(const char* key)
```

Parameters

- `key` (Required)

Returns

- if successful: the number of bytes in the value stored against `key`; `0` otherwise.

Notes

- This method will fail if `key` is not of type `Bytes`.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

getType

Get the Preferences data type of a given key within the currently open namespace.

```
PreferenceType getType(const char* key)
```

Parameters

- `key` (Required)

Returns

- an `int` value as per Table 2 below.
- a value of `10` (PT_INVALID) if the call fails.

Notes

- The return values are enumerated in `Preferences.h`. Table 2 includes the enumerated values for information.
- A return value can map to more than one Prefs Type.
- The method will fail if: the namespace is not open; the key does not exist; the provided key exceeds 15 characters.

Table 2: **Table 2 — getType Return Values**

Return value	Prefs Type	Data Type	Enumerated Value
0	Char	int8_t	PT_I8
1	UChar	uint8_t	PT_U8
	Bool	bool	
2	Short	int16_t	PT_I16
3	UShort	uint16_t	PT_U16
4	Int	int32_t	PT_I32
	Long		
5	UInt	uint32_t	PT_U32
	ULong		
6	Long64	int64_t	PT_I64
7	ULong64	uint64_t	PT_U64
8	String	String	PT_STR
		*char	
9	Double	double_t	PT_BLOB
	Float	float_t	
	Bytes	uint8_t	
10	-	-	PT_INVALID

freeEntries

Get the number of free entries available in the key table of the currently open namespace.

```
size_t freeEntries()
```

Parameters

- none

Returns

- if successful: the number of free entries available in the key table of the currently open namespace; `0` otherwise.

Notes

- keys storing values of type `Bool`, `Char`, `UChar`, `Short`, `UShort`, `Int`, `UInt`, `Long`, `ULong`, `Long64`, `ULong64` use one entry in the key table.
- keys storing values of type `Float` and `Double` use three entries in the key table.
- Arduino or c-string `String` types use a minimum of two key table entries with the number of entries increasing with the length of the string.
- keys storing values of type `Bytes` use a minimum of three key table entries with the number of entries increasing with the number of bytes stored.
- A message providing the reason for a failed call is sent to the arduino-esp32 `log_e` facility.

2.3.15 Pulse Counter

About

Note: This peripheral is not supported yet by the Arduino API's.

2.3.16 ESP Rainmaker

About

This library allows to work with ESP RainMaker.

ESP RainMaker is an end-to-end solution offered by Espressif to enable remote control and monitoring for ESP32-S2 and ESP32 based products without any configuration required in the Cloud. The primary components of this solution are:

- Claiming Service (to get the Cloud connectivity credentials)
- RainMaker library (i.e. this library, to develop the firmware)
- RainMaker Cloud (backend, offering remote connectivity)
- RainMaker Phone App/CLI (Client utilities for remote access)

The key features of ESP RainMaker are:

1. Ability to define own devices and parameters, of any type, in the firmware.
2. Zero configuration required on the Cloud.
3. Phone apps that dynamically render the UI as per the device information.

Additional information about ESP RainMaker can be found here.

ESP RainMaker Agent API

RMaker.initNode

This initializes the ESP RainMaker agent, wifi and creates the node.

You can also set the configuration of the node using the following API

`RMaker.setTimeSync(bool val)`

NOTE: If you want to set the configuration for the node then these configuration API must be called before *RMaker.initNode()*.

```
Node initNode(const char *name, const char *type);
```

- `name` Name of the node
- `type` Type of the node

This function will return object of Node.

RMaker.start

It starts the ESP RainMaker agent.

NOTE:

1. ESP RainMaker agent should be initialized before this call.
2. Once ESP RainMaker agent starts, compulsorily call WiFi.beginProvision() API.

```
esp_err_t start();
```

This function will return *ESP_OK* on success or *Error* in case of failure.

RMaker.stop

It stops the ESP RainMaker agent which was started using *RMaker.start()*.

```
esp_err_t stop()
```

This function will return

1. *ESP_OK* : On success
2. Error in case of failure.

RMaker.deinitNode

It deinitializes the ESP RainMaker agent and the node created using *RMaker.initNode()*.

```
esp_err_t deinitNode(Node node)
```

- `node` : Node object created using *RMaker.initNode()*

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

RMaker.enableOTA

It enables OTA as per the ESP RainMaker Specification. For more details refer ESP RainMaker documentation. check here.

```
esp_err_t enableOTA(ota_type_t type);
```

- **`type`** [The OTA workflow type.]
  - OTA_USING_PARAMS
  - OTA_USING_TOPICS

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

RMaker.enableSchedule

This API enables the scheduling service for the node. For more information, check here.

```
esp_err_t enableSchedule();
```

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

RMaker.enableScenes

This API enables the Scenes service for the node. It should be called after *RMaker.initNode()* and before *RMaker.start()*. For more information, check here.

```
esp_err_t enableScenes()
```

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

RMaker.setTimeZone

This API set's the timezone as a user friendly location string. Check here for a list of valid values.

NOTE : default value is "Asia/Shanghai".

This API comes into picture only when working with scheduling.

```
esp_err_t setTimeZone(const char *tz);
```

- `tz` : Valid values as specified in documentation.

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

ESP RainMaker Node API

Node class expose API's for node.

NOTE : my_node is the object of Node class.

my_node.getNodeID

It returns the unique node_id assigned to the node. This node_id is usually the MAC address of the board.

```
char * getNodeID()
```

- `tz` : Valid values as specified in documentation.

This function will return

1. *char* * : Pointer to a NULL terminated node_id string.

my_node.getNodeInfo

It returns pointer to the node_info_t as configured during node initialisation.

```
node_info_t * getNodeInfo();
```

This function will return

1. *node_info_t* : Pointer to the structure node_info_t on success.
2. *NULL* : On failure.

ESP RainMaker node info

It has following data member

1. char * name
2. char * type
3. char * fw_version
4. char * model

my_node.addNodeAttr

It adds a new attribute as the metadata to the node.

NOTE : Only string values are allowed.

```
esp_err_t addNodeAttr(const char *attr_name, const char *val);
```

- `attr_name` : Name of the attribute
- `val` : Value of the attribute

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

my_node.addDevice

It adds a device to the node.

NOTE :

- This is the mandatory API to register device to node.
- Single Node can have multiple devices.
- Device name should be unique for each device.

```
esp_err_t addDevice(Device device);
```

- `device` : Device object

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

my_node.removeDevice

It removes a device from the node.

```
esp_err_t removeDevice(Device device);
```

- `device` : Device object

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

ESP RainMaker Device API

Device class expose API's for virtual devices on the node. Parameterized constructor is defined which creates the virtual device on the node. Using Device class object you can create your own device.

NOTE : my_device is the object of Device class

```
Device my_device(const char *dev_name, const char *dev_type, void *priv_data);
```

- `dev_name` : Unique device name
- **`dev_type`** [Optional device type. It can be kept NULL.]
  - **Standard Device Types**
    - ESP_RMAKER_DEVICE_SWITCH
    - ESP_RMAKER_DEVICE_LIGHTBULB
    - ESP_RMAKER_DEVICE_FAN
    - ESP_RMAKER_DEVICE_TEMP_SENSOR
- `priv_data` : Private data associated with the device. This will be passed to the callbacks.

NOTE : This created device should be added to the node using `my_node.addDevice(my_device);`

- Sample example

```
Device my_device("Switch");
Device my_device("Switch1", NULL, NULL);
```

- Here, dev_name is compulsory, rest are optional.
- Node can have multiple device, each device should have unique device name.

Standard Devices

- Classes are defined for the standard devices.
- Creating object of these class creates the standard device with default parameters to it.
- **Class for standard devices**
  - Switch
  - LightBulb
  - TemperatureSensor
  - Fan

```
Switch my_switch(const char *dev_name, void *priv_data, bool power);
```

- `dev_name` : Unique device name by default it is "switch" for switch device.
- `priv_data` : Private data associated with the device. This will be passed to the callbacks.
- `power` : It is the value that can be set for primary parameter.

Sample example for standard device.

```
Switch switch1;
Switch switch2("switch2", NULL, true);
```

- *"switch2"* : Name for standard device.
- *NULL* : Private data for the device, which will be used in callback.
- *true* : Default value for the primary param, in case of switch it is power.

NOTE: No parameter are compulsory for standard devices. However if you are creating two objects of same standard class then in that case you will have to set the device name, if not then both device will have same name which is set by default, hence device will not get create. *Device name should be unique for each device.*

my_device.getDeviceName

It returns the name of the Device.

```
const char * getDeviceName();
```

- `device` : Device object

This function will return

- *char* *: Returns Device name.

NOTE: Each device on the node should have unique device name.

my_device.addDeviceAttr

It adds attribute to the device. Device attributes are reported only once after a boot-up as part of the node configuration. Eg. Serial Number

```
esp_err_t addDeviceAttr(const char *attr_name, const char *val);
```

- `attr_name` : Name of the attribute
- `val` : Value of the attribute

This funtion will return

1. *ESP_OK* : On success
2. Error in case of failure

my_device.deleteDevice

It deletes the device created using parameterized constructor.

This device should be first removed from the node using *my_node.removeDevice(my_device)*.

```
esp_err_t deleteDevice();
```

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

my_device.addXParam

It adds standard parameter to the device.

NOTE: X is the default name by which parameter is referred, you can specify your own name to each parameter.

- Eg. *my_device.addPowerParam(true)* here power parameter is referred with name Power.
- Eg. *my_device.addHueParam(12)* here hue parameter is referred with name Hue.

You can specify your own name to each parameter

- Eg. *my_device.addNameParam("NickName")* here name parameter is referred with name NickName.
- Eg. *my_device.addPowerParam(true, "FanPower")* here power parameter is referred with name FanPower.

Standard Parameters

- **These are the standard parameters.**
  - Name : ESP_RMAKER_DEF_NAME_PARAM
  - Power : ESP_RMAKER_DEF_POWER_NAME
  - Brightness : ESP_RMAKER_DEF_BRIGHTNESS_NAME
  - Hue : ESP_RMAKER_DEF_HUE_NAME
  - Saturation : ESP_RMAKER_DEF_SATURATION_NAME
  - Intensity : ESP_RMAKER_DEF_INTENSITY_NAME

  - CCT : ESP_RMAKER_DEF_CCT_NAME
  - Direction : ESP_RMAKER_DEF_DIRECTION_NAME
  - Speed : ESP_RMAKER_DEF_SPEED_NAME
  - Temperature : ESP_RMAKER_DEF_TEMPERATURE_NAME

```
esp_err_t addNameParam(const char *param_name = ESP_RMAKER_DEF_NAME_PARAM);
esp_err_t addPowerParam(bool val, const char *param_name = ESP_RMAKER_DEF_POWER_NAME);
esp_err_t addBrightnessParam(int val, const char *param_name = ESP_RMAKER_DEF_BRIGHTNESS_
↪NAME);
esp_err_t addHueParam(int val, const char *param_name = ESP_RMAKER_DEF_HUE_NAME);
esp_err_t addSaturationParam(int val, const char *param_name = ESP_RMAKER_DEF_SATURATION_
↪NAME);
esp_err_t addIntensityParam(int val, const char *param_name = ESP_RMAKER_DEF_INTENSITY_
↪NAME);
esp_err_t addCCTParam(int val, const char *param_name = ESP_RMAKER_DEF_CCT_NAME);
esp_err_t addDirectionParam(int val, const char *param_name = ESP_RMAKER_DEF_DIRECTION_
↪NAME);
esp_err_t addSpeedParam(int val, const char *param_name = ESP_RMAKER_DEF_SPEED_NAME);
esp_err_t addTempratureParam(float val, const char *param_name = ESP_RMAKER_DEF_
↪TEMPERATURE_NAME);
```

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

NOTE : Care should be taken while accessing name of parameter. Above mentioned are the two ways using which default name of parameters can be accessed. Either LHS or RHS.

my_device.assignPrimaryParam

It assigns a parameter (already added using addXParam() or addParam()) as a primary parameter, which can be used by clients (phone apps specifically) to give prominence to it.

```
esp_err_t assignPrimaryParam(param_handle_t *param);
```

- `param` : Handle of the parameter. It is obtained using *my_device.getParamByName()*.

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

my_device.getParamByName

```
param_handle_t * getParamByName(const char *param_name);
```

- `param_name` : It is the name of the parameter which was added using addXparam() or addParam().

This function will return object of the parameter.

my_device.addParam

It allows user to add custom parameter to the device created using *Param* class.

```
esp_err_t addParam(Param parameter);
```

- `parameter` : Object of Param

This function will return

1.`ESP_OK` : On success 2. Error in case of failure

NOTE: Param class exposes API's to create the custom parameter.

my_device.updateAndReportParam

It updates the parameter assosicated with particular device on ESP RainMaker cloud.

```
esp_err_t updateAndReportParam(const char *param_name, value);
```

- `param_name` : Name of the parameter
- `value` : Value to be updated. It can be int, bool, char * , float.

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

my_device.addCb

It registers read and write callback for the device which will be invoked as per requests received from the cloud (or other paths as may be added in future).

```
void addCb(deviceWriteCb write_cb, deviceReadCb read_cb);
```

- **`write_cb`** [Function with signature] func_name(Device *device, Param *param, const param_val_t val, void *priv_data, write_ctx_t *ctx);
- **`read_cb`** [Function with signature] func_name(Device *device, Param *param, void *priv_data, read_ctx_t *ctx);

Parameters

param_val_t val

Value can be accessed as below

1. *bool* : val.val.b
2. *integer* : val.val.i
3. *float* : val.val.f
4. *char* * : val.val.s

ESP RainMaker Param API

Param class expose API's for creating custom parameters for the devices and report and update values associated with parameter to the ESP RainMaker cloud. Parameterized constructor is defined which creates custom parameter.

NOTE : *my_param* is the object of Param class.

```
Param my_param(const char *param_name, const char *param_type, param_val_t val, uint8_t␣
↪properties);
```

- `param_name` : Name of the parameter
- `param_type` : Type of the parameter. It is optional can be kept NULL.
- `val` : Define the default value for the parameter. It should be defined using *value(int ival)* , *value(bool bval)* , *value(float fval)* , *value(char *sval)*.
- **`properties`** [Properties of the parameter, which will be a logical OR of flags.]
  - **Flags**
    - PROP_FLAG_WRITE
    - PROP_FLAG_READ
    - PROP_FLAG_TIME_SERIES
    - PROP_FLAG_PERSIST

Sample example :

```
Param my_param(const char *param_name, const char *param_type, param_val_t val, uint8_t␣
↪properties);
Param my_param("bright", NULL, value(30), PROP_FLAG_READ | PROP_FLAG_WRITE | PROP_FLAG_
↪PERSIST);
```

NOTE : Parameter created using Param class should be added to the device using *my_device.addParam(my_param);*

my_param.addUIType

Add a UI type to the parameter. This will be used by the Phone apps (or other clients) to render appropriate UI for the given parameter. Please refer the RainMaker documentation here for supported UI Types.

```
esp_err_t addUIType(const char *ui_type);
```

- **`ui_type`** [String describing the UI Type.]
  - **Standard UI Types**
    - ESP_RMAKER_UI_TOGGLE
    - ESP_RMAKER_UI_SLIDER

        * ESP_RMAKER_UI_DROPDOWN
        * ESP_RMAKER_UI_TEXT

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

my_param.addBounds

Add bounds for an integer/float parameter. This can be used to add bounds (min/max values) for a given integer/float parameter. Eg. brightness will have bounds as 0 and 100 if it is a percentage.

```
esp_err_t addBounds(param_val_t min, param_val_t max, param_val_t step);
```

- `min` : Minimum value
- `max` : Maximum value
- `step` : step Minimum stepping

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

Sample example : my_param.addBounds(value(0), value(100), value(5));

my_param.updateAndReport

It updates the parameter and report it to ESP RainMaker cloud. This is called in callback.

```
esp_err_t updateAndReport(param_val_t val);
```

- `val` : New value of the parameter

This function will return

1. *ESP_OK* : On success
2. Error in case of failure

NOTE:

- This API should always be called inside device write callback, if you aimed at updating n reporting parameter values, changed via RainMaker Client (Phone App), to the ESP RainMaker cloud.
- If not called then paramter values will not be updated to the ESP RainMaker cloud.

printQR

This API displays QR code, which is used in provisioning.

```
printQR(const char *serv_name, const char *pop, const char *transport);
```

- `name` : Service name used in provisioning API.
- `pop` : Proof of posession used in provisioning API.
- **`transport :`**
    1. *softap* : In case of provisioning using SOFTAP.
    2. *ble* : In case of provisioning using BLE.

RMakerFactoryReset

Reset the device to factory defaults.

```
RMakerFactoryReset(int seconds);
```

- `seconds` : Time in seconds after which the chip should reboot after doing a factory reset.

RMakerWiFiReset

Reset Wi-Fi credentials.

```
RMakerWiFiReset(int seconds);
```

- `seconds` : Time in seconds after which the chip should reboot after doing a Wi-Fi reset.

2.3.17 Reset Reason

About

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Example

To get started with Reset Reason, you can try:

Reset Reason

```
/*
*  Print last reset reason of ESP32
*  ================================
*
*  Use either of the methods print_reset_reason
*  or verbose_print_reset_reason to display the
*  cause for the last reset of this device.
*
*  Public Domain License.
*
*  Author:
*  Evandro Luis Copercini - 2017
*/

#ifdef ESP_IDF_VERSION_MAJOR // IDF 4+
#if CONFIG_IDF_TARGET_ESP32 // ESP32/PICO-D4
#include "esp32/rom/rtc.h"
#elif CONFIG_IDF_TARGET_ESP32S2
#include "esp32s2/rom/rtc.h"
#elif CONFIG_IDF_TARGET_ESP32C3
#include "esp32c3/rom/rtc.h"
#elif CONFIG_IDF_TARGET_ESP32S3
#include "esp32s3/rom/rtc.h"
#else
#error Target CONFIG_IDF_TARGET is not supported
#endif
#else // ESP32 Before IDF 4.0
#include "rom/rtc.h"
#endif

#define uS_TO_S_FACTOR 1000000  /* Conversion factor for micro seconds to seconds */

void print_reset_reason(int reason)
{
  switch ( reason)
  {
    case 1 : Serial.println ("POWERON_RESET");break;          /**<1,  Vbat power on␣
↪reset*/
    case 3 : Serial.println ("SW_RESET");break;               /**<3,  Software reset␣
↪digital core*/
    case 4 : Serial.println ("OWDT_RESET");break;             /**<4,  Legacy watch dog␣
↪reset digital core*/
    case 5 : Serial.println ("DEEPSLEEP_RESET");break;        /**<5,  Deep Sleep reset␣
↪digital core*/
    case 6 : Serial.println ("SDIO_RESET");break;             /**<6,  Reset by SLC␣
↪module, reset digital core*/
    case 7 : Serial.println ("TG0WDT_SYS_RESET");break;       /**<7,  Timer Group0 Watch␣
↪dog reset digital core*/
    case 8 : Serial.println ("TG1WDT_SYS_RESET");break;       /**<8,  Timer Group1 Watch␣
↪dog reset digital core*/
```

(continues on next page)

(continued from previous page)

```
    case 9 : Serial.println ("RTCWDT_SYS_RESET");break;        /**<9,  RTC Watch dog␣
↪Reset digital core*/
    case 10 : Serial.println ("INTRUSION_RESET");break;        /**<10, Instrusion tested␣
↪to reset CPU*/
    case 11 : Serial.println ("TGWDT_CPU_RESET");break;        /**<11, Time Group reset␣
↪CPU*/
    case 12 : Serial.println ("SW_CPU_RESET");break;           /**<12, Software reset␣
↪CPU*/
    case 13 : Serial.println ("RTCWDT_CPU_RESET");break;       /**<13, RTC Watch dog␣
↪Reset CPU*/
    case 14 : Serial.println ("EXT_CPU_RESET");break;          /**<14, for APP CPU,␣
↪reseted by PRO CPU*/
    case 15 : Serial.println ("RTCWDT_BROWN_OUT_RESET");break;/**<15, Reset when the vdd␣
↪voltage is not stable*/
    case 16 : Serial.println ("RTCWDT_RTC_RESET");break;       /**<16, RTC Watch dog␣
↪reset digital core and rtc module*/
    default : Serial.println ("NO_MEAN");
  }
}

void verbose_print_reset_reason(int reason)
{
  switch ( reason)
  {
    case 1  : Serial.println ("Vbat power on reset");break;
    case 3  : Serial.println ("Software reset digital core");break;
    case 4  : Serial.println ("Legacy watch dog reset digital core");break;
    case 5  : Serial.println ("Deep Sleep reset digital core");break;
    case 6  : Serial.println ("Reset by SLC module, reset digital core");break;
    case 7  : Serial.println ("Timer Group0 Watch dog reset digital core");break;
    case 8  : Serial.println ("Timer Group1 Watch dog reset digital core");break;
    case 9  : Serial.println ("RTC Watch dog Reset digital core");break;
    case 10 : Serial.println ("Instrusion tested to reset CPU");break;
    case 11 : Serial.println ("Time Group reset CPU");break;
    case 12 : Serial.println ("Software reset CPU");break;
    case 13 : Serial.println ("RTC Watch dog Reset CPU");break;
    case 14 : Serial.println ("for APP CPU, reseted by PRO CPU");break;
    case 15 : Serial.println ("Reset when the vdd voltage is not stable");break;
    case 16 : Serial.println ("RTC Watch dog reset digital core and rtc module");break;
    default : Serial.println ("NO_MEAN");
  }
}

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(2000);

  Serial.println("CPU0 reset reason:");
  print_reset_reason(rtc_get_reset_reason(0));
  verbose_print_reset_reason(rtc_get_reset_reason(0));
```

(continues on next page)

(continued from previous page)

```
  Serial.println("CPU1 reset reason:");
  print_reset_reason(rtc_get_reset_reason(1));
  verbose_print_reset_reason(rtc_get_reset_reason(1));

  // Set ESP32 to go to deep sleep to see a variation
  // in the reset reason. Device will sleep for 5 seconds.
  esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
  Serial.println("Going to sleep");
  esp_deep_sleep(5 * uS_TO_S_FACTOR);
}

void loop() {
  // put your main code here, to run repeatedly:

}

/*
  Example Serial Log:
  ====================

rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0x00
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0008,len:8
load:0x3fff0010,len:160
load:0x40078000,len:10632
load:0x40080000,len:252
entry 0x40080034
CPU0 reset reason:
RTCWDT_RTC_RESET
RTC Watch dog reset digital core and rtc module
CPU1 reset reason:
EXT_CPU_RESET
for APP CPU, reseted by PRO CPU
Going to sleep
ets Jun  8 2016 00:22:57

rst:0x5 (DEEPSLEEP_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0x00
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0008,len:8
load:0x3fff0010,len:160
load:0x40078000,len:10632
load:0x40080000,len:252
entry 0x40080034
CPU0 reset reason:
DEEPSLEEP_RESET
Deep Sleep reset digital core
CPU1 reset reason:
EXT_CPU_RESET
```

(continues on next page)

(continued from previous page)

```
for APP CPU, reseted by PRO CPU
Going to sleep
ets Jun  8 2016 00:22:57

rst:0x5 (DEEPSLEEP_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0x00
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0008,len:8
load:0x3fff0010,len:160
load:0x40078000,len:10632
load:0x40080000,len:252
entry 0x40080034
CPU0 reset reason:
DEEPSLEEP_RESET
Deep Sleep reset digital core
CPU1 reset reason:
EXT_CPU_RESET
for APP CPU, reseted by PRO CPU
Going to sleep

*/
```

2.3.18 RMT

About

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Remote Control Transceiver (RMT) peripheral was designed to act as an infrared transceiver.

Example

To get started with RMT, you can try:

RMT Write Neo Pixel

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "Arduino.h"

#include "esp32-hal.h"

// The effect seen in ESP32C3, ESP32S2 and ESP32S3 is like a Blink of RGB LED
#if CONFIG_IDF_TARGET_ESP32S2
```

(continues on next page)

(continued from previous page)

```
#define BUILTIN_RGBLED_PIN   18
#elif CONFIG_IDF_TARGET_ESP32S3
#define BUILTIN_RGBLED_PIN   48
#elif CONFIG_IDF_TARGET_ESP32C3
#define BUILTIN_RGBLED_PIN   8
#else
#define BUILTIN_RGBLED_PIN   21   // ESP32 has no builtin RGB LED
#endif

#define NR_OF_LEDS   8*4
#define NR_OF_ALL_BITS 24*NR_OF_LEDS

//
// Note: This example uses Neopixel LED board, 32 LEDs chained one
//      after another, each RGB LED has its 24 bit value
//      for color configuration (8b for each color)
//
//      Bits encoded as pulses as follows:
//
//      "0":
//         +-------+              +--
//         |       |              |
//         |       |              |
//         |       |              |
//      ---|       |--------------|
//         +       +              +
//         | 0.4us |   0.85 0us   |
//
//      "1":
//         +--------------+       +--
//         |              |       |
//         |              |       |
//         |              |       |
//         |              |       |
//      ---+              +-------+
//         |    0.8us     | 0.4us |

rmt_data_t led_data[NR_OF_ALL_BITS];

rmt_obj_t* rmt_send = NULL;

void setup()
{
    Serial.begin(115200);

    if ((rmt_send = rmtInit(BUILTIN_RGBLED_PIN, RMT_TX_MODE, RMT_MEM_64)) == NULL)
    {
        Serial.println("init sender failed\n");
    }

    float realTick = rmtSetTick(rmt_send, 100);
    Serial.printf("real tick set to: %fns\n", realTick);
```

(continues on next page)

(continued from previous page)

```
}

int color[] =  { 0x55, 0x11, 0x77 };  // RGB value
int led_index = 0;

void loop()
{
    // Init data with only one led ON
    int led, col, bit;
    int i=0;
    for (led=0; led<NR_OF_LEDS; led++) {
        for (col=0; col<3; col++ ) {
            for (bit=0; bit<8; bit++){
                if ( (color[col] & (1<<(7-bit))) && (led == led_index) ) {
                    led_data[i].level0 = 1;
                    led_data[i].duration0 = 8;
                    led_data[i].level1 = 0;
                    led_data[i].duration1 = 4;
                } else {
                    led_data[i].level0 = 1;
                    led_data[i].duration0 = 4;
                    led_data[i].level1 = 0;
                    led_data[i].duration1 = 8;
                }
                i++;
            }
        }
    }
    // make the led travel in the pannel
    if ((++led_index)>=NR_OF_LEDS) {
        led_index = 0;
    }

    // Send the data
    rmtWrite(rmt_send, led_data, NR_OF_ALL_BITS);

    delay(100);
}
```

Complete list of RMT examples.

2.3.19 SDIO

About

Note: This peripheral is not supported yet by the Arduino API's.

2.3.20 SD MMC

About

Note: This is a work in progress project and this section is still missing. If you want to contribute, please see the Contributions Guide.

Example

To get started with SD_MMC, you can try:

SDMMC Test

```
/*
 * Connect the SD card to the following pins:
 *
 * SD Card | ESP32
 *    D2       12
 *    D3       13
 *    CMD      15
 *    VSS      GND
 *    VDD      3.3V
 *    CLK      14
 *    VSS      GND
 *    D0       2  (add 1K pull up after flashing)
 *    D1       4
 */

#include "FS.h"
#include "SD_MMC.h"

void listDir(fs::FS &fs, const char * dirname, uint8_t levels){
    Serial.printf("Listing directory: %s\n", dirname);

    File root = fs.open(dirname);
    if(!root){
        Serial.println("Failed to open directory");
        return;
    }
    if(!root.isDirectory()){
        Serial.println("Not a directory");
        return;
    }

    File file = root.openNextFile();
    while(file){
        if(file.isDirectory()){
            Serial.print("  DIR : ");
            Serial.println(file.name());
```

(continues on next page)

(continued from previous page)

```
            if(levels){
                listDir(fs, file.path(), levels -1);
            }
        } else {
            Serial.print("  FILE: ");
            Serial.print(file.name());
            Serial.print("  SIZE: ");
            Serial.println(file.size());
        }
        file = root.openNextFile();
    }
}

void createDir(fs::FS &fs, const char * path){
    Serial.printf("Creating Dir: %s\n", path);
    if(fs.mkdir(path)){
        Serial.println("Dir created");
    } else {
        Serial.println("mkdir failed");
    }
}

void removeDir(fs::FS &fs, const char * path){
    Serial.printf("Removing Dir: %s\n", path);
    if(fs.rmdir(path)){
        Serial.println("Dir removed");
    } else {
        Serial.println("rmdir failed");
    }
}

void readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\n", path);

    File file = fs.open(path);
    if(!file){
        Serial.println("Failed to open file for reading");
        return;
    }

    Serial.print("Read from file: ");
    while(file.available()){
        Serial.write(file.read());
    }
}

void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("Failed to open file for writing");
```

(continues on next page)

(continued from previous page)

```
        return;
    }
    if(file.print(message)){
        Serial.println("File written");
    } else {
        Serial.println("Write failed");
    }
}

void appendFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Appending to file: %s\n", path);

    File file = fs.open(path, FILE_APPEND);
    if(!file){
        Serial.println("Failed to open file for appending");
        return;
    }
    if(file.print(message)){
        Serial.println("Message appended");
    } else {
        Serial.println("Append failed");
    }
}

void renameFile(fs::FS &fs, const char * path1, const char * path2){
    Serial.printf("Renaming file %s to %s\n", path1, path2);
    if (fs.rename(path1, path2)) {
        Serial.println("File renamed");
    } else {
        Serial.println("Rename failed");
    }
}

void deleteFile(fs::FS &fs, const char * path){
    Serial.printf("Deleting file: %s\n", path);
    if(fs.remove(path)){
        Serial.println("File deleted");
    } else {
        Serial.println("Delete failed");
    }
}

void testFileIO(fs::FS &fs, const char * path){
    File file = fs.open(path);
    static uint8_t buf[512];
    size_t len = 0;
    uint32_t start = millis();
    uint32_t end = start;
    if(file){
        len = file.size();
        size_t flen = len;
        start = millis();
```

(continues on next page)

(continued from previous page)

```
        while(len){
            size_t toRead = len;
            if(toRead > 512){
                toRead = 512;
            }
            file.read(buf, toRead);
            len -= toRead;
        }
        end = millis() - start;
        Serial.printf("%u bytes read for %u ms\n", flen, end);
        file.close();
    } else {
        Serial.println("Failed to open file for reading");
    }


    file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("Failed to open file for writing");
        return;
    }

    size_t i;
    start = millis();
    for(i=0; i<2048; i++){
        file.write(buf, 512);
    }
    end = millis() - start;
    Serial.printf("%u bytes written for %u ms\n", 2048 * 512, end);
    file.close();
}

void setup(){
    Serial.begin(115200);
    if(!SD_MMC.begin()){
        Serial.println("Card Mount Failed");
        return;
    }
    uint8_t cardType = SD_MMC.cardType();

    if(cardType == CARD_NONE){
        Serial.println("No SD_MMC card attached");
        return;
    }

    Serial.print("SD_MMC Card Type: ");
    if(cardType == CARD_MMC){
        Serial.println("MMC");
    } else if(cardType == CARD_SD){
        Serial.println("SDSC");
    } else if(cardType == CARD_SDHC){
        Serial.println("SDHC");
```

(continues on next page)

(continued from previous page)

```
    } else {
        Serial.println("UNKNOWN");
    }

    uint64_t cardSize = SD_MMC.cardSize() / (1024 * 1024);
    Serial.printf("SD_MMC Card Size: %lluMB\n", cardSize);

    listDir(SD_MMC, "/", 0);
    createDir(SD_MMC, "/mydir");
    listDir(SD_MMC, "/", 0);
    removeDir(SD_MMC, "/mydir");
    listDir(SD_MMC, "/", 2);
    writeFile(SD_MMC, "/hello.txt", "Hello ");
    appendFile(SD_MMC, "/hello.txt", "World!\n");
    readFile(SD_MMC, "/hello.txt");
    deleteFile(SD_MMC, "/foo.txt");
    renameFile(SD_MMC, "/hello.txt", "/foo.txt");
    readFile(SD_MMC, "/foo.txt");
    testFileIO(SD_MMC, "/test.txt");
    Serial.printf("Total space: %lluMB\n", SD_MMC.totalBytes() / (1024 * 1024));
    Serial.printf("Used space: %lluMB\n", SD_MMC.usedBytes() / (1024 * 1024));
}

void loop(){

}
```

Complete list of SD MMC examples.

2.3.21 SigmaDelta

About

ESP32 provides a second-order sigma delta modulation module and 8 (4 for ESP32-C3) independent modulation channels. The channels are capable to output 1-bit signals (output index: 100 ~ 107) with sigma delta modulation.

ESP32 SoC	Number of SigmaDelta channels
ESP32	8
ESP32-S2	8
ESP32-C3	4
ESP32-S3	8

Arduino-ESP32 SigmaDelta API

sigmaDeltaSetup

This function is used to setup the SigmaDelta channel frequency and resolution.

```
uint32_t sigmaDeltaSetup(uint8_t pin, uint8_t channel, uint32_t freq);
```

- `pin` select GPIO pin.
- `channel` select SigmaDelta channel.
- `freq` select frequency.
  - range is 1-14 bits (1-20 bits for ESP32).

This function will return `frequency` configured for the SigmaDelta channel. If `0` is returned, error occurs and the SigmaDelta channel was not configured.

sigmaDeltaWrite

This function is used to set duty for the SigmaDelta channel.

```
void sigmaDeltaWrite(uint8_t channel, uint8_t duty);
```

- `channel` select SigmaDelta channel.
- `duty` select duty to be set for selected channel.

sigmaDeltaRead

This function is used to get configured duty for the SigmaDelta channel.

```
uint8_t sigmaDeltaRead(uint8_t channel)
```

- `channnel` select SigmaDelta channel.

This function will return `duty` configured for the selected SigmaDelta channel.

sigmaDeltaDetachPin

This function is used to detach pin from SigmaDelta.

```
void sigmaDeltaDetachPin(uint8_t pin);
```

- `pin` select GPIO pin.

Example Applications

Here is example use of SigmaDelta:

```
void setup()
{
    //setup on pin 18, channel 0 with frequency 312500 Hz
    sigmaDeltaSetup(18,0, 312500);
    //initialize channel 0 to off
    sigmaDeltaWrite(0, 0);
}

void loop()
{
    //slowly ramp-up the value
    //will overflow at 256
    static uint8_t i = 0;
    sigmaDeltaWrite(0, i++);
    delay(100);
}
```

2.3.22 SPI

About

For some APIs, the reference to be used is the same as the Arduino Core.

Arduino API Reference

SPI Reference

SPI Description

Example

To get started with SPI, you can try:

SPI Multiple Buses

```
/* The ESP32 has four SPi buses, however as of right now only two of
 * them are available to use, HSPI and VSPI. Simply using the SPI API
 * as illustrated in Arduino examples will use VSPI, leaving HSPI unused.
 *
 * However if we simply intialise two instance of the SPI class for both
 * of these buses both can be used. However when just using these the Arduino
 * way only will actually be outputting at a time.
 *
 * Logic analyser capture is in the same folder as this example as
 * "multiple_bus_output.png"
```

(continues on next page)

(continued from previous page)

```
 *
 * created 30/04/2018 by Alistair Symonds
 */
#include <SPI.h>

// Define ALTERNATE_PINS to use non-standard GPIO pins for SPI bus

#ifdef ALTERNATE_PINS
  #define VSPI_MISO   2
  #define VSPI_MOSI   4
  #define VSPI_SCLK   0
  #define VSPI_SS     33

  #define HSPI_MISO   26
  #define HSPI_MOSI   27
  #define HSPI_SCLK   25
  #define HSPI_SS     32
#else
  #define VSPI_MISO   MISO
  #define VSPI_MOSI   MOSI
  #define VSPI_SCLK   SCK
  #define VSPI_SS     SS

  #define HSPI_MISO   12
  #define HSPI_MOSI   13
  #define HSPI_SCLK   14
  #define HSPI_SS     15
#endif

#if CONFIG_IDF_TARGET_ESP32S2 || CONFIG_IDF_TARGET_ESP32S3
#define VSPI FSPI
#endif

static const int spiClk = 1000000; // 1 MHz

//uninitialised pointers to SPI objects
SPIClass * vspi = NULL;
SPIClass * hspi = NULL;

void setup() {
  //initialise two instances of the SPIClass attached to VSPI and HSPI respectively
  vspi = new SPIClass(VSPI);
  hspi = new SPIClass(HSPI);

  //clock miso mosi ss

#ifndef ALTERNATE_PINS
  //initialise vspi with default pins
  //SCLK = 18, MISO = 19, MOSI = 23, SS = 5
  vspi->begin();
#else
  //alternatively route through GPIO pins of your choice
```

(continues on next page)

(continued from previous page)

```
  vspi->begin(VSPI_SCLK, VSPI_MISO, VSPI_MOSI, VSPI_SS); //SCLK, MISO, MOSI, SS
#endif

#ifndef ALTERNATE_PINS
  //initialise hspi with default pins
  //SCLK = 14, MISO = 12, MOSI = 13, SS = 15
  hspi->begin();
#else
  //alternatively route through GPIO pins
  hspi->begin(HSPI_SCLK, HSPI_MISO, HSPI_MOSI, HSPI_SS); //SCLK, MISO, MOSI, SS
#endif

  //set up slave select pins as outputs as the Arduino API
  //doesn't handle automatically pulling SS low
  pinMode(vspi->pinSS(), OUTPUT); //VSPI SS
  pinMode(hspi->pinSS(), OUTPUT); //HSPI SS

}

// the loop function runs over and over again until power down or reset
void loop() {
  //use the SPI buses
  spiCommand(vspi, 0b01010101); // junk data to illustrate usage
  spiCommand(hspi, 0b11001100);
  delay(100);
}

void spiCommand(SPIClass *spi, byte data) {
  //use it as you would the regular arduino SPI API
  spi->beginTransaction(SPISettings(spiClk, MSBFIRST, SPI_MODE0));
  digitalWrite(spi->pinSS(), LOW); //pull SS slow to prep other end for transfer
  spi->transfer(data);
  digitalWrite(spi->pinSS(), HIGH); //pull ss high to signify end of data transfer
  spi->endTransaction();
}
```

2.3.23 Timer

About

The ESP32 SoCs contains from 2 to 4 hardware timers. They are all 64-bit (54-bit for ESP32-C3) generic timers based on 16-bit pre-scalers and 64-bit (54-bit for ESP32-C3) up / down counters which are capable of being auto-reloaded.

ESP32 SoC	Number of timers
ESP32	4
ESP32-S2	4
ESP32-C3	2
ESP32-S3	4

Arduino-ESP32 Timer API

timerBegin

This function is used to configure the timer. After successful setup the timer will automatically start.

```
hw_timer_t * timerBegin(uint8_t num, uint16_t divider, bool countUp);
```

- `num` select timer number.
- `divider` select timer divider. Sets how quickly the timer counter is "ticking".
- `countUp` select timer direction. Sets if the counter should be incrementing or decrementing.

This function will return `timer` structure if configuration is successful. If `NULL` is returned, error occurs and the timer was not configured.

timerEnd

This function is used to end timer.

```
void timerEnd(hw_timer_t *timer);
```

- `timer` timer struct.

timerSetConfig

This function is used to configure initialized timer (timerBegin() called).

```
uint32_t timerGetConfig(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `configuration` as uint32_t number. This can be translated by inserting it to struct `timer_cfg_t.val`.

timerAttachInterrupt

This function is used to attach interrupt to timer.

```
void timerAttachInterrupt(hw_timer_t *timer, void (*fn)(void), bool edge);
```

- `timer` timer struct.
- `fn` funtion to be called when interrupt is triggered.
- `edge` select edge to trigger interrupt (only LEVEL trigger is currently supported).

timerDetachInterrupt

This function is used to detach interrupt from timer.

```
void timerDetachInterrupt(hw_timer_t *timer);
```

- `timer` timer struct.

timerStart

This function is used to start counter of the timer.

```
void timerStart(hw_timer_t *timer);
```

- `timer` timer struct.

timerStop

This function is used to stop counter of the timer.

```
void timerStop(hw_timer_t *timer);
```

- `timer` timer struct.

timerRestart

This function is used to restart counter of the timer.

```
void timerRestart(hw_timer_t *timer);
```

- `timer` timer struct.

timerWrite

This function is used to set counter value of the timer.

```
void timerWrite(hw_timer_t *timer, uint64_t val);
```

- `timer` timer struct.
- `val` counter value to be set.

timerSetDivider

This function is used to set the divider of the timer.

```
void timerSetDivider(hw_timer_t *timer, uint16_t divider);
```

- `timer` timer struct.
- `divider` divider to be set.

timerSetCountUp

This function is used to configure counting direction of the timer.

```
void timerSetCountUp(hw_timer_t *timer, bool countUp);
```

- `timer` timer struct.
- `countUp` select counting direction (`true` = increment).

timerSetAutoReload

This function is used to set counter value of the timer.

```
void timerSetAutoReload(hw_timer_t *timer, bool autoreload);
```

- `timer` timer struct.
- `autoreload` select autoreload (`true` = enabled).

timerStarted

This function is used to get if the timer is running.

```
bool timerStarted(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `true` if the timer is running. If `false` is returned, timer is stopped.

timerRead

This function is used to read counter value of the timer.

```
uint64_t timerRead(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `counter value` of the timer.

timerReadMicros

This function is used to read counter value in microseconds of the timer.

```
uint64_t timerReadMicros(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `counter value` of the timer in microseconds.

timerReadMilis

This function is used to read counter value in miliseconds of the timer.

```
uint64_t timerReadMilis(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `counter value` of the timer in miliseconds.

timerReadSeconds

This function is used to read counter value in seconds of the timer.

```
double timerReadSeconds(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `counter value` of the timer in seconds.

timerGetDivider

This function is used to get divider of the timer.

```
uint16_t timerGetDivider(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `divider` of the timer.

timerGetCountUp

This function is used get counting direction of the timer.

```
bool timerGetCountUp(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `true` if the timer counting direction is UP (incrementing). If `false` returned, the timer counting direction is DOWN (decrementing).

timerGetAutoReload

This function is used to get configuration of auto reload of the timer.

```
bool timerGetAutoReload(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `true` if the timer auto reload is enabled. If `false` returned, the timer auto reload is disabled.

timerAlarmEnable

This function is used to enable generation of timer alarm events.

```
void timerAlarmEnable(hw_timer_t *timer);
```

- `timer` timer struct.

timerAlarmDisable

This function is used to disable generation of timer alarm events.

```
void timerAlarmDisable(hw_timer_t *timer);
```

- `timer` timer struct.

timerAlarmWrite

This function is used to configure alarm value and autoreload of the timer.

```
void timerAlarmWrite(hw_timer_t *timer, uint64_t alarm_value, bool autoreload);
```

- `timer` timer struct.
- `alarm_value` alarm value to generate event.
- `autoreload` enabled/disabled autorealod.

timerAlarmEnabled

This function is used to get status of timer alarm.

```
bool timerAlarmEnabled(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `true` if the timer alarm is enabled. If `false` returned, the timer alarm is disabled.

timerAlarmRead

This function is used to read alarm value of the timer.

```
uint64_t timerAlarmRead(hw_timer_t *timer);
```

- `timer` timer struct.

timerAlarmReadMicros

This function is used to read alarm value of the timer in microseconds.

```
uint64_t timerAlarmReadMicros(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `alarm value` of the timer in microseconds.

timerAlarmReadSeconds

This function is used to read alarm value of the timer in seconds.

```
double timerAlarmReadSeconds(hw_timer_t *timer);
```

- `timer` timer struct.

This function will return `alarm value` of the timer in seconds.

Example Applications

There are 2 examples uses of Timer:

Repeat timer example:

```
/*
 Repeat timer example

 This example shows how to use hardware timer in ESP32. The timer calls onTimer
 function every second. The timer can be stopped with button attached to PIN 0
 (IO0).

 This example code is in the public domain.
 */

// Stop button is attached to PIN 0 (IO0)
#define BTN_STOP_ALARM    0

hw_timer_t * timer = NULL;
volatile SemaphoreHandle_t timerSemaphore;
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;

volatile uint32_t isrCounter = 0;
```

(continues on next page)

(continued from previous page)

```
volatile uint32_t lastIsrAt = 0;

void ARDUINO_ISR_ATTR onTimer(){
  // Increment the counter and set the time of ISR
  portENTER_CRITICAL_ISR(&timerMux);
  isrCounter++;
  lastIsrAt = millis();
  portEXIT_CRITICAL_ISR(&timerMux);
  // Give a semaphore that we can check in the loop
  xSemaphoreGiveFromISR(timerSemaphore, NULL);
  // It is safe to use digitalRead/Write here if you want to toggle an output
}

void setup() {
  Serial.begin(115200);

  // Set BTN_STOP_ALARM to input mode
  pinMode(BTN_STOP_ALARM, INPUT);

  // Create semaphore to inform us when the timer has fired
  timerSemaphore = xSemaphoreCreateBinary();

  // Use 1st timer of 4 (counted from zero).
  // Set 80 divider for prescaler (see ESP32 Technical Reference Manual for more
  // info).
  timer = timerBegin(0, 80, true);

  // Attach onTimer function to our timer.
  timerAttachInterrupt(timer, &onTimer, true);

  // Set alarm to call onTimer function every second (value in microseconds).
  // Repeat the alarm (third parameter)
  timerAlarmWrite(timer, 1000000, true);

  // Start an alarm
  timerAlarmEnable(timer);
}

void loop() {
  // If Timer has fired
  if (xSemaphoreTake(timerSemaphore, 0) == pdTRUE){
    uint32_t isrCount = 0, isrTime = 0;
    // Read the interrupt count and time
    portENTER_CRITICAL(&timerMux);
    isrCount = isrCounter;
    isrTime = lastIsrAt;
    portEXIT_CRITICAL(&timerMux);
    // Print it
    Serial.print("onTimer no. ");
    Serial.print(isrCount);
    Serial.print(" at ");
    Serial.print(isrTime);
```

(continues on next page)

(continued from previous page)

```
    Serial.println(" ms");
  }
  // If button is pressed
  if (digitalRead(BTN_STOP_ALARM) == LOW) {
    // If timer is still running
    if (timer) {
      // Stop and free timer
      timerEnd(timer);
      timer = NULL;
    }
  }
}
```

Watchdog timer example:

```
#include "esp_system.h"

const int button = 0;         //gpio to use to trigger delay
const int wdtTimeout = 3000;  //time in ms to trigger the watchdog
hw_timer_t *timer = NULL;

void ARDUINO_ISR_ATTR resetModule() {
  ets_printf("reboot\n");
  esp_restart();
}

void setup() {
  Serial.begin(115200);
  Serial.println();
  Serial.println("running setup");

  pinMode(button, INPUT_PULLUP);                    //init control pin
  timer = timerBegin(0, 80, true);                  //timer 0, div 80
  timerAttachInterrupt(timer, &resetModule, true);  //attach callback
  timerAlarmWrite(timer, wdtTimeout * 1000, false); //set time in us
  timerAlarmEnable(timer);                          //enable interrupt
}

void loop() {
  Serial.println("running main loop");

  timerWrite(timer, 0); //reset timer (feed watchdog)
  long loopTime = millis();
  //while button is pressed, delay up to 3 seconds to trigger the timer
  while (!digitalRead(button)) {
    Serial.println("button pressed");
    delay(500);
  }
  delay(1000); //simulate work
  loopTime = millis() - loopTime;

  Serial.print("loop time is = ");
```

(continues on next page)

(continued from previous page)

```
  Serial.println(loopTime); //should be under 3000
}
```

2.3.24 TOUCH

About

Touch sensor is a peripheral, that has an internal oscilator circuit and it measures charge/discharge frequency over a fixed period of time on respective GPIO pins. Therefore these touch sensors are also known as capacitive sensors. For example, if you touch any of these pins, finger electrical charge will change this number of cycles, by changing the RC circuit attached to the touch sensor. The TouchRead() will return the number of cycles (charges/discharges) in a certain time (meas). The change of this count will be used to validate if a touch has happened or not. These pins can be easily integrated into capacitive pads, and replace mechanical buttons.

Note: Touch peripheral is not present in every SoC. Refer to datasheet of each chip for more info.

Arduino-ESP32 TOUCH API

TOUCH common API

touchRead

This function gets the touch sensor data. Each touch sensor has a counter to count the number of charge/discharge cycles. When the pad is 'touched', the value in the counter will change because of the larger equivalent capacitance. The change of the data determines if the pad has been touched or not.

```
touch_value_t touchRead(uint8_t pin);
```

- `pin` GPIO pin to read TOUCH value

This function will return touch pad value as uint16_t (ESP32) or uint32_t (ESP32-S2/S3).

touchSetCycles

This function is used to set cycles that measurement operation takes. The result from touchRead, threshold and detection accuracy depend on these values. The defaults are setting touchRead to take ~0.5ms.

```
void touchSetCycles(uint16_t measure, uint16_t sleep);
```

- `measure` Sets the time that it takes to measure touch sensor value
- `sleep` Sets waiting time before next measure cycle

touchAttachInterrupt

This function is used to attach interrupt to the touch pad. The function will be called if a touch sensor value falls below the given threshold for ESP32 or rises above the given threshold for ESP32-S2/S3. To determine a proper threshold value between touched and untouched state, use touchRead() function.

```
void touchAttachInterrupt(uint8_t pin, void (*userFunc)(void), touch_value_t threshold);
```

- `pin` GPIO TOUCH pad pin
- `userFunc` Function to be called when interrupt is triggered
- `threshold` Sets the threshold when to call interrupt

touchAttachInterruptArg

This function is used to attach interrupt to the touch pad. In the function called by ISR you have the given arguments available.

```
void touchAttachInterruptArg(uint8_t pin, void (*userFunc)(void*), void *arg, touch_
↪value_t threshold);
```

- `pin` GPIO TOUCH pad pin
- `userFunc` Function to be called when interrupt is triggered
- `arg` Sets arguments to the interrupt
- `threshold` Sets the threshold when to call interrupt

touchDetachInterrupt

This function is used to detach interrupt from the touch pad.

```
void touchDetachInterrupt(uint8_t pin);
```

- `pin` GPIO TOUCH pad pin.

touchSleepWakeUpEnable

This function is used to setup touch pad as the wake up source from the deep sleep.

Note: ESP32-S2 and ESP32-S3 only support one sleep wake up touch pad.

```
void touchSleepWakeUpEnable(uint8_t pin, touch_value_t threshold);
```

- `pin` GPIO TOUCH pad pin
- `threshold` Sets the threshold when to wake up

TOUCH API specific for ESP32 chip (TOUCH_V1)

touchInterruptSetThresholdDirection

This function is used to tell the driver if it shall activate the interrupt if the sensor is lower or higher than the threshold value. Default is lower.

```
void touchInterruptSetThresholdDirection(bool mustbeLower);
```

TOUCH API specific for ESP32S2 and ESP32S3 chip (TOUCH_V2)

touchInterruptGetLastStatus

This function is used get the lastest ISR status for the touch pad.

```
bool touchInterruptGetLastStatus(uint8_t pin);
```

This function returns true if the touch pad has been and continues pressed or false otherwise.

Example Applications

Example of reading the touch sensor.

```
// ESP32 Touch Test
// Just test touch pin - Touch0 is T0 which is on GPIO 4.

void setup()
{
  Serial.begin(115200);
  delay(1000); // give me time to bring up serial monitor
  Serial.println("ESP32 Touch Test");
}

void loop()
{
  Serial.println(touchRead(T1));  // get value using T1
  delay(1000);
}
```

A usage example for the touch interrupts.

```
/*
This is an example how to use Touch Intrrerupts
The bigger the threshold, the more sensible is the touch
*/

int threshold = 40;
bool touch1detected = false;
bool touch2detected = false;
```

(continues on next page)

(continued from previous page)

```
void gotTouch1(){
 touch1detected = true;
}

void gotTouch2(){
 touch2detected = true;
}

void setup() {
  Serial.begin(115200);
  delay(1000); // give me time to bring up serial monitor
  Serial.println("ESP32 Touch Interrupt Test");
  touchAttachInterrupt(T2, gotTouch1, threshold);
  touchAttachInterrupt(T3, gotTouch2, threshold);
}

void loop(){
  if(touch1detected){
    touch1detected = false;
    Serial.println("Touch 1 detected");
  }
  if(touch2detected){
    touch2detected = false;
    Serial.println("Touch 2 detected");
  }
}
```

More examples can be found in our repository -> Touch examples.

2.3.25 USB API

Note: This feature is only supported on ESP chips that have USB peripheral, like the ESP32-S2 and ESP32-S3. Some chips, like the ESP32-C3 include native CDC+JTAG peripheral that is not covered here.

About

The **Universal Serial Bus** is a widely used peripheral to exchange data between devices. USB was introduced on the ESP32, supporting both device and host mode.

To learn about the USB, see the USB.org for developers.

USB as Device

In the device mode, the ESP32 acts as an USB device, like a mouse or keyboard to be connected to a host device, like your computer or smartphone.

USB as Host

The USB host mode, you can connect devices on the ESP32, like external modems, mouse and keyboards.

Note: This mode is still under development for the ESP32.

API Description

This is the common USB API description.

For more supported USB classes implementation, see the following sections:

USB CDC

About

USB Communications Device Class API. This class is used to enable communication between the host and the device.

This class is often used to enable serial communication and can be used to flash the firmware on the ESP32 without the external USB to Serial chip.

APIs

onEvent

Event handling functions.

```
void onEvent(esp_event_handler_t callback);
```

```
void onEvent(arduino_usb_cdc_event_t event, esp_event_handler_t callback);
```

Where `event` can be:

- ARDUINO_USB_CDC_ANY_EVENT
- ARDUINO_USB_CDC_CONNECTED_EVENT
- ARDUINO_USB_CDC_DISCONNECTED_EVENT

- ARDUINO_USB_CDC_LINE_STATE_EVENT
- ARDUINO_USB_CDC_LINE_CODING_EVENT
- ARDUINO_USB_CDC_RX_EVENT
- ARDUINO_USB_CDC_TX_EVENT
- ARDUINO_USB_CDC_RX_OVERFLOW_EVENT
- ARDUINO_USB_CDC_MAX_EVENT

setRxBufferSize

The `setRxBufferSize` function is used to set the size of the RX buffer.

```
size_t setRxBufferSize(size_t size);
```

setTxTimeoutMs

This function is used to define the time to reach the timeout for the TX.

```
void setTxTimeoutMs(uint32_t timeout);
```

begin

This function is used to start the peripheral using the default CDC configuration.

```
void begin(unsigned long baud);
```

Where:

- `baud` is the baud rate.

end

This function will finish the peripheral as CDC and release all the allocated resources. After calling `end` you need to use `begin` again in order to initialize the USB CDC driver again.

```
void end();
```

available

This function will return if there are messages in the queue.

```
int available(void);
```

The return is the number of bytes available to read.

availableForWrite

This function will return if the hardware is available to write data.

```
int availableForWrite(void);
```

peek

This function is used to `peek` messages from the queue.

```
int peek(void);
```

read

This function is used to read the bytes available.

```
size_t read(uint8_t *buffer, size_t size);
```

Where:

- `buffer` is the pointer to the buffer to be read.
- `size` is the number of bytes to be read.

write

This function is used to write the message.

```
size_t write(const uint8_t *buffer, size_t size);
```

Where:

- `buffer` is the pointer to the buffer to be written.
- `size` is the number of bytes to be written.

flush

This function is used to flush the data.

```
void flush(void);
```

baudRate

This function is used to get the `baudRate`.

```
uint32_t baudRate();
```

setDebugOutput

This function will enable the debug output, usually from the *UART0*, to the USB CDC.

```
void setDebugOutput(bool);
```

enableReboot

This function enables the device to reboot by the DTR as RTS signals.

```
void enableReboot(bool enable);
```

rebootEnabled

This function will return if the reboot is enabled.

```
bool rebootEnabled(void);
```

Example Code

Here is an example of how to use the USB CDC.

USBSerial

```
#if ARDUINO_USB_MODE
#warning This sketch should be used when USB is in OTG mode
void setup(){}
void loop(){}
#else
#include "USB.h"

#if ARDUINO_USB_CDC_ON_BOOT
#define HWSerial Serial0
#define USBSerial Serial
#else
#define HWSerial Serial
USBCDC USBSerial;
#endif

static void usbEventCallback(void* arg, esp_event_base_t event_base, int32_t event_id,␣
↪void* event_data){
```

(continues on next page)

(continued from previous page)

```
  if(event_base == ARDUINO_USB_EVENTS){
    arduino_usb_event_data_t * data = (arduino_usb_event_data_t*)event_data;
    switch (event_id){
      case ARDUINO_USB_STARTED_EVENT:
        HWSerial.println("USB PLUGGED");
        break;
      case ARDUINO_USB_STOPPED_EVENT:
        HWSerial.println("USB UNPLUGGED");
        break;
      case ARDUINO_USB_SUSPEND_EVENT:
        HWSerial.printf("USB SUSPENDED: remote_wakeup_en: %u\n", data->suspend.remote_
↪wakeup_en);
        break;
      case ARDUINO_USB_RESUME_EVENT:
        HWSerial.println("USB RESUMED");
        break;

      default:
        break;
    }
  } else if(event_base == ARDUINO_USB_CDC_EVENTS){
    arduino_usb_cdc_event_data_t * data = (arduino_usb_cdc_event_data_t*)event_data;
    switch (event_id){
      case ARDUINO_USB_CDC_CONNECTED_EVENT:
        HWSerial.println("CDC CONNECTED");
        break;
      case ARDUINO_USB_CDC_DISCONNECTED_EVENT:
        HWSerial.println("CDC DISCONNECTED");
        break;
      case ARDUINO_USB_CDC_LINE_STATE_EVENT:
        HWSerial.printf("CDC LINE STATE: dtr: %u, rts: %u\n", data->line_state.dtr, data-
↪>line_state.rts);
        break;
      case ARDUINO_USB_CDC_LINE_CODING_EVENT:
        HWSerial.printf("CDC LINE CODING: bit_rate: %u, data_bits: %u, stop_bits: %u,␣
↪parity: %u\n", data->line_coding.bit_rate, data->line_coding.data_bits, data->line_
↪coding.stop_bits, data->line_coding.parity);
        break;
      case ARDUINO_USB_CDC_RX_EVENT:
        HWSerial.printf("CDC RX [%u]:", data->rx.len);
        {
            uint8_t buf[data->rx.len];
            size_t len = USBSerial.read(buf, data->rx.len);
            HWSerial.write(buf, len);
        }
        HWSerial.println();
        break;
       case ARDUINO_USB_CDC_RX_OVERFLOW_EVENT:
        HWSerial.printf("CDC RX Overflow of %d bytes", data->rx_overflow.dropped_bytes);
        break;

      default:
```

(continues on next page)

(continued from previous page)

```
        break;
    }
  }
}

void setup() {
  HWSerial.begin(115200);
  HWSerial.setDebugOutput(true);

  USB.onEvent(usbEventCallback);
  USBSerial.onEvent(usbEventCallback);

  USBSerial.begin();
  USB.begin();
}

void loop() {
  while(HWSerial.available()){
    size_t l = HWSerial.available();
    uint8_t b[l];
    l = HWSerial.read(b, l);
    USBSerial.write(b, l);
  }
}
#endif /* ARDUINO_USB_MODE */
```

USB MSC

About

USB Mass Storage Class API. This class makes the device accessible as a mass storage device and allows you to transfer data between the host and the device.

One of the examples for this mode is to flash the device by dropping the firmware binary like a flash memory device when connecting the ESP32 to the host computer.

APIs

begin

This function is used to start the peripheral using the default MSC configuration.

```
bool begin(uint32_t block_count, uint16_t block_size);
```

Where:

- `block_count` set the disk sector count.
- `block_size` set the disk sector size.

This function will return `true` if the configuration was successful.

end

This function will finish the peripheral as MSC and release all the allocated resources. After calling `end` you need to use `begin` again in order to initialize the USB MSC driver again.

```
void end();
```

vendorID

This function is used to define the vendor ID.

```
void vendorID(const char * vid);//max 8 chars
```

productID

This function is used to define the product ID.

```
void productID(const char * pid);//max 16 chars
```

productRevision

This function is used to define the product revision.

```
void productRevision(const char * ver);//max 4 chars
```

mediaPresent

Set the `mediaPresent` configuration.

```
void mediaPresent(bool media_present);
```

onStartStop

Set the `onStartStop` callback function.

```
void onStartStop(msc_start_stop_cb cb);
```

onRead

Set the `onRead` callback function.

```
void onRead(msc_read_cb cb);
```

onWrite

Set the onWrite callback function.

```
void onWrite(msc_write_cb cb);
```

Example Code

Here is an example of how to use the USB MSC.

FirmwareMSC

```
#if ARDUINO_USB_MODE
#warning This sketch should be used when USB is in OTG mode
void setup(){}
void loop(){}
#else
#include "USB.h"
#include "FirmwareMSC.h"

#if !ARDUINO_USB_MSC_ON_BOOT
FirmwareMSC MSC_Update;
#endif
#if ARDUINO_USB_CDC_ON_BOOT
#define HWSerial Serial0
#define USBSerial Serial
#else
#define HWSerial Serial
USBCDC USBSerial;
#endif

static void usbEventCallback(void* arg, esp_event_base_t event_base, int32_t event_id,␣
↪void* event_data){
  if(event_base == ARDUINO_USB_EVENTS){
    arduino_usb_event_data_t * data = (arduino_usb_event_data_t*)event_data;
    switch (event_id){
      case ARDUINO_USB_STARTED_EVENT:
        HWSerial.println("USB PLUGGED");
        break;
      case ARDUINO_USB_STOPPED_EVENT:
        HWSerial.println("USB UNPLUGGED");
        break;
      case ARDUINO_USB_SUSPEND_EVENT:
        HWSerial.printf("USB SUSPENDED: remote_wakeup_en: %u\n", data->suspend.remote_
↪wakeup_en);
        break;
      case ARDUINO_USB_RESUME_EVENT:
        HWSerial.println("USB RESUMED");
        break;
```

(continues on next page)

(continued from previous page)

```
      default:
        break;
    }
  } else if(event_base == ARDUINO_FIRMWARE_MSC_EVENTS){
    arduino_firmware_msc_event_data_t * data = (arduino_firmware_msc_event_data_t*)event_
↪data;
    switch (event_id){
      case ARDUINO_FIRMWARE_MSC_START_EVENT:
        HWSerial.println("MSC Update Start");
        break;
      case ARDUINO_FIRMWARE_MSC_WRITE_EVENT:
        //HWSerial.printf("MSC Update Write %u bytes at offset %u\n", data->write.size,␣
↪data->write.offset);
        HWSerial.print(".");
        break;
      case ARDUINO_FIRMWARE_MSC_END_EVENT:
        HWSerial.printf("\nMSC Update End: %u bytes\n", data->end.size);
        break;
      case ARDUINO_FIRMWARE_MSC_ERROR_EVENT:
        HWSerial.printf("MSC Update ERROR! Progress: %u bytes\n", data->error.size);
        break;
      case ARDUINO_FIRMWARE_MSC_POWER_EVENT:
        HWSerial.printf("MSC Update Power: power: %u, start: %u, eject: %u", data->power.
↪power_condition, data->power.start, data->power.load_eject);
        break;

      default:
        break;
    }
  }
}

void setup() {
  HWSerial.begin(115200);
  HWSerial.setDebugOutput(true);

  USB.onEvent(usbEventCallback);
  MSC_Update.onEvent(usbEventCallback);
  MSC_Update.begin();
  USBSerial.begin();
  USB.begin();
}

void loop() {
  // put your main code here, to run repeatedly
}
#endif /* ARDUINO_USB_MODE */
```

USB Common

These are the common APIs for the USB driver.

onEvent

Event handling function to set the callback.

```
void onEvent(esp_event_handler_t callback);
```

Event handling function for the specific event.

```
void onEvent(arduino_usb_event_t event, esp_event_handler_t callback);
```

Where `event` can be:

- ARDUINO_USB_ANY_EVENT
- ARDUINO_USB_STARTED_EVENT
- ARDUINO_USB_STOPPED_EVENT
- ARDUINO_USB_SUSPEND_EVENT
- ARDUINO_USB_RESUME_EVENT
- ARDUINO_USB_MAX_EVENT

VID

Set the Vendor ID. This 16 bits identification is used to identify the company that develops the product.

Note: You can't define your own VID. If you need your own VID, you need to buy one. See https://www.usb.org/getting-vendor-id for more details.

```
bool VID(uint16_t v);
```

Get the Vendor ID.

```
uint16_t VID(void);
```

Returns the Vendor ID. The default value for the VID is: `0x303A`.

PID

Set the Product ID. This 16 bits identification is used to identify the product.

```
bool PID(uint16_t p);
```

Get the Product ID.

```
uint16_t PID(void);
```

Returns the Product ID. The default PID is: `0x0002`.

firmwareVersion

Set the firmware version. This is a 16 bits unsigned value.

```
bool firmwareVersion(uint16_t version);
```

Get the firmware version.

```
uint16_t firmwareVersion(void);
```

Return the 16 bits unsigned value. The default value is: `0x100`.

usbVersion

Set the USB version.

```
bool usbVersion(uint16_t version);
```

Get the USB version.

```
uint16_t usbVersion(void);
```

Return the USB version. The default value is: `0x200` (USB 2.0).

usbPower

Set the USB power as mA (current).

Note: This configuration does not change the physical power output. This is only used for the USB device information.

```
bool usbPower(uint16_t mA);
```

Get the USB power configuration.

```
uint16_t usbPower(void);
```

Return the current in mA. The default value is: `0x500` (500mA).

usbClass

Set the USB class.

```
bool usbClass(uint8_t _class);
```

Get the USB class.

```
uint8_t usbClass(void);
```

Return the USB class. The default value is: `TUSB_CLASS_MISC`.

usbSubClass

Set the USB sub-class.

```
bool usbSubClass(uint8_t subClass);
```

Get the USB sub-class.

```
uint8_t usbSubClass(void);
```

Return the USB sub-class. The default value is: `MISC_SUBCLASS_COMMON`.

usbProtocol

Define the USB protocol.

```
bool usbProtocol(uint8_t protocol);
```

Get the USB protocol.

```
uint8_t usbProtocol(void);
```

Return the USB protocol. The default value is: `MISC_PROTOCOL_IAD`

usbAttributes

Set the USB attributes.

```
bool usbAttributes(uint8_t attr);
```

Get the USB attributes.

```
uint8_t usbAttributes(void);
```

Return the USB attributes. The default value is: `TUSB_DESC_CONFIG_ATT_SELF_POWERED`

webUSB

This function is used to enable the `webUSB` functionality.

```
bool webUSB(bool enabled);
```

This function is used to get the `webUSB` setting.

```
bool webUSB(void);
```

Return the `webUSB` setting (*Enabled* or *Disabled*)

productName

This function is used to define the product name.

```
bool productName(const char * name);
```

This function is used to get the product's name.

```
const char * productName(void);
```

manufacturerName

This function is used to define the manufacturer name.

```
bool manufacturerName(const char * name);
```

This function is used to get the manufacturer's name.

```
const char * manufacturerName(void);
```

serialNumber

This function is used to define the serial number.

```
bool serialNumber(const char * name);
```

This function is used to get the serial number.

```
const char * serialNumber(void);
```

The default serial number is: `0`.

webUSBURL

This function is used to define the `webUSBURL`.

```
bool webUSBURL(const char * name);
```

This function is used to get the `webUSBURL`.

```
const char * webUSBURL(void);
```

The default `webUSBURL` is: https://espressif.github.io/arduino-esp32/webusb.html

enableDFU

This function is used to enable the DFU capability.

```
bool enableDFU();
```

begin

This function is used to start the peripheral using the default configuration.

```
bool begin();
```

Example Code

There are a collection of USB device examples on the project GitHub, including Firmware MSC update, USB CDC, HID and composite device.

2.3.26 Wi-Fi API

About

The Wi-Fi API provides support for the 802.11b/g/n protocol driver. This API includes:

- Station mode (STA mode or Wi-Fi client mode). ESP32 connects to an access point
- AP mode (aka Soft-AP mode or Access Point mode). Devices connect to the ESP32
- Security modes (WPA2, WPA3 etc.)
- Scanning for access points

Working as AP

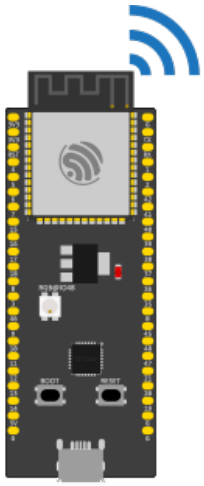
In this mode, the ESP32 is configured as an Access Point (AP) and it's capable of receiving incoming connections from other devices (stations) by providing a Wi-Fi network.

This mode can be used for serving an HTTP or HTTPS server inside the ESP32, for example.

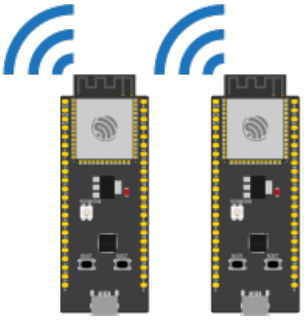
Working as STA

The STA mode is used to connect the ESP32 to a Wi-Fi network, provided by an Access Point.

This is the mode to be used if you want to connect your project to the Internet.

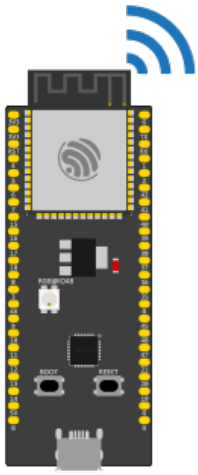


ESP32 as Wi-Fi Access Point (AP)

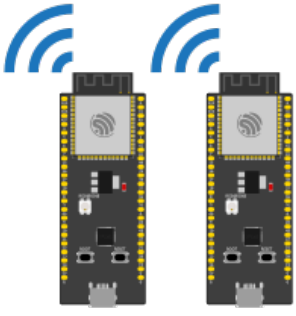


ESP32' as Wi-Fi Station (STA)

Wi-Fi Access Point (AP)



ESP32 as Wi-Fi Station (STA)



ESP32' as Wi-Fi Station (STA)

API Description

Here is the description of the WiFi API.

Common API

Here are the common APIs that are used for both modes, AP and STA.

useStaticBuffers

This function is used to set the memory allocation mode for the Wi-Fi buffers.

```
static void useStaticBuffers(bool bufferMode);
```

- Set `true` to use the Wi-Fi buffers memory allocation as **static**.
- Set `false` to set the buffers memory allocation to **dynamic**.

The use of dynamic allocation is recommended to save memory and reduce resources usage. However, the dynamic performs slightly slower than the static allocation. Use static allocation if you want to have more performance and if your application is multi-tasking.

By default, the memory allocation will be set to **dynamic** if this function is not being used.

setDualAntennaConfig

Configures the Dual antenna functionallity. This function should be used only on the **ESP32-WROOM-DA** module or any other ESP32 with RF switch.

```
bool setDualAntennaConfig(uint8_t gpio_ant1, uint8_t gpio_ant2, wifi_rx_ant_t rx_mode,␣
↪wifi_tx_ant_t tx_mode);
```

- `gpio_ant1` Configure the GPIO number for the antenna 1 connected to the RF switch (default `GPIO2` on ESP32-WROOM-DA)
- `gpio_ant2` Configure the GPIO number for the antenna 2 connected to the RF switch (default `GPIO25` on ESP32-WROOM-DA)
- `rx_mode` Set the RX antenna mode. See wifi_rx_ant_t for the options.
- `tx_mode` Set the TX antenna mode. See wifi_tx_ant_t for the options.

Return `true` if the configuration was successful.

For the `rx_mode` you can use the following configuration:

- `WIFI_RX_ANT0` Selects the antenna 1 for all RX activity.
- `WIFI_RX_ANT1` Selects the antenna 2 for all RX activity.
- `WIFI_RX_ANT_AUTO` Selects the antenna for RX automatically.

For the `tx_mode` you can use the following configuration:

- `WIFI_TX_ANT0` Selects the antenna 1 for all TX activity.
- `WIFI_TX_ANT1` Selects the antenna 2 for all TX activity.
- `WIFI_TX_ANT_AUTO` Selects the antenna for TX automatically.

WiFiAP

The `WiFiAP` is used to configure and manage the Wi-Fi as an Access Point. This is where you can find the related functions for the AP.

Basic Usage

To start the Wi-Fi as an Access Point.

```
WiFi.softAP(ssid, password);
```

Please see the full WiFiAP example in: *ap example*.

AP Configuration

softAP

Use the function `softAP` to configure the Wi-Fi AP characteristics:

```
bool softAP(const char* ssid, const char* passphrase = NULL, int channel = 1, int ssid_
↪hidden = 0, int max_connection = 4, bool ftm_responder = false);
```

Where:

- `ssid` sets the Wi-Fi network SSID.
- `passphrase` sets the Wi-Fi network password. If the network is open, set as `NULL`.
- `channel` configures the Wi-Fi channel.
- `ssid_hidden` sets the network as hidden.
- `max_connection` sets the maximum number of simultaneous connections. The default is 4.
- `ftm_responder` sets the Wi-Fi FTM responder feature. **Only for ESP32-S2 and ESP32-C3 SoC!**

Return `true` if the configuration was successful.

softAPConfig

Function used to configure the IP as static (fixed) as well as the gateway and subnet.

```
bool softAPConfig(IPAddress local_ip, IPAddress gateway, IPAddress subnet);
```

Where:

- `local_ip` sets the local IP address.
- `gateway` sets the gateway IP.
- `subnet` sets the subnet mask.

The function will return `true` if the configuration is successful.

AP Connection

softAPdisconnect

Function used to force the AP disconnection.

```
bool softAPdisconnect(bool wifioff = false);
```

Where:

- `wifioff` sets the Wi-Fi off if `true`.

The function will return `true` if the configuration is successful.

softAPgetStationNum

This function returns the number of clients connected to the AP.

```
uint8_t softAPgetStationNum();
```

softAPIP

Function to get the AP IPv4 address.

```
IPAddress softAPIP();
```

The function will return the AP IP address in `IPAddress` format.

softAPBroadcastIP

Function to get the AP IPv4 broadcast address.

```
IPAddress softAPBroadcastIP();
```

The function will return the AP broadcast address in `IPAddress` format.

softAPNetworkID

Get the softAP network ID.

```
IPAddress softAPNetworkID();
```

The function will return the AP network address in `IPAddress` format.

softAPSubnetCIDR

Get the softAP subnet CIDR.

```
uint8_t softAPSubnetCIDR();
```

softAPenableIpV6

Function used to enable the IPv6 support.

```
bool softAPenableIpV6();
```

The function will return `true` if the configuration is successful.

softAPIPv6

Function to get the IPv6 address.

```
IPv6Address softAPIPv6();
```

The function will return the AP IPv6 address in `IPv6Address` format.

softAPgetHostname

Function to get the AP hostname.

```
const char * softAPgetHostname();
```

softAPsetHostname

Function to set the AP hostname.

```
bool softAPsetHostname(const char * hostname);
```

Where:

- `hostname` sets the device hostname.

The function will return `true` if the configuration is successful.

softAPmacAddress

Function to define the AP MAC address.

```
uint8_t* softAPmacAddress(uint8_t* mac);
```

Where:

- `mac` sets the new MAC address.

Function to get the AP MAC address.

```
String softAPmacAddress(void);
```

softAPSSID

Function to get the AP SSID.

```
String softAPSSID(void) const;
```

Returns the AP SSID.

WiFiSTA

The `WiFiSTA` is used to configure and manage the Wi-Fi as Station. The related functions for the STA are here.

Basic Usage

The following code shows the basic usage of the WifiSTA functionality.

```
WiFi.begin(ssid, password);
```

Where the `ssid` and `password` are from the network you want to connect the ESP32.

To check if the connection is successful, you can use:

```
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
}
```

After a successful connection, you can print the IP address given by the network.

```
Serial.println("IP address: ");
Serial.println(WiFi.localIP());
```

Please see the full example of the WiFiSTA in: *sta example*.

STA Configuration

begin

- Functions `begin` are used to configure and start the Wi-Fi.

```
wl_status_t begin(const char* ssid, const char *passphrase = NULL, int32_t channel = 0,␣
↪const uint8_t* bssid = NULL, bool connect = true);
```

Where:

- `ssid` sets the AP SSID.
- `passphrase` sets the AP password. Set as `NULL` for open networks.
- `channel` sets the Wi-Fi channel.

- `uint8_t* bssid` sets the AP BSSID.
- `connect` sets `true` to connect to the configured network automatically.

```
wl_status_t begin(char* ssid, char *passphrase = NULL, int32_t channel = 0, const uint8_
↪t* bssid = NULL, bool connect = true);
```

Where:

- `ssid` sets the AP SSID.
- `passphrase` sets the AP password. Set as `NULL` for open networks.
- `channel` sets the Wi-Fi channel.
- `bssid` sets the AP BSSID.
- `connect` sets `true` to connect to the configured network automatically.

Function to start the connection after being configured.

```
wl_status_t begin();
```

config

Function `config` is used to configure Wi-Fi. After configuring, you can call function `begin` to start the Wi-Fi process.

```
bool config(IPAddress local_ip, IPAddress gateway, IPAddress subnet, IPAddress dns1 =␣
↪(uint32_t)0x00000000, IPAddress dns2 = (uint32_t)0x00000000);
```

Where:

- `local_ip` sets the local IP.
- `gateway` sets the gateway IP.
- `subnet` sets the subnet mask.
- `dns1` sets the DNS.
- `dns2` sets the DNS alternative option.

The function will return `true` if the configuration is successful.

The `IPAddress` format is defined by 4 bytes as described here:

```
IPAddress(uint8_t first_octet, uint8_t second_octet, uint8_t third_octet, uint8_t fourth_
↪octet);
```

Example:

```
IPAddress local_ip(192, 168, 10, 20);
```

See the `WiFiClientStaticIP.ino` for more details on how to use this feature.

STA Connection

reconnect

Function used to reconnect the Wi-Fi connection.

```
bool reconnect();
```

disconnect

Function to force disconnection.

```
bool disconnect(bool wifioff = false, bool eraseap = false);
```

Where:

- `wifioff` use `true` to turn the Wi-Fi radio off.
- `eraseap` use `true` to erase the AP configuration from the NVS memory.

The function will return `true` if the configuration is successful.

isConnected

Function used to get the connection state.

```
bool isConnected();
```

Return the connection state.

setAutoConnect

Function is deprecated.

getAutoConnect

Function is deprecated.

setAutoReconnect

Function used to set the automatic reconnection if the connection is lost.

```
bool setAutoReconnect(bool autoReconnect);
```

Where:

- `autoConnect` is set to `true` to enable this option.

getAutoReconnect

Function used to get the automatic reconnection if the connection is lost.

```
bool getAutoReconnect();
```

The function will return `true` if this setting is enabled.

setMinSecurity

Function used to set the minimum security for AP to be considered connectable.

```
bool setMinSecurity(wifi_auth_mode_t minSecurity);
```

Where:

- `minSecurity` is the minimum security for AP to be considered connectable. Default is `WIFI_AUTH_WPA2_PSK`.

WiFiMulti

The `WiFiMulti` allows you to add more than one option for the AP connection while running as a station.

To add the AP, use the following function. You can add multiple AP's and this library will handle the connection.

```
bool addAP(const char* ssid, const char *passphrase = NULL);
```

After adding the AP's, run by the following function.

```
uint8_t run(uint32_t connectTimeout=5000);
```

To see how to use the `WiFiMulti`, take a look at the `WiFiMulti.ino` example available.

WiFiScan

To perform the Wi-Fi scan for networks, you can use the following functions:

Start scan WiFi networks available.

```
int16_t scanNetworks(bool async = false, bool show_hidden = false, bool passive = false,␣
↪uint32_t max_ms_per_chan = 300, uint8_t channel = 0);
```

Called to get the scan state in Async mode.

```
int16_t scanComplete();
```

Delete last scan result from RAM.

```
void scanDelete();
```

Loads all infos from a scanned wifi in to the ptr parameters.

```
bool getNetworkInfo(uint8_t networkItem, String &ssid, uint8_t &encryptionType, int32_t &
↪RSSI, uint8_t* &BSSID, int32_t &channel);
```

To see how to use the `WiFiScan`, take a look at the `WiFiScan.ino` example available.

Examples

Wi-Fi AP Example

```
/*
  WiFiAccessPoint.ino creates a WiFi access point and provides a web server on it.

  Steps:
  1. Connect to the access point "yourAp"
  2. Point your web browser to http://192.168.4.1/H to turn the LED on or http://192.168.
↪4.1/L to turn it off
     OR
     Run raw TCP "GET /H" and "GET /L" on PuTTY terminal with 192.168.4.1 as IP address␣
↪and 80 as port

  Created for arduino-esp32 on 04 July, 2018
  by Elochukwu Ifediora (fedy0)
*/

#include <WiFi.h>
#include <WiFiClient.h>
#include <WiFiAP.h>

#define LED_BUILTIN 2   // Set the GPIO pin where you connected your test LED or comment␣
↪this line out if your dev board has a built-in LED

// Set these to your desired credentials.
const char *ssid = "yourAP";
const char *password = "yourPassword";

WiFiServer server(80);


void setup() {
  pinMode(LED_BUILTIN, OUTPUT);

  Serial.begin(115200);
  Serial.println();
  Serial.println("Configuring access point...");

  // You can remove the password parameter if you want the AP to be open.
  WiFi.softAP(ssid, password);
  IPAddress myIP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(myIP);
  server.begin();

  Serial.println("Server started");
}

void loop() {
  WiFiClient client = server.available();   // listen for incoming clients
```

(continues on next page)

(continued from previous page)

```
  if (client) {                             // if you get a client,
    Serial.println("New Client.");           // print a message out the serial port
    String currentLine = "";                // make a String to hold incoming data from␣
↪the client
    while (client.connected()) {            // loop while the client's connected
      if (client.available()) {             // if there's bytes to read from the client,
        char c = client.read();             // read a byte, then
        Serial.write(c);                    // print it out the serial monitor
        if (c == '\n') {                    // if the byte is a newline character

          // if the current line is blank, you got two newline characters in a row.
          // that's the end of the client HTTP request, so send a response:
          if (currentLine.length() == 0) {
            // HTTP headers always start with a response code (e.g. HTTP/1.1 200 OK)
            // and a content-type so the client knows what's coming, then a blank line:
            client.println("HTTP/1.1 200 OK");
            client.println("Content-type:text/html");
            client.println();

            // the content of the HTTP response follows the header:
            client.print("Click <a href=\"/H\">here</a> to turn ON the LED.<br>");
            client.print("Click <a href=\"/L\">here</a> to turn OFF the LED.<br>");

            // The HTTP response ends with another blank line:
            client.println();
            // break out of the while loop:
            break;
          } else {    // if you got a newline, then clear currentLine:
            currentLine = "";
          }
        } else if (c != '\r') {  // if you got anything else but a carriage return␣
↪character,
          currentLine += c;      // add it to the end of the currentLine
        }

        // Check to see if the client request was "GET /H" or "GET /L":
        if (currentLine.endsWith("GET /H")) {
          digitalWrite(LED_BUILTIN, HIGH);               // GET /H turns the LED on
        }
        if (currentLine.endsWith("GET /L")) {
          digitalWrite(LED_BUILTIN, LOW);                // GET /L turns the LED off
        }
      }
    }
    // close the connection:
    client.stop();
    Serial.println("Client Disconnected.");
  }
}
```

Wi-Fi STA Example

```
/*
    Go to thingspeak.com and create an account if you don't have one already.
    After logging in, click on the "New Channel" button to create a new channel for your␣
↪data. This is where your data will be stored and displayed.
    Fill in the Name, Description, and other fields for your channel as desired, then␣
↪click the "Save Channel" button.
    Take note of the "Write API Key" located in the "API keys" tab, this is the key you␣
↪will use to send data to your channel.
    Replace the channelID from tab "Channel Settings" and privateKey with "Read API Keys
↪" from "API Keys" tab.
    Replace the host variable with the thingspeak server hostname "api.thingspeak.com"
    Upload the sketch to your ESP32 board and make sure that the board is connected to␣
↪the internet. The ESP32 should now send data to your Thingspeak channel at the␣
↪intervals specified by the loop function.
    Go to the channel view page on thingspeak and check the "Field1" for the new␣
↪incoming data.
    You can use the data visualization and analysis tools provided by Thingspeak to␣
↪display and process your data in various ways.
    Please note, that Thingspeak accepts only integer values.

    You can later check the values at https://thingspeak.com/channels/2005329
    Please note that this public channel can be accessed by anyone and it is possible␣
↪that more people will write their values.
 */

#include <WiFi.h>

const char* ssid     = "your-ssid"; // Change this to your WiFi SSID
const char* password = "your-password"; // Change this to your WiFi password

const char* host = "api.thingspeak.com"; // This should not be changed
const int httpPort = 80; // This should not be changed
const String channelID   = "2005329"; // Change this to your channel ID
const String writeApiKey = "V6YOTILH9I7D51F9"; // Change this to your Write API key
const String readApiKey = "34W6LGLIFXD56MPM"; // Change this to your Read API key

// The default example accepts one data filed named "field1"
// For your own server you can ofcourse create more of them.
int field1 = 0;

int numberOfResults = 3; // Number of results to be read
int fieldNumber = 1; // Field number which will be read out

void setup()
{
    Serial.begin(115200);
    while(!Serial){delay(100);}

    // We start by connecting to a WiFi network
```

(continues on next page)

(continued from previous page)

```
    Serial.println();
    Serial.println("*****************************************************");
    Serial.print("Connecting to ");
    Serial.println(ssid);

    WiFi.begin(ssid, password);

    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }

    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
}

void readResponse(WiFiClient *client){
  unsigned long timeout = millis();
  while(client->available() == 0){
    if(millis() - timeout > 5000){
      Serial.println(">>> Client Timeout !");
      client->stop();
      return;
    }
  }

  // Read all the lines of the reply from server and print them to Serial
  while(client->available()) {
    String line = client->readStringUntil('\r');
    Serial.print(line);
  }

  Serial.printf("\nClosing connection\n\n");
}

void loop(){
  WiFiClient client;
  String footer = String(" HTTP/1.1\r\n") + "Host: " + String(host) + "\r\n" +
↪"Connection: close\r\n\r\n";

  // WRITE ---------------------------------------------------------------------------------
↪--------------
  if (!client.connect(host, httpPort)) {
    return;
  }

  client.print("GET /update?api_key=" + writeApiKey + "&field1=" + field1 + footer);
  readResponse(&client);

  // READ ----------------------------------------------------------------------------------
↪-------------
```

(continues on next page)

(continued from previous page)

```
  String readRequest = "GET /channels/" + channelID + "/fields/" + fieldNumber + ".json?
↪results=" + numberOfResults + " HTTP/1.1\r\n" +
                       "Host: " + host + "\r\n" +
                       "Connection: close\r\n\r\n";

  if (!client.connect(host, httpPort)) {
    return;
  }

  client.print(readRequest);
  readResponse(&client);

  // ------------------------------------------------------------------------------------
↪-------------

  ++field1;
  delay(10000);
}
```

References

CHAPTER

THREE

GUIDES

3.1 Compile Arduino libs with ESP_LOGx

There are 2 primary approaches and both of them involve editing file `configs/defconfig.common`. Edit the file directly and then build. Later you can `git restore configs/defconfig.common` to go back. Copy the file `cp configs/defconfig.common configs/defconfig.debug` and edit the debug version.

`vim configs/defconfig.common` or `vim configs/defconfig.debug`

Edit **line 44** containing by default `CONFIG_LOG_DEFAULT_LEVEL_ERROR=y` to one of the following lines depending on your desired log level:

```
CONFIG_LOG_DEFAULT_LEVEL_NONE=y # No output
CONFIG_LOG_DEFAULT_LEVEL_ERROR=y # Errors - default
CONFIG_LOG_DEFAULT_LEVEL_WARN=y # Warnings
CONFIG_LOG_DEFAULT_LEVEL_INFO=y # Info
CONFIG_LOG_DEFAULT_LEVEL_DEBUG=y # Debug
CONFIG_LOG_DEFAULT_LEVEL_VERBOSE=y # Verbose
```

Then simply build the libs for all SoCs or one specific SoC. Note that building for all SoCs takes a lot of time, so if you are working only with specific SoC(s), build only for those.

Note: If you have copied the `defconfig` file and the debug settings are in file `configs/defconfig.debug` add flag `debug` to compilation command. Example : `./build.sh debug`

- **Option 1**: Build for all SoCs: `./build.sh`
- **Option 2**: Build for one SoC: `./build.sh -t <soc>`. The exact text to choose the SoC:
  - `esp32`
  - `esp32s2`
  - `esp32c3`
  - `esp32s3`
  - Example: `./build.sh -t esp32`
  - A wrong format or non-existing SoC will result in the error sed: can't read sdkconfig: No such file or directory

---

3.2 Documentation Contribution Guidelines

3.2.1 Introduction

This is a guideline for the Arduino ESP32 project documentation. The idea for this guideline is to show how to start collaborating on the project.

The guideline works to give you the directions and to keep the documentation more concise, helping users to better understand the structure.

3.2.2 About Documentation

We all know how important documentation is. This project is no different.

This documentation was created in a collaborative and open way, letting everyone contribute, from a small typo fix to a new chapter writing. We try to motivate our community by giving all the support needed through this guide.

The documentation is in **English only**. Future translations can be added when we finish the essential content in English first.

3.2.3 How to Collaborate

Everyone with some knowledge to share is welcome to collaborate.

One thing you need to consider is the fact that your contribution must be concise and assertive since it will be used by people developing projects. The information is very important for everyone, be sure you are not making the developer's life harder!

3.2.4 Documentation Guide

This documentation is based on the Sphinx with reStructuredText and hosted by ReadTheDocs.

If you want to get started with Sphinx, see the official documentation:

- Documentation Index
- Basics
- Directives

First Steps

Before starting your collaboration, you need to get the documentation source code from the Arduino-ESP32 project.

- **Step 1** - Fork the Arduino-ESP32 to your GitHub account.
- **Step 2** - Check out the recently created fork.
- **Step 3** - Create a new branch for the changes/addition to the docs.
- **Step 4** - Write!

Requirements

To properly work with the documentation, you need to install some packages in your system.

```
pip install -U Sphinx
pip install -r requirements.txt
```

The requirements file is under the `docs` folder.

Using Visual Studio Code

If you are using the Visual Studio Code, you can install some extensions to help you while writing documentation.

reStructuredText Pack

We also recommend you install to grammar check extension to help you to review English grammar.

Grammarly

Building

To build the documentation and generate the HTLM files, you can use the following command inside the `docs` folder. After a successful build, you can check the files inside the *build/html* folder.

```
make html
```

This step is essential to ensure that there are no syntax errors and also to see the final result.

If everything is ok, you will see some output logs similar to this one:

```
Running Sphinx v2.3.1
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 35 source files that are out of date
updating environment: [extensions changed ('sphinx_tabs.tabs')] 41 added, 3 changed, 0␣
↪removed
reading sources... [100%] tutorials/tutorials
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] tutorials/tutorials
generating indices...  genindexdone
writing additional pages...  searchdone
copying images... [100%] tutorials/../_static/tutorials/peripherals/tutorial_peripheral_
↪diagram.png
copying static files... ... done
copying extra files... done
dumping search index in English (code: en)... done
dumping object inventory... done
build succeeded.
```

The HTML pages are in build/html.

3.2.5 Sections

The Arduino ESP32 is structured in some sections to make it easier to maintain. Here is a brief description of this structure.

API

In this section, you will include all the documentation about drivers, libraries, and any other related to the core.

In this section, we do not add general information. For more general information, we have sections for other related parts, like the FAQ, library builder, troubleshooting, etc.

Boards

Here is the place to add any special guide on the development boards, pin layout, schematics, and any other relevant content.

Common

In this folder, you can add all common information used in several different places. This helps to make documentation easily maintainable.

Guides

This is the place to add the guides for common applications, IDEs configuration, and any other information that can be used as a guideline.

Tutorials

If you want to add a specific tutorial related to the Arduino core for ESP32, this is the place. The intention is not to create a blog or a demo area, but this can be used to add some complex description or to add some more information about APIs.

Images and Assets

All the files used on the documentation must be stored in the `_static` folder. Be sure that the content used is not with any copyright restriction.

3.2.6 Documentation Rules

Here are some guidelines to help you. We also recommend copying a sample file from the same category you are creating.

This will help you to follow the structure as well as to get inspired.

Basic Structure

To help you create a new section from scratch, we recommend you include this structure in your content if it applies.

- **About - Brief description of the document.**
  - Description of the peripheral, driver, protocol, including all different modes and configurations.
- **API** - Description of each public function, macros, and structs.
- **Basic Usage**
- **Example Application**

About Section

In this section, you need to add a brief description of the API. If you are describing a peripheral API, you should explain a little bit about the peripheral and the working modes, if it's applicable.

API Functions

To add a new function description, you must know that the users only have access to the public functions.

Here is an example of how to add the function description from I2C API:

```
setPins
^^^^^^^

This function is used to define the ``SDA`` and ``SCL`` pins.

.. note:: Call this function before ``begin`` to change the pins from the default ones.

.. code-block:: arduino

    bool setPins(int sdaPin, int sclPin);

* ``sdaPin`` sets the GPIO to be used as the I2C peripheral data line.

* ``sclPin`` sets the GPIO to be used as the I2C peripheral clock line.

The default pins may vary from board to board. On the *Generic ESP32* the default I2C␣
↪pins are:

* ``sdaPin`` **GPIO21**

* ``sclPin`` **GPIO22**

This function will return ``true`` if the peripheral was configured correctly.
```

Be sure to include a very comprehensive description, add all the parameters in and out, and describe the desired output.

If the function uses a specific structure, you can also describe the structure in the same function block or add a specific section if the structure is shared with other functions.

Basic Usage

Some APIs are more complex to use or require more steps in order to configure or initialize. If the API is not straightforward in terms of usability, please consider adding a how-to-use section describing all the steps to get the API configured.

Here is an example:

```
Basic Usage
^^^^^^^^^^^

To start using I2C as slave mode on the Arduino, the first step is to include the ``Wire.
↪h`` header to the sketch.

.. code-block:: arduino

    #include "Wire.h"

Before calling ``begin``, you must create two callback functions to handle the␣
↪communication with the master device.

.. code-block:: arduino

    Wire.onReceive(onReceive);

and

.. code-block:: arduino

    Wire.onRequest(onRequest);

The ``onReceive`` will handle the request from the ``master`` device upon a slave read␣
↪request and the ``onRequest`` will handle the answer to the master.

Now, we can start the peripheral configuration by calling ``begin`` function with the␣
↪device address.

.. code-block:: arduino

    Wire.begin((uint8_t)I2C_DEV_ADDR);

By using ``begin`` without any arguments, all the settings will be done by using the␣
↪default values. To set the values on your own, see the function description. This␣
↪function is described here: `i2c begin`_
```

Example Application

It is very important to include at least one application example or a code snippet to help people using the API.

If the API does not have any application example, you can embed the code directly. However, if the example is available, you must include it as a literal block.

```
.. literalinclude:: ../../../libraries/WiFi/examples/WiFiAccessPoint/WiFiAccessPoint.ino
    :language: arduino
```

3.2.7 Sphinx Basics

Heading Levels

The heading levels used on this documentation are:

- **H1**: - (Dash)
- **H2**: * (Asterisk)
- **H3**: ^ (Circumflex)
- **H4**: # (Sharp)

Code Block

To add a code block, you can use the following structure:

```
.. code-block:: arduino
    bool begin(); //Code example
```

Links

To include links to external content, you can use two ways.

- First option:

```
`Arduino Wire Library`_

_Arduino Wire Library: https://www.arduino.cc/en/reference/wire
```

- Second option:

```
`Arduino Wire Library <https://www.arduino.cc/en/reference/wire>`_
```

Images

To include images in the docs, first, add all the files into the `_static` folder with a filename that makes sense for the topic.

After that, you can use the following structure to include the image in the docs.

```
.. figure:: ../_static/arduino_i2c_master.png
    :align: center
    :width: 720
    :figclass: align-center
```

You can adjust the `width` according to the image size.

Be sure the file size does not exceed 600kB.

Support

If you need support on the documentation, you can ask a question in the discussion here.

3.2.8 Additional Guidelines

If you want to contribute with code on the Arduino ESP32 core, be sure to follow the ESP-IDF Documenting Code as a reference.

3.3 Arduino IDE Tools Menu

3.3.1 Introduction

This guide is a walkthrough of the Arduino IDE configuration menu for the ESP32 System on Chip (SoC's). In this guide, you will see the most relevant configuration to get your project optimized and working.

Since some boards and SoC's may vary in terms of hardware configuration, be sure you know all the board characteristics that you are using, like flash memory size, SoC variant (ESP32 family), PSRAM, etc.

Note: To help you identify the characteristics, you can see the Espressif Product Selector.

3.3.2 Arduino IDE

The Arduino IDE is widely used for ESP32 on Arduino development and offers a wide variety of configurations.

3.3.3 Tools Menu

To properly configure your project build and flash, some settings must be done in order to get it compiled and flashed without any issues. Some boards are natively supported and almost no configuration is required. However, if your is not yet supported or you have a custom board, you need to configure the environment by yourself.

For more details or to add a new board, see the boards.txt file.

3.3.4 Generic Options

Most of the options are available for every ESP32 family. Some options will be available only for specific targets, like the USB configuration.

Board

This option is the target board and must be selected in order to get all the default configuration settings. Once you select the correct board, you will see that some configurations will be automatically selected, but be aware that some boards can have multiple versions (i.e different flash sizes).

To select the board, go to `Tools -> Board -> ESP32 Arduino` and select the target board.

If your board is not present on this list, you can select the generic `ESP32-XX Dev Module`.

Currently, we have one generic development module for each of the supported targets.

If the board selected belongs to another SoC family, you will see the following information at the build output:

```
A fatal error occurred:  This chip is ESP32 not ESP32-S2. Wrong --chip
argument?
```

Upload Speed

To select the flashing speed, change the `Tools -> Upload Speed`. This value will be used for flashing the code to the device.

Note: If you have issues while flashing the device at high speed, try to decrease this value. This could be due to the external serial-to-USB chip limitations.

CPU Frequency

On this option, you can select the CPU clock frequency. This option is critical and must be selected according to the high-frequency crystal present on the board and the radio usage (Wi-Fi and Bluetooth).

In some applications, reducing the CPU clock frequency is recommended in order to reduce power consumption.

If you don't know why you should change this frequency, leave the default option.

Flash Frequency

Use this function to select the flash memory frequency. The frequency will be dependent on the memory model.

- **40MHz**
- **80MHz**

If you don't know if your memory supports **80Mhz**, you can try to upload the sketch using the **80MHz** option and watch the log output via the serial monitor.

Note: In some boards/SoC, the flash frequency is automatically selected according to the flash mode. In some cases (i.e ESP32-S3), the flash frequency is up to 120MHz.

Flash Mode

This option is used to select the SPI communication mode with the flash memory.

Depending on the application, this mode can be changed in order to increase the flash communication speed.

- **QIO - Quad I/O Fast Read**
  - Four SPI pins are used to write to the flash and to read from the flash.
- **DIO - Dual I/O Fast Read**
  - Two SPI pins are used to write to the flash and to read from the flash.
- **QOUT - Quad Output Fast Read**
  - Four SPI pins are used to read the flash data.
- **DOUT - Dual Output Fast Read**
  - Two SPI pins are used to read flash data.
- **OPI - Octal I/O**
  - Eight SPI pins are used to write and to read from the flash.

If you don't know how the board flash is physically connected or the flash memory model, try the **QIO** at **80MHz** first.

Flash Size

This option is used to select the flash size. The flash size should be selected according to the flash model used on your board.

- **2MB** (16Mb)
- **4MB** (32Mb)
- **8MB** (64Mb)
- **16MB** (128Mb)

If you choose the wrong size, you may have issues when selecting the partition scheme.

Embedded Flash

Some SoC has embedded flash. The ESP32-S3 is a good example.

Note: Check the manufacturer part number of your SoC/module to see the right version.

Example: **ESP32-S3FH4R2**

This particular ESP32-S3 variant comes with 4MB Flash and 2MB PSRAM.

Options for Embedded Flash

- **Fx4** 4MB Flash (*QIO*)
- **Fx8** 8MB Flash (*QIO*)
- **V** 1.8V SPI

The **x** stands for the temperature range specification.

- **H** High Temperature (*-40 to 85ºC*)
- **N** Low Temeprature (*-40 to 65ºC*)

For more details, please see the corresponding datasheet at Espressif Product Selector.

Partition Scheme

This option is used to select the partition model according to the flash size and the resources needed, like storage area and OTA (Over The Air updates).

Note: Be careful selecting the right partition according to the flash size. If you select the wrong partition, the system will crash.

Core Debug Level

This option is used to select the Arduino core debugging level to be printed to the serial debug.

- **None** - Prints nothing.
- **Error** - Only at error level.
- **Warning** - Only at warning level and above.
- **Info** - Only at info level and above.
- **Debug** - Only at debug level and above.
- **Verbose** - Prints everything.

PSRAM

The PSRAM is an internal or external extended RAM present on some boards, modules or SoC.

This option can be used to `Enable` or `Disable` PSRAM. In some SoCs, you can select the PSRAM mode as the following.

- **QSPI PSRAM** - Quad PSRAM
- **OPI PSRAM** - Octal PSRAM

Embedded PSRAM

Some SoC has embedded PSRAM. The ESP32-S3 is a good example.

Example: **ESP32-S3FH4R2**

This particular ESP32-S3 comes with 4MB Flash and 2MB PSRAM.

Options for Embedded Flash and PSRAM

- **R2** 2MB PSRAM (*QSPI*)
- **R8** 8MB PSRAM (*OPI*)
- **V** 1.8V SPI

The **x** stands for the temperature range specification.

- **H** High Temperature (*-40 to 85ºC*)
- **N** Low Temeprature (*-40 to 65ºC*)

For more details, please see the corresponding datasheet at Espressif Product Selector.

Arduino Runs On

This function is used to select the core that runs the Arduino core. This is only valid if the target SoC has 2 cores.

When you have some heavy task running, you might want to run this task on a different core than the Arduino tasks. For this reason, you have this configuration to select the right core.

Events Run On

This function is also used to select the core that runs the Arduino events. This is only valid if the target SoC has 2 cores.

Erase All Flash Before Sketch Upload

This option selects the flash memory region to be erased before uploading the new sketch.

- **Disabled** - Upload the sketch without erasing all flash contents. (Default)
- **Enabled** - Erase all flash contents before uploading the sketch.

Port

This option is used to select the serial port to be used on the flashing and monitor.

3.3.5 USB Options

Some ESP32 families have a USB peripheral. This peripheral can be used for flashing and debugging.

To see the supported list for each SoC, see this section: Libraries.

The USB option will be available only if the correct target is selected.

USB CDC On Boot

The USB Communications Device Class, or USB CDC, is a class used for basic communication to be used as a regular serial controller (like RS-232).

This class is used for flashing the device without any other external device attached to the SoC.

This option can be used to `Enable` or `Disable` this function at the boot. If this option is `Enabled`, once the device is connected via USB, one new serial port will appear in the list of the serial ports. Use this new serial port for flashing the device.

This option can be used as well for debugging via the `Serial Monitor` using **CDC** instead of the **UART0**.

To use the UART as serial output, you can use `Serial0.print("Hello World!");` instead of `Serial.print("Hello World!");` which will be printed using USB CDC.

USB Firmware MSC On Boot

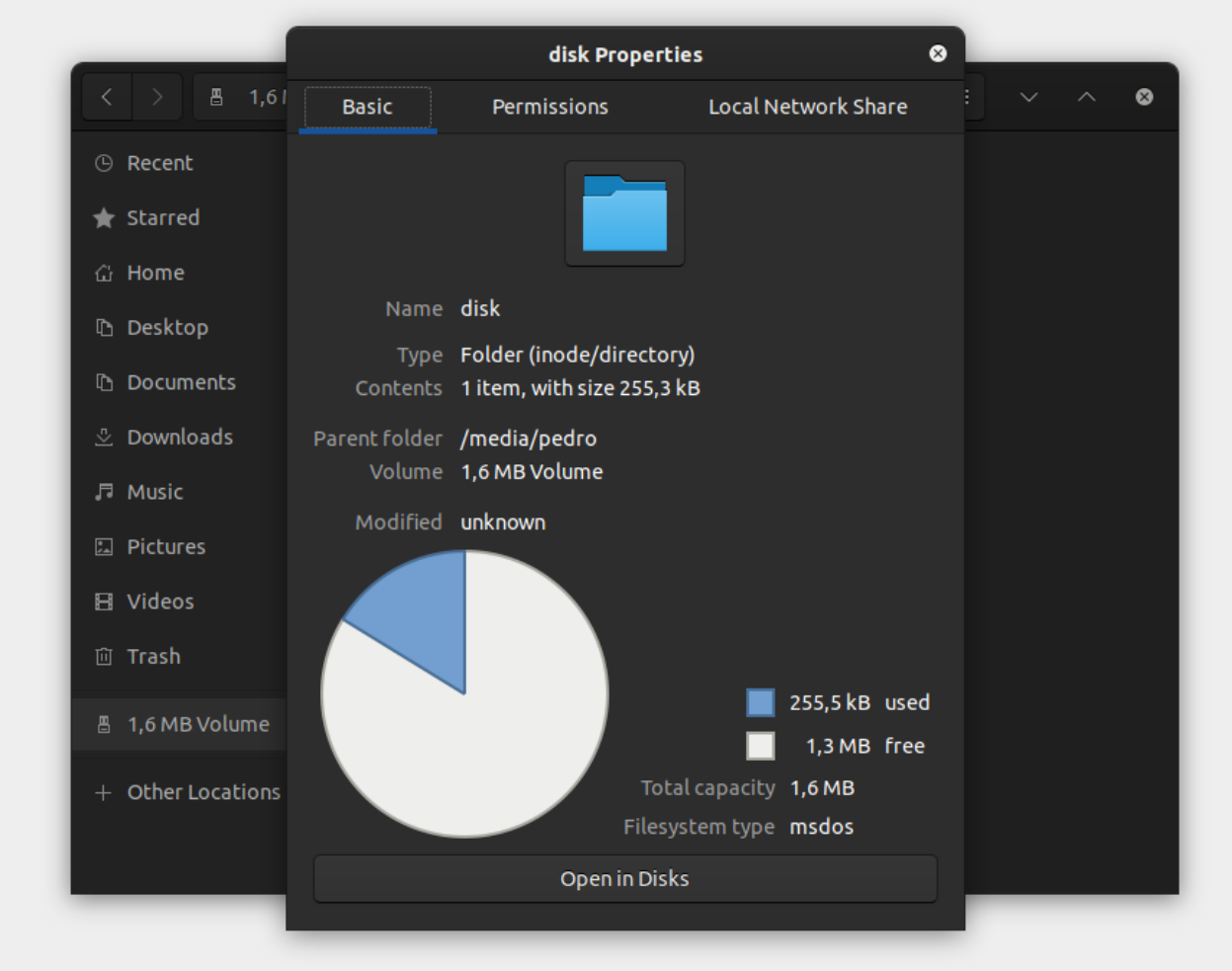
The USB Mass Storage Class, or USB MSC, is a class used for storage devices, like a USB flash drive.

This option can be used to `Enable` or `Disable` this function at the boot. If this option is `Enabled`, once the device is connected via USB, one new storage device will appear in the system as a storage drive. Use this new storage drive to write and read files or to drop a new firmware binary to flash the device.

USB DFU On Boot

The USB Device Firmware Upgrade is a class used for flashing the device through USB.

This option can be used to `Enable` or `Disable` this function at the boot. If this option is `Enabled`, once the device is connected via USB, the device will appear as a USB DFU capable device.


disk Properties
Basic
Permissions
Local Network Share
Recent
Starred
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
1,6 MB Volume
Other Locations
Name disk
Type Folder (inode/directory)
Contents 1 item, with size 255,3 kB
Parent folder /media/pedro
Volume 1,6 MB Volume
Modified unknown
255,5 kB used
1,3 MB free
Total capacity 1,6 MB
Filesystem type msdos
Open in Disks

CHAPTER

FOUR

TUTORIALS

4.1 Basic Tutorial

4.1.1 Introduction

This is the basic tutorial and should be used as template for other tutorials.

4.1.2 Requirements

- Arduino IDE
- ESP32 Board
- Good USB Cable

4.1.3 Steps

Here are the steps for this tutorial.

1. Open the Arduino IDE
2. Build and Flash the *blink* project.

4.1.4 Code

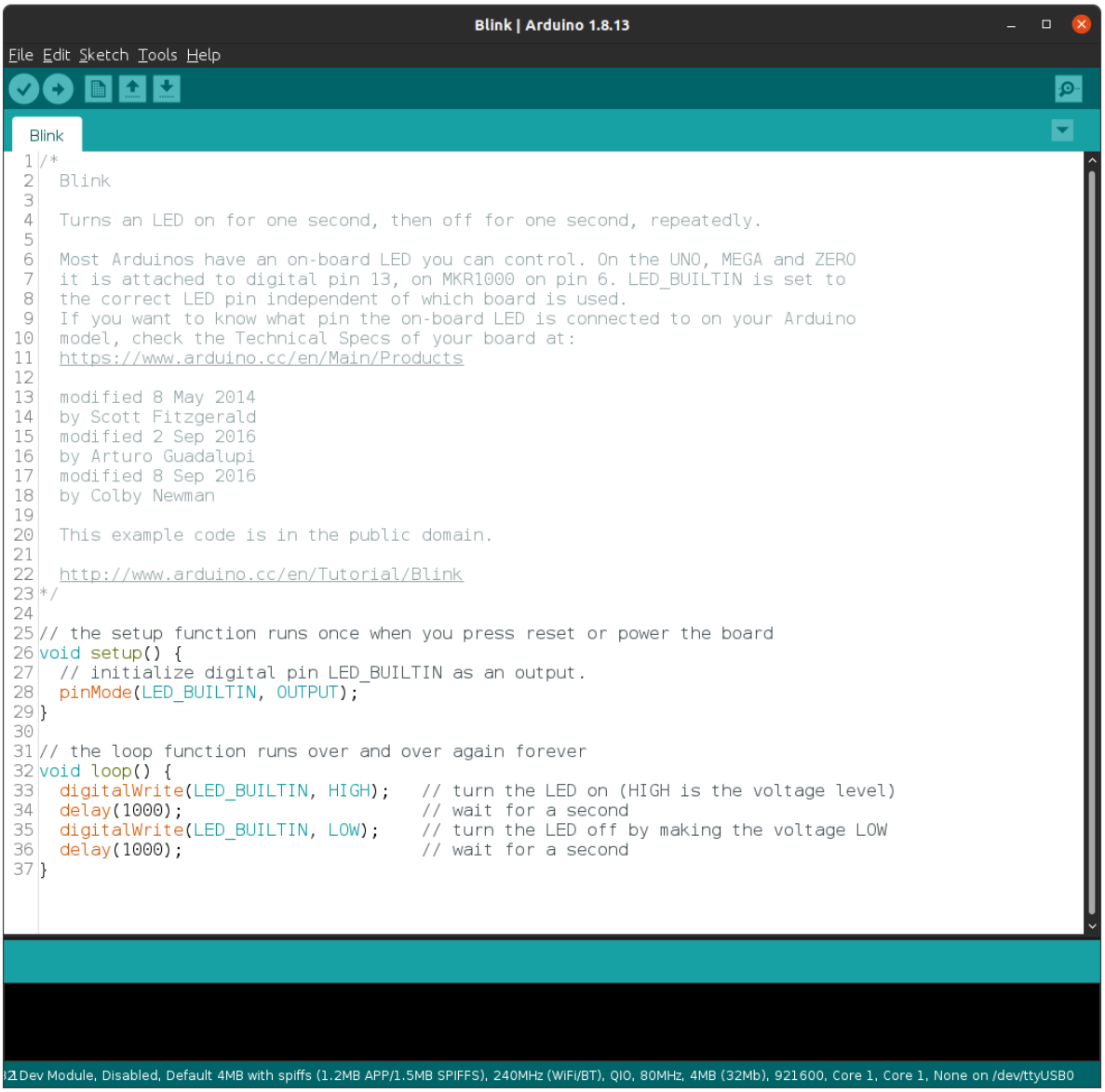
Listing 1: Blink.ino

```
/*
Blink

Turns an LED on for one second, then off for one second, repeatedly.

Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
the correct LED pin independent of which board is used.
If you want to know what pin the on-board LED is connected to on your Arduino
model, check the Technical Specs of your board at:
https://www.arduino.cc/en/Main/Products
```

(continues on next page)


Blink | Arduino 1.8.13
File Edit Sketch Tools Help
Blink
/*
Blink
Turns an LED on for one second, then off for one second, repeatedly.
Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
the correct LED pin independent of which board is used.
If you want to know what pin the on-board LED is connected to on your Arduino
model, check the Technical Specs of your board at:
https://www.arduino.cc/en/Main/Products
modified 8 May 2014
by Scott Fitzgerald
modified 2 Sep 2016
by Arturo Guadalupi
modified 8 Sep 2016
by Colby Newman
This example code is in the public domain.
http://www.arduino.cc/en/Tutorial/Blink
*/
// the setup function runs once when you press reset or power the board
void setup() {
// initialize digital pin LED_BUILTIN as an output.
pinMode(LED_BUILTIN, OUTPUT);
}
// the loop function runs over and over again forever
void loop() {
digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
delay(1000); // wait for a second
digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
delay(1000); // wait for a second
}
Dev Module, Disabled, Default 4MB with spiffs (1.2MB APP/1.5MB SPIFFS), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, Core 1, Core 1, None on /dev/ttyUSB0

(continued from previous page)

```
modified 8 May 2014
by Scott Fitzgerald
modified 2 Sep 2016
by Arturo Guadalupi
modified 8 Sep 2016
by Colby Newman

This example code is in the public domain.

http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset or power the board
void setup() {
// initialize digital pin LED_BUILTIN as an output.
pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
delay(1000);                       // wait for a second
digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
delay(1000);                       // wait for a second
}
```

4.1.5 Log Output

If the log output from the serial monitor is relevant, please add here:

```
I (0) cpu_start: App cpu up.
I (418) cpu_start: Pro cpu start user code
I (418) cpu_start: cpu freq: 160000000
```

4.1.6 Resources

- ESP32 Datasheet (Datasheet)

4.2 Blink Interactive Tutorial

4.2.1 Introduction

This is the interactive blink tutorial using Wokwi. For this tutorial, you don't need the ESP32 board or the Arduino toolchain.

Note: If you don't want to use this tutorial with the simulation, you can copy and paste the *Example Code* from Wokwi editor and use it on the Arduino IDE or PlatformIO.

4.2.2 About this Tutorial

This tutorial is the most basic for any get started. In this tutorial, we will show how to set a GPIO pin as an output to drive a LED to blink each 1 second.

4.2.3 Step by step

In order to make this simple blink tutorial, you'll need to do the following steps.

1. **Define the GPIO for the LED.**

```
#define LED 2
```

This `#define LED 2` will be used to set the GPIO2 as the `LED` output pin.

2. **Setup.**

Inside the `setup()` function, we need to add all things we want to run once during the startup. Here we'll add the `pinMode` function to set the pin as output.

```
void setup() {
    pinMode(LED, OUTPUT);
}
```

The first argument is the GPIO number, already defined and the second is the mode, here defined as an output.

3. **Main Loop.**

After the `setup`, the code runs the `loop` function infinitely. Here we will handle the GPIO in order to get the LED blinking.

```
void loop() {
    digitalWrite(LED, HIGH);
    delay(100);
    digitalWrite(LED, LOW);
    delay(100);
}
```

The first function is the `digitalWrite()` with two arguments:

- GPIO: Set the GPIO pin. Here defined by our `LED` connected to the GPIO2.
- State: Set the GPIO state as HIGH (ON) or LOW (OFF).

This first `digitalWrite` we will set the LED ON.

After the `digitalWrite`, we will set a `delay` function in order to wait for some time, defined in milliseconds.

Now we can set the GPIO to `LOW` to turn the LED off and `delay` for more few milliseconds to get the LED blinking.

4. **Run the code.**

To run this code, you'll need a development board and the Arduino toolchain installed on your computer. If you don't have both, you can use the simulator to test and edit the code.

4.2.4 Simulation

This simulator is provided by Wokwi and you can test the blink code and play with some modifications to learn more about this example.

Change the parameters, like the delay period, to test the code right on your browser. You can add more LEDs, change the GPIO, and more.

4.2.5 Example Code

Here is the full blink code.

```
#define LED 2

void setup() {
    pinMode(LED, OUTPUT);
}

void loop() {
    digitalWrite(LED, HIGH);
    delay(100);
    digitalWrite(LED, LOW);
    delay(100);
}
```

4.2.6 Resources

- ESP32 Datasheet (Datasheet)
- Wokwi (Wokwi Website)

4.3 USB CDC and DFU Flashing

4.3.1 Introduction

Since the ESP32-S2 introduction, Espressif has been working on USB peripheral support for some of the SoC families, including the ESP32-C3 and the ESP32-S3.

This new peripheral allows a lot of new possibilities, including flashing the firmware directly to the SoC without any external USB-to-Serial converter.

In this tutorial, you will be guided on how to use the embedded USB to flash the firmware.

The current list of supported SoCs:

SoC	USB Peripheral Support
ESP32-S2	CDC and DFU
ESP32-C3	CDC only
ESP32-S3	CDC and DFU

It's important that your board includes the USB connector attached to the embedded USB from the SoC. If your board doesn't have the USB connector, you can attach an external one to the USB pins.

These instructions it will only work on the supported devices with the embedded USB peripheral. This tutorial will not work if you are using an external USB-to-serial converter like FTDI, CP2102, CH340, etc.

For a complete reference to the Arduino IDE tools menu, please see the Tools Menus reference guide.

4.3.2 USB DFU

The USB DFU (Device Firmware Upgrade) is a class specification from the USB standard that adds the ability to upgrade the device firmware by the USB interface.

Flashing Using DFU

Note: DFU is only supported by the ESP32-S2 and ESP32-S3. See the table of supported SoCs.

To use the USB DFU to flash the device, you will need to configure some settings in the Arduino IDE according to the following steps:

1. Enter into Download Mode manually

This step is done only for the first time you flash the firmware in this mode. To enter into the download mode, you need to press and hold BOOT button and press and release the RESET button.

To check if this procedure was done correctly, now you will see the new USB device listed in the available ports. Select this new device in the **Port** option.

2. Configure the USB DFU

In the next step you can set the USB DFU as default on BOOT and for flashing.

Go to the Tools menu in the Arduino IDE and set the following options:

For ESP32-S2

- USB DFU On Boot -> Enable
- Upload Mode -> Internal USB

For ESP32-S3

- USB Mode -> USB-OTG (TinyUSB)
- USB DFU On Boot -> Enabled

Setp 3 - Flash

Now you can upload your sketch to the device. After flashing, you need to manually reset the device.

Note: On the USB DFU, you can't use the USB for the serial output for the logging, just for flashing. To enable the serial output, use the CDC option instead. If you want to use the USB DFU for just upgrading the firmware using the manual download mode, this will work just fine, however, for developing please consider using USB CDC.

4.3.3 USB CDC

The USB CDC (Communications Device Class) allows you to communicate to the device like in a serial interface. This mode can be used on the supported targets to flash and monitor the device in a similar way on devices that uses the external serial interfaces.

To use the USB CDC, you need to configure your device in the Tools menu:

1. Enter into Download Mode manually

Similar to the DFU mode, you will need to enter into download mode manually. To enter into the download mode, you need to press and hold BOOT button and press and release the RESET button.

To check if this procedure was done correctly, now you will see the new USB device listed in the available ports. Select this new device in the **Port** option.

2. Configure the USB CDC

For ESP32-S2

- USB CDC On Boot -> Enabled
- Upload Mode -> Internal USB

For ESP32-C3

- USB CDC On Boot -> Enabled

For ESP32-S3

- USB CDC On Boot -> Enabled
- Upload Mode -> UART0 / Hardware CDC

3. Flash and Monitor

You can now upload your sketch to the device. After flashing for the first time, you need to manually reset the device.

This procedure enables the flashing and monitoring thought the internal USB and does not requires you to manually enter into the download mode or to do the manual reset after flashing.

To monitor the device, you need to select the USB port and open the Monitor tool selecting the correct baud rate (usually 115200) according to the `Serial.begin()` defined in your code.

4.3.4 Hardware

If you are developing a custom hardware using the compatible SoC, and want to remove the external USB-to-Serial chip, this feature will complete substitute the needs of the external chip. See the SoC datasheet for more details about this peripheral.

4.4 GPIO Matrix and Pin Mux

4.4.1 Introduction

This is a basic introduction to how the peripherals work in the ESP32. This tutorial can be used to understand how to define the peripheral usage and its corresponding pins.

In some microcontrollers' architecture, the peripherals are attached to specific pins and cannot be redefined to another one.

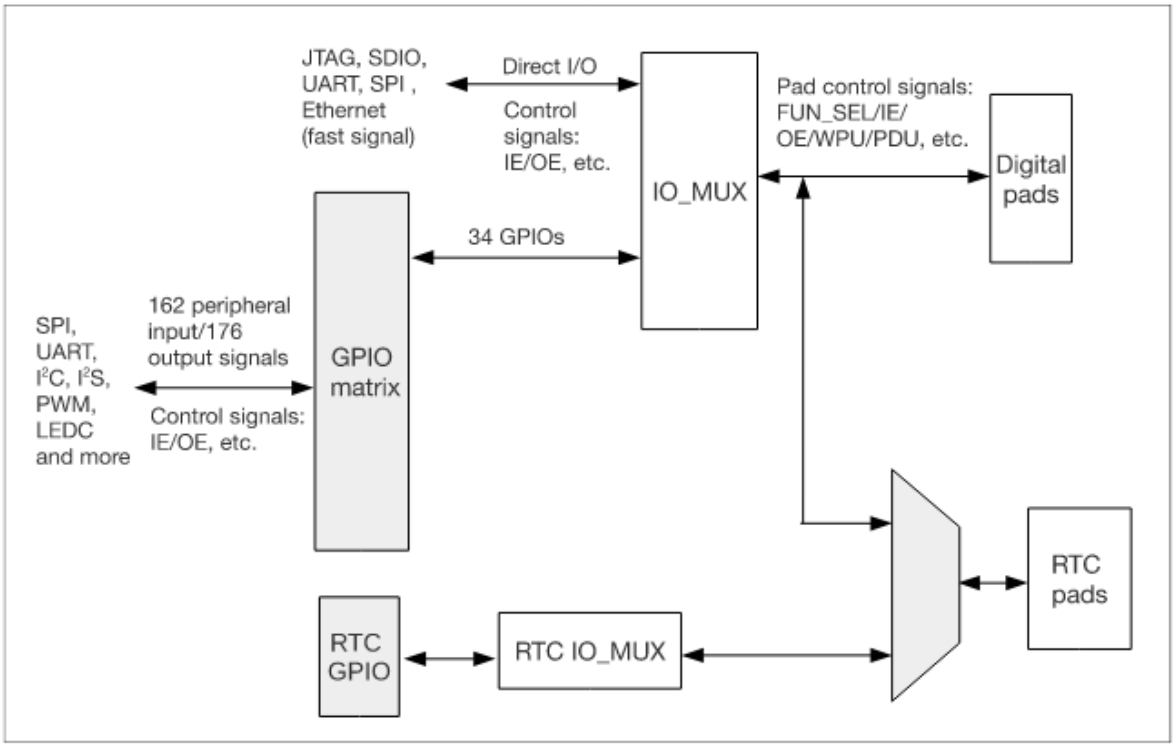
For example:

The *XYZ* MCU defines that the I2C peripheral SDA signal is the IO5 on the physical pin 10 and the SCL is on the IO6 and physical pin 11.

This means that, in your hardware project, you **NEED** to use these pins as the I2C and this cannot be changed due to the internal architecture. In this case, you must be very careful during the hardware design to not make any mistake by switching the SDA and SCL connections. Firmware will not help you if you do so.

4.4.2 GPIO Matrix and Pin Mux

The ESP32 architecture includes the capability of configuring some peripherals to any of the GPIOs pins, managed by the IO MUX GPIO. Essentially, this capability means that we can route the internal peripheral into a different physical pin using the IO MUX and the GPIO Matrix.



It means that in the scenario of the *XYZ* MCU, in the ESP32 we can use any of the GPIOs to route the SDA (input/output) and the SCL (output).

To use this functionality, we must be aware of some precautions:

- Some of the GPIOs are **INPUT** only.
- Some peripherals have output signals and must be used on GPIO's capable to be configured as **OUTPUT**.
- Some peripherals, mostly the high speed ones, ADC, DAC, Touch, and JTAG use dedicated GPIOs pins.

Warning: Before assigning the peripheral pins in your design, double check if the pins you're using are appropriate. The input-only pins cannot be used for peripherals that require output or input/output signals.

The greatest advantage of this functionality is the fact that we don't need to be fully dependent on the physical pin, since we can change according to our needs. This can facilitate the hardware design routing or in some cases, fix some

pin swap mistake during the hardware design phase.

4.4.3 Peripherals

Here is the basic peripherals list present on the ESP32. The peripheral list may vary from each ESP32 SoC family. To see all peripherals available on the ESP32-S2 and ESP32-C3, check each of the datasheets.

Peripheral Table

Type	Function
ADC	Dedicated GPIOs
DAC	Dedicated GPIOs
Touch Sensor	Dedicated GPIOs
JTAG	Dedicated GPIOs
SD/SDIO/MMC HostController	Dedicated GPIOs
Motor PWM	Any GPIO
SDIO/SPI SlaveController	Dedicated GPIOs
UART	Any GPIO[1]
I2C	Any GPIO
I2S	Any GPIO
LED PWM	Any GPIO
RMT	Any GPIO
GPIO	Any GPIO
Parallel QSPI	Dedicated GPIOs
EMAC	Dedicated GPIOs
Pulse Counter	Any GPIO
TWAI	Any GPIO
USB	Dedicated GPIOs

[1] except for the download/programming mode decided by the bootloader.

This table is present on each datasheet provided by Espressif.

4.4.4 Usage Examples

In the Arduino Uno, we have the I2C pins defined by hardware, A4 is the SDA and A5 the SCL. In this case, we do not need to set these pins in the `Wire.begin();` function, because they are already into the Wire library.

```
void setup()
{
    Wire.begin(); // join i2c bus (address optional for master)
}
```

Now, for the ESP32, the default pins for the I2C are SDA (GPIO21) and SCL (GPIO22). We can use a different pin as alternative for the default ones if you need to change the pins. To change the pins, we must call the `Wire.setPins(int sda, int scl);` function before calling `Wire.begin();`.

```
int sda_pin = 16; // GPIO16 as I2C SDA
int scl_pin = 17; // GPIO17 as I2C SCL
```

(continues on next page)

(continued from previous page)

```
void setup()
{
    Wire.setPins(sda_pin, scl_pin); // Set the I2C pins before begin
    Wire.begin(); // join i2c bus (address optional for master)
}
```

A similar approach also applies for the other peripherals.

4.4.5 Datasheet

- ESP32 (Datasheet)
- ESP32-S2 (Datasheet)
- ESP32-C3 (Datasheet)
- ESP32-S3 (Datasheet)

4.4.6 Resources

4.5 Partition Table

4.5.1 Introduction

Partition table is used to define the flash memory organization and the different kind of data will be stored on each partition.

You can use one of the available partition table scheme or create your own. You can see all the different schemes on the tools/partitions folder or by the Arduino IDE tools menu *Tools -> Partition Scheme*.

The partition table is created by a .CSV (Comma-separeted Values) file with the following structure:

```
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size, Flags
```

Where:

1. **Name**

   Is the partition name and must be a unique name. This name is not relevant for the system and the size must be at maximum of 16-chars (no special chars).

2. **Type**

   This is the type of the partition. This value can be `data` or `app`.

   - `app` type is used to define the partition that will store the application.
   - `data` type can be used to define the partition that stores general data, not the application.

3. **SubType**

   The SubType defines the usage of the `app` and `data` partitions.

   data

   `ota`

The ota subtype is used to store the OTA information. This partition is used only when the OTA is used to select the initialization partition, otherwise no need to add it to your custom partition table. The size of this partition should be a fixed size of 8kB (0x2000 bytes).

`nvs`

The nvs partition subtype is used to define the partition to store general data, like the WiFi data, device PHY calibration data and any other data to be stored on the non-volatile memory. This kind of partition is suitable for small custom configuration data, cloud certificates, etc. Another usage for the NVS is to store sensitive data, since the NVS supports encryption. It is highly recommended to add at least one nvs partition, labeled with the name nvs, in your custom partition tables with size of at least 12kB (0x3000 bytes). If needed, you can increase the size of the nvs partition. The recommended size for this partition is from 12kb to 64kb. Although larger NVS partitions can be defined, we recommend using FAT or SPIFFS filesystem for storage of larger amounts of data.

`coredump`

The coredump partition subtype is used to store the core dump on the flash. The core dump is used to analyze critical errors like crash and panic. This function must be enabled in the project configuration menu and set the data destination to flash. The recommended size for this partition is 64kB (0x10000).

`nvs_keys`

The nvs_keys partition subtype is used to store the keys when the NVS encryption is used. The size for this partition is 4kB (0x1000).

`fat`

The fat partition subtype defines the FAT filesystem usage, and it is suitable for larger data and if this data is often updated and changed. The FAT FS can be used with wear leveling feature to increase the erase/modification cycles per memory sector and encryption for sensitive data storage, like cloud certificates or any other data that may be protected. To use FAT FS with wear leveling see the example.

`spiffs`

The spiffs partition subtype defines the SPI flash filesystem usage, and it is also suitable for larger files and it also performs the wear leveling and file system consistency check. The SPIFFS do not support flash encryption.

app

`factory`

The factory partition subtype is the default application. The bootloader will set this partition as the default application initialization if no OTA partition is found, or the OTA partitions are empty. If the OTA partition is used, the ota_0 can be used as the default application and the factory can be removed from the partition table to save memory space.

`ota_0` to `ota_15`

The ota_x partition subtype is used for the Over-the air update. The OTA feature requires at least two ota_x partition (usually ota_0 and ota_1) and it also requires the ota partition to keep the OTA information data. Up to 16 OTA partitions can be defined but only two are needed for basic OTA feature.

`test`

The test partition subtype is used for factory test procedures.

4. **Offset**

The offset defines the partition start address. The offset is defined by the sum of the offset and the size of the earlier partition.

Note: Offset must be multiple of 4kB (0x1000) and for app partitions it must be aligned by 64kB (0x10000). If left blank, the offset will be automatically calculated based on the end of the previous partition, including any necessary alignment, however, the offset for the first partition must be always set as **0x9000** and for the first application partition **0x10000**.

---

5. **Size**

   Size defines the amount of memory to be allocated on the partition. The size can be formatted as decimal, hex numbers (0x prefix), or using unit prefix K (kilo) or M (mega) i.e: 4096 = 4K = 0x1000.

6. **Flags**

   The last column in the CSV file is the flags and it is currently used to define if the partition will be encrypted by the flash encryption feature.

For example, **the most common partition** is the `default_8MB.csv` (see tools/partitions folder for some examples):

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x5000,
otadata,  data, ota,     0xe000,  0x2000,
app0,     app,  ota_0,   0x10000, 0x330000,
app1,     app,  ota_1,   0x340000,0x330000,
spiffs,   data, spiffs,  0x670000,0x190000,
```

4.5.2 Using a Custom Partition Scheme

To create your own partition table, you can create the `partitions.csv` file **in the same folder you created your sketch**. The build system will automatically pick the partition table file and use it instead of the predefined ones.

Here is an example you can use for a custom partition table:

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     36K,     20K,
otadata,  data, ota,     56K,     8K,
app0,     app,  ota_0,   64K,     2M,
app1,     app,  ota_1,   ,        2M,
spiffs,   data, spiffs,  ,        8M,
```

This partition will use about 12MB of the 16MB flash. The offset will be automatically calculated after the first application partition and the units are in K and M.

A alternative is to create the new partition table as a new file in the tools/partitions folder and edit the boards.txt file to add your custom partition table.

4.5.3 Examples

2MB no OTA

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     36K,     20K,
factory,  app,  factory, 64K,     1900K,
```

4MB no OTA

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     36K,     20K,
factory,  app,  factory, 64K,     4000K,
```

4MB with OTA

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     36K,     20K,
otadata,  data, ota,     56K,     8K,
app0,     app,  ota_0,   64K,     1900K,
app1,     app,  ota_1,   ,        1900K,
```

8MB no OTA with Storage

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     36K,     20K,
factory,  app,  factory, 64K,     2M,
spiffs,   data, spiffs,  ,        5M,
```

8MB with OTA and Storage

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     36K,     20K,
otadata,  data, ota,     56K,     8K,
app0,     app,  ota_0,   64K,     2M,
app1,     app,  ota_1,   ,        2M,
spiffs,   data, spiffs,  ,        3M,
```

4.5.4 Reference

This documentation was based on the How to use custom partition tables on ESP32 article.

4.6 Preferences

4.6.1 Introduction

The Preferences library is unique to arduino-esp32. It should be considered as the replacement for the Arduino EEPROM library.

It uses a portion of the on-board non-volatile memory (NVS) of the ESP32 to store data. This data is retained across restarts and loss of power events to the system.

Preferences works best for storing many small values, rather than a few large values. If you need to store large amounts of data, consider using a file system library such as LitteFS.

The Preferences library is usable by all ESP32 variants.

4.6.2 Preferences Attributes

Preferences data is stored in NVS in sections called a "`namespace`". Within each namespace are a set of `key-value` pairs. The "`key`" is the name of the data item and the "`value`" is, well, the value of that piece of data. Kind of like variables. The key is the name of the variable and the value is its value. Like variables, a `key-value` pair has a data type.

Multiple namespaces are permitted within NVS. The name of each namespace must be unique. The keys within that namespace are unique to that namespace. Meaning the same key name can be used in multiple namespaces without conflict.

Namespace and key names are case sensitive.

Each key name must be unique within a namespace.

Namespace and key names are character strings and are limited to a maximum of 15 characters.

Only one namespace can be open (in use) at a time.

4.6.3 Library Overview

Library methods are provided to:

- create a namespace;
- open and close a namespace;
- store and retrieve data within a namespace for supported data types;
- determine if a key value has been initialized;
- delete a `key-value` pair;
- delete all `key-value` pairs in a namespace;
- determine data types stored against a key;
- determine the number of key entries available in the namespace.

Preferences directly suports the following data types:

Table 1: **Table 1 — Preferences Types**

Preferences Type	Data Type	Size (bytes)
Bool	bool	1
Char	int8_t	1
UChar	uint8_t	1
Short	int16_t	2
UShort	uint16_t	2
Int	int32_t	4
UInt	uint32_t	4
Long	int32_t	4
ULong	uint32_t	4
Long64	int64_t	8
ULong64	uint64_t	8
Float	float_t	8
Double	double_t	8
String	const char*	variable
	String	
Bytes	uint8_t	variable

String values can be stored and retrieved either as an Arduino String or as a null terminated `char` array (C-string).

Bytes type is used for storing and retrieving an arbitrary number of bytes in a namespace.

4.6.4 Workflow

Preferences workflow, once everything is initialized, is pretty simple.

To store a value:

- Open the namespace in read-write mode.
- Put the value into the key.
- Close the namespace.

To retrieve a value:

- Open the namespace in read-only mode.
- Use the key to get the value.
- Close the namespace.

(Technically, you can retrieve a value if the namespace is open in either read-only or read-write mode but it's good practice to open the namespace in read-only mode if you are only retrieving values.)

When storing information, a "`put[PreferencesType]`" method referenced to its key is used.

When retrieving information a "`get[PreferencesType]`" method referenced to its key is used.

Ensuring that the data types of your "`get`'s" and "`put`'s" all match, you're good to go.

The nuance is in initializing everything at the start.

Before you can store or retrieve anything using Preferences, both the namespace and the key within that namespace need to exist. So the workflow is:

1. Create or open the namespace.
2. Test for the existence of a key that should exist if the namespace has been initialized.

3. If that key does not exist, create the key(s).
4. Carry on with the rest of your sketch where data can now be stored and retrieved from the namespace.

Each step is discussed below.

Note: From here on when referring in general to a method used to store or retrieve data we'll use the shorthand "`putX`" and "`getX`" where the "`X`" is understood to be a Preferences Type; Bool, UInt, Char, and so on from the Preferences Types table above.

Create or Open the Namespace

In your sketch, first insert a declaration of a `Preferences` object by including a line like;

```
Preferences mySketchPrefs;    // "mySketchPrefs" is the name of the Preferences object.
                              //  Can be whatever you want.
```

This object is used with the Preferences methods to access the namespace and the key-value pairs it contains.

A namespace is made available for use with the `.begin` method:

```
mySketchPrefs.begin("myPrefs", false)
```

If the namespace does not yet exist, this will create and then open the namespace `myPrefs`.

If the namespace already exists, this will open the namespace `myPrefs`.

If the second argument is `false` the namespace is opened in read-write (RW) mode — values can be stored in to and retrieved from the namespace. If it is `true` the namespace is opened in read-only (RO) mode — values can be retrieved from the namespace but nothing can be stored.

Test for Initial Existence of Your Key(s)

When the ESP32 boots, there is no inherent way to know if this is the very first time it has ever powered on or if it is a subsequent launch and it has run its sketch before. We can use Preferences to store information that is retained across reboots that we can read, and based on that, decide if this is a first-time run and take the required actions if so.

We do this by testing for the existence of a certain key within a namespace. If that key exists, it is safe to assume the key was created during the first-time run of the sketch and so the namespace has already been initialized.

To determine if a key exists, use:

```
isKey("myTestKey")
```

This returns `true` if "`myTestKey`" exists in the namespace, and `false` if it does not.

By example, consider this code segment:

```
Preferences mySketchPrefs;
String doesExist;

mySketchPrefs.begin("myPrefs", false);   // open (or create and then open if it does not
                                         //  yet exist) the namespace "myPrefs" in RW␣
↪mode.
```

(continues on next page)

(continued from previous page)

```
bool doesExist = mySketchPrefs.isKey("myTestKey");

if (doesExist == false) {
    /*
       If doesExist is false, we will need to create our
        namespace key(s) and store a value into them.
   */

   // Insert your "first time run" code to create your keys & assign their values below␣
↪here.
}
else {
   /*
       If doesExist is true, the key(s) we need have been created before
        and so we can access their values as needed during startup.
   */

   // Insert your "we've been here before" startup code below here.
}
```

Creating Namespace Keys and Storing Values

To create a key, we use one of the `.putX` methods, matching `"X"` to the Preferences Type of the data we wish to store:

```
myPreferences.putX("myKeyName", value)
```

If `"myKeyName"` does not exist in the namespace, it is first created and then `value` is stored against that keyname. The namespace must be open in RW mode to do this. Note that `value` is not optional and must be provided with every "`.putX`" statement. Thus every key within a namespace will always hold a valid value.

An example is:

```
myPreferences.putFloat("pi", 3.14159265359);    // stores an float_t data type
                                                //  against the key "pi".
```

Reading Values From a Namespace

Once a key exists in a namespace and the namespace is open, its value is retrieved using one of the `getX` methods, matching `"X"` to the type of data stored against that key.

```
myPreferences.getX("myKeyName")
```

Like so:

```
String myString = myPreferences.getString("myStringKey");
```

This will retrieve the String value from the namespace key `"myStringKey"` and assign it to the String type variable `myString`.

Summary

So the basics of using Preferences are:

1. You cannot store into or retrieve from a `key-value` pair until a namespace is created and opened and the key exists in that namespace.
2. If the key already exists, it was created the first time the sketch was run.
3. A key value can be retrieved regardless of the mode in which the namespace was opened, but a value can only be stored if the namespace is open in read-write mode.
4. Data types of the "`get`'s" and "`put`'s" must match.
5. Remember the 15 character limit for namespace and key names.

4.6.5 Real World Example

Here is part of a `setup()` function that uses Preferences.

Its purpose is to set either a factory default configuration if the system has never run before, or use the last configuration if it has.

When started, the system has no way of knowing which of the above conditions is true. So the first thing it does after opening the namespace is check for the existence of a key that we have predetermined can only exist if we have previously run the sketch. Based on its existence we decide if a factory default set of operating parameters should be used (and in so doing create the namespace keys and populate the values with defaults) or if we should use operating parameters from the last time the system was running.

```
#include <Preferences.h>

#define RW_MODE false
#define RO_MODE true

Preferences stcPrefs;

void setup() {

   // not the complete setup(), but in setup(), include this...

   stcPrefs.begin("STCPrefs", RO_MODE);           // Open our namespace (or create it
                                                  //  if it doesn't exist) in in RO mode.

   bool tpInit = stcPrefs.isKey("nvsInit");       // Test for the existence of the
↪"already initialized" key.

   if (tpInit == false) {
      // If tpInit is 'false', the key "nvsInit" does not yet exist therefore this
      //  must be our first-time run. We need to set up our Preferences namespace keys.␣
↪So...
      stcPrefs.end();                             // close the namespace in RO mode and..
↪.
      stcPrefs.begin("STCPrefs", RW_MODE);        //  reopen it in RW mode.


      // The .begin() method created the "STCPrefs" namespace and since this is our
```

(continues on next page)

(continued from previous page)

```
      //  first-time run we will create our keys and store the initial "factory default"␣
↪values.
      stcPrefs.putUChar("curBright", 10);
      stcPrefs.putString("talChan", "one");
      stcPrefs.putLong("talMax", -220226);
      stcPrefs.putBool("ctMde", true);

      stcPrefs.putBool("nvsInit", true);          // Create the "already initialized"␣
↪key and store a value.

      // The "factory defaults" are created and stored so...
      stcPrefs.end();                             // Close the namespace in RW mode and..
↪.
      stcPrefs.begin("STCPrefs", RO_MODE);        //  reopen it in RO mode so the setup␣
↪code
                                                  //  outside this first-time run 'if'␣
↪block
                                                  //  can retrieve the run-time values
                                                  //  from the "STCPrefs" namespace.
   }

   // Retrieve the operational parameters from the namespace
   //  and save them into their run-time variables.
   currentBrightness = stcPrefs.getUChar("curBright");  //
   tChannel = stcPrefs.getString("talChan");            //  The LHS variables were␣
↪defined
   tChanMax = stcPrefs.getLong("talMax");               //   earlier in the sketch.
   ctMode = stcPrefs.getBool("ctMde");                  //

   // All done. Last run state (or the factory default) is now restored.
   stcPrefs.end();                                     // Close our preferences␣
↪namespace.

   // Carry on with the rest of your setup code...

   // When the sketch is running, it updates any changes to an operational parameter
   //  to the appropriate key-value pair in the namespace.

}
```

4.6.6 Utility Functions

There are a few other functions useful when working with namespaces.

Deleting key-value Pairs

```
preferences.clear();
```

- Deletes *all* the key-value pairs in the currently opened namespace.
  - The namespace still exists.
  - The namespace must be open in read-write mode for this to work.

```
preferences.remove("keyname");
```

- Deletes the "keyname" and value associated with it from the currently opened namespace.
  - The namespace must be open in read-write mode for this to work.
  - Tip: use this to remove the "test key" to force a "factory reset" during the next reboot (see the *Real World Example* above).

If either of the above are used, the `key-value` pair will need to be recreated before using it again.

Determining the Number of Available Keys

For each namespace, Preferences keeps track of the keys in a key table. There must be an open entry in the table before a key can be created. This method will return the number of entires available in the table.

```
freeEntries()
```

To send to the serial monitor the number of available entries the following could be used.

```
Preferences mySketchPrefs;

mySketchPrefs.begin("myPrefs", true);
size_t whatsLeft = freeEntries();    // this method works regardless of the mode in␣
↪which the namespace is opened.
Serial.printf("There are: %u entries available in the namespace table.\n", whatsLeft);
mySketchPrefs.end();
```

The number of available entries in the key table changes depending on the number of keys in the namespace and also the dynamic size of certain types of data stored in the namespace. Details are in the Preferences API Reference.

Do note that the number of entries in the key table does not guarantee that there is room in the opened NVS namespace for all the data to be stored in that namespace. Refer to the espressif Non-volatile storage library documentation for full details.

Determining the Type of a key-value Pair

Keeping track of the data types stored against a key-value pair is one of the bookkeeping tasks left to you. Should you want to discover the Preferences data type stored against a given key, use this method:

```
getType("myKey")
```

As in:

```
PreferenceType whatType = getType("myKey");
```

The value returned is a `PreferenceType` value that maps to a Preferences Type. Refer to the description in the Preferences API Reference for details.

4.6.7 Working with Large Data

Recall that the Preferences library works best for storing many small values, rather than a few large values. Regardless, it may be desirable to store larger amounts of arbitrary data than what is provided by the basic types in the Preferences Types table above.

The library provides the following methods to facilitate this.

```
putBytes("myBytesKey", value, valueLen)
getBytes("myBytesKey", buffer, valueLen)
getBytesLength("myBytesKey")
```

The `put` and `get` `Bytes` methods store and retrieve the data. The `getBytesLength` method is used to find the size of the data stored against the key (which is needed to retrieve `Bytes` data).

As the names of the methods imply, they operate on variable length bytes of data (often referred to as a "blob") and not on individual elements of a certain data type.

Meaning if you store for example an array of type `int16_t` against a `Bytes` type key, the value of that key becomes a series of bytes with no associated data type. Or if you like, all data stored as a blob gets converted to a series of `uint8_t` type bytes.

As a result, when using the `getBytes` method to retrieve the value of the key, what is returned to the buffer is a series of `uint8_t` bytes. It is up to you to manage the data types and size of the arrays and buffers when retrieving `Bytes` data.

Fortunately this is not as difficult as it may sound as the `getBytesLength` method and the `sizeof` operator help with keeping track of it all.

This is best explained with an example. Here the `Bytes` methods are used to store and retrieve an array, while ensuring the data type is preserved.

```
/*
 *  An example sketch using the Preferences "Bytes" methods
 *   to store and retrieve an arbitrary number of bytes in
 *   a namespace.
 */

#include <Preferences.h>

#define RO_MODE true
#define RW_MODE false

void setup() {

    Preferences mySketchPrefs;

    Serial.begin(115200);
    delay(250);

    mySketchPrefs.begin("myPrefs", RW_MODE);    // open (or create) the namespace "myPrefs
↪" in RW mode
    mySketchPrefs.clear();                      // delete any previous keys in this␣
↪namespace
```

(continues on next page)

(continued from previous page)

```
    // Create an array of test values. We're using hex numbers throughout to better show␣
↪how the bytes move around.
    int16_t myArray[] = { 0x1112, 0x2122, 0x3132, 0x4142, 0x5152, 0x6162, 0x7172 };

    Serial.println("Printing myArray...");
    for (int i = 0; i < sizeof(myArray) / sizeof(int16_t); i++) {
        Serial.print(myArray[i], HEX); Serial.print(", ");
    }
    Serial.println("\r\n");

    // In the next statement, the second sizeof() needs to match the data type of the␣
↪elements of myArray
    Serial.print("The number of elements in myArray is: "); Serial.println(␣
↪sizeof(myArray) / sizeof(int16_t) );
    Serial.print("But the size of myArray in bytes is: "); Serial.println(␣
↪sizeof(myArray) );
    Serial.println("");

    Serial.println("Storing myArray into the Preferences namespace \"myPrefs\" against␣
↪the key \"myPrefsBytes\".");
    // Note: in the next statement, to store the entire array, we must use the
    //  size of the arrray in bytes, not the number of elements in the array.
    mySketchPrefs.putBytes( "myPrefsBytes", myArray, sizeof(myArray) );
    Serial.print("The size of \"myPrefsBytes\" is (in bytes): "); Serial.println(␣
↪mySketchPrefs.getBytesLength("myPrefsBytes") );
    Serial.println("");

    int16_t myIntBuffer[20] = {};   // No magic about 20. Just making a buffer (array)␣
↪big enough.
    Serial.println("Retrieving the value of myPrefsBytes into myIntBuffer.");
    Serial.println("   - Note the data type of myIntBuffer matches that of myArray");
    mySketchPrefs.getBytes( "myPrefsBytes", myIntBuffer, mySketchPrefs.getBytesLength(
↪"myPrefsBytes") );

    Serial.println("Printing myIntBuffer...");
    // In the next statement, sizeof() needs to match the data type of the elements of␣
↪myArray
    for (int i = 0; i < mySketchPrefs.getBytesLength("myPrefsBytes") / sizeof(int16_t);␣
↪i++) {
        Serial.print(myIntBuffer[i], HEX); Serial.print(", ");
    }
    Serial.println("\r\n");

    Serial.println("We can see how the data from myArray is actually stored in the␣
↪namespace as follows.");
    uint8_t myByteBuffer[40] = {};  // No magic about 40. Just making a buffer (array)␣
↪big enough.
    mySketchPrefs.getBytes( "myPrefsBytes", myByteBuffer, mySketchPrefs.getBytesLength(
↪"myPrefsBytes") );

    Serial.println("Printing myByteBuffer...");
```

(continues on next page)

(continued from previous page)

```
    for (int i = 0; i < mySketchPrefs.getBytesLength("myPrefsBytes"); i++) {
       Serial.print(myByteBuffer[i], HEX); Serial.print(", ");
    }
    Serial.println("");

}

void loop() {
  ;
}
```

The resulting output is:

```
Printing myArray...
1112, 2122, 3132, 4142, 5152, 6162, 7172,

The number of elements in myArray is: 7
But the size of myArray in bytes is: 14

Storing myArray into the Preferences namespace "myPrefs" against the key "myPrefsBytes".
The size of "myPrefsBytes" is (in bytes): 14

Retrieving the value of myPrefsBytes into myIntBuffer.
   - Note the data type of myIntBuffer matches that of myArray
Printing myIntBuffer...
1112, 2122, 3132, 4142, 5152, 6162, 7172,

We can see how the data from myArray is actually stored in the namespace as follows.
Printing myByteBuffer...
12, 11, 22, 21, 32, 31, 42, 41, 52, 51, 62, 61, 72, 71,
```

You can copy the sketch and change the data type and values in `myArray` and follow along with the code and output to see how the `Bytes` methods work. The data type of `myIntBuffer` should be changed to match that of `myArray` (and check the "`sizeof()`'s" where indicated in the comments).

The main takeaway is to remember you're working with bytes and so attention needs to be paid to store all the data based on the size of its type and to manage the buffer size and data type for the value retrieved.

4.6.8 Multiple Namespaces

As stated earlier, multiple namespaces can exist in the Preferences NVS partition. However, only one namespace at a time can be open (in use).

If you need to access a different namespace, close the one before opening the other. For example:

```
Preferences currentNamespace;

   currentNamespace.begin("myNamespace", false);
      // do stuff...

   currentNamespace.end();                              // closes 'myNamespace'
```

(continues on next page)

(continued from previous page)

```
  currentNamespace.begin("myOtherNamespace", false);   // opens a different Preferences␣
↪namesspace.
    // do other stuff...

  currentNamespace.end();                              // closes 'myOtherNamespace'
```

Here the "currentNamespace" object is reused, but different Preferences objects can be declared and used. Just remember to keep it all straight as all "putX's" and "getX's", etc. will only operate on the single currently opened namespace.

4.6.9 A Closer Look at getX

Methods in the Preferences library return a status code that can be used to determine if the method completed successfully. This is described in the Preferences API Reference.

Assume we have a key named "favourites" that contains a value of a String data type.

After executing the statement:

```
dessert = mySketchPrefs.getString("favourites");
```

the variable dessert will contain the value of the string stored against the key "favourites".

But what if something went wrong and the getString call failed to retrieve the key value? How would we be able to detect the error?

With Preferences, the getX methods listed in Table 2 below will return a default value if an error is encountered.

Table 2: **Table 2 — getX Methods Defaults**

Preferences Type	Default Return Value
Char, UChar, Short, UShort, Int, UInt, Long, ULong, Long64, ULong64	0
Bool	false
Float Double	NAN
String (String)	""
String (* buf)	\0

Thus to detect an error we could compare the value returned against its default return value and if they are equal assume an error occurred and take the appropriate action.

But what if a method default return value is also a potential legitimate value? How can we then know if an error occurred?

As it turns out, the complete form of the getX methods for each of the Preferences Types in Table 2 is:

```
preferences.getX("myKey", myDefault)
```

In this form the method will return either the value associated with "myKey" or, if an error occurred, return the value myDefault, where myDefault must be the same data type as the getX.

Returning to the example above:

```
dessert = mySketchPrefs.getString("favourites", "gravel");
```

will assign to the variable `dessert` the String `gravel` if an error occurred, or the value stored against the key `favourites` if not.

If we predetermine a default value that is outside all legitimate values, we now have a way to test if an error actually occurred.

In summary, if you need to confirm that a value was retrieved without error from a namespace, use the complete form of the `getX` method with a predetermined default "this can only happen if an error" value and compare that against the value returned by the call. Otherwise, you can omit the default value as the call will return the default for that particular `getX` method.

Additional detail is given in the Preferences API Reference.

4.6.10 Advanced Item

In the arduino-esp32 implementation of Preferences there is no method to completely remove a namespace. As a result, over the course of a number of projects, it is possible that the ESP32 NVS Preferences partition becomes cluttered or full.

To completely erase and reformat the NVS memory used by Preferences, create and run a sketch that contains:

```
#include <nvs_flash.h>

void setup() {

    nvs_flash_erase();      // erase the NVS partition and...
    nvs_flash_init();       // initialize the NVS partition.
    while (true);

}

void loop() {
   ;
}
```

Warning: You should download a new sketch to your board immediately after running the above or else it will reformat the NVS partition every time it is powered up or restarted!

4.6.11 Resources

- Preferences API Reference
- Non-volatile storage library (espressif-IDF API Reference)
- Official ESP-IDF documentation (espressif-IDF Reference)

4.6.12 Contribute

To contribute to this project, see How to contribute.

If you have any **feedback** or **issue** to report on this tutorial, please open an issue or fix it by creating a new PR. Contributions are more than welcome!

Before creating a new issue, be sure to try the Troubleshooting and to check if the same issue was already created by someone else.

CHAPTER

FIVE

ADVANCED UTILITIES

5.1 Library Builder

5.1.1 About

Espressif provides a tool to simplify building your own compiled libraries for use in Arduino IDE (or your favorite IDE).

This tool can be used to change the project or a specific configuration according to your needs.

5.1.2 Installing

To install the Library Builder into your environment, please, follow the instructions below.

- Clone the ESP32 Arduino lib builder:

```
git clone https://github.com/espressif/esp32-arduino-lib-builder
```

- Go to the `esp32-arduino-lib-builder` folder:

```
cd esp32-arduino-lib-builder
```

- Build:

```
./build.sh
```

If everything works, you may see the following message: `Successfully created esp32 image.`

Dependencies

To build the library you will need to install some dependencies. Maybe you already have installed it, but it is a good idea to check before building.

- Install all dependencies (**Ubuntu**):

```
sudo apt-get install git wget curl libssl-dev libncurses-dev flex bison gperf cmake␣
↪ninja-build ccache jq
```

- Install Python and upgrade pip:

```
sudo apt-get install python3
sudo pip install --upgrade pip
```

- Install all required packages:

```
pip install --user setuptools pyserial click cryptography future pyparsing pyelftools
```

5.1.3 Building

If you have all the dependencies met, it is time to build the libraries.

To build using the default configuration:

```
./build.sh
```

Custom Build

There are some options to help you create custom libraries. You can use the following options:

Usage

```
build.sh [-s] [-A arduino_branch] [-I idf_branch] [-i idf_commit] [-c path] [-t <target>
↪] [-b <build|menuconfig|idf_libs|copy_bootloader|mem_variant>] [config ...]
```

Skip Install/Update

Skip installing/updating of ESP-IDF and all components

```
./build.sh -s
```

This option can be used if you already have the ESP-IDF and all components already in your environment.

Set Arduino-ESP32 Branch

Set which branch of arduino-esp32 to be used for compilation

```
./build.sh -A <arduino_branch>
```

Set ESP-IDF Branch

Set which branch of ESP-IDF is to be used for compilation

```
./build.sh -I <idf_branch>
```

Set the ESP-IDF Commit

Set which commit of ESP-IDF to be used for compilation

```
./build.sh -i <idf_commit>
```

Deploy

Deploy the build to github arduino-esp32

```
./build.sh -d
```

Set the Arduino-ESP32 Destination Folder

Set the arduino-esp32 folder to copy the result to. ex. '$HOME/Arduino/hardware/espressif/esp32'

```
./build.sh -c <path>
```

This function is used to copy the compiled libraries to the Arduino folder.

Set the Target

Set the build target(chip). ex. 'esp32s3'

```
./build.sh -t <target>
```

This build command will build for the ESP32-S3 target. You can specify other targets.

- esp32
- esp32s2
- esp32c3
- esp32s3

Set Build Type

Set the build type. ex. 'build' to build the project and prepare for uploading to a board.

Note: This command depends on the `-t` argument.

```
./build.sh -t esp32 -b <build|menuconfig|idf_libs|copy_bootloader|mem_variant>
```

Additional Configuration

Specify additional configs to be applied. ex. 'qio 80m' to compile for QIO Flash@80MHz. Requires -b

Note: This command requires the `-b` to work properly.

```
./build.sh -t esp32 -b idf_libs qio 80m
```

5.2 Arduino as an ESP-IDF component

This method is recommended for advanced users. To use this method, you will need to have the ESP-IDF toolchain installed.

For a simplified method, see Installing using Boards Manager.

5.2.1 ESP32 Arduino lib-builder

If you don't need any modifications in the default Arduino ESP32 core, we recommend you to install using the Boards Manager.

Arduino Lib Builder is the tool that integrates ESP-IDF into Arduino. It allows you to customize the default settings used by Espressif and try them in Arduino IDE.

For more information see Arduino lib builder

5.2.2 Installation

Note: Latest Arduino Core ESP32 version is now compatible with ESP-IDF v4.4. Please consider this compatibility when using Arduino as a component in ESP-IDF.

1. Download and install ESP-IDF.
    - For more information see Get Started.
2. Create a blank ESP-IDF project (use sample_project from /examples/get-started) or choose one of the examples.
3. In the project folder, create a new folder called `components` and clone this repository inside the newly created folder.

```
mkdir -p components && \
cd components && \
git clone https://github.com/espressif/arduino-esp32.git arduino && \
cd arduino && \
git submodule update --init --recursive && \
cd ../.. && \
idf.py menuconfig
```

Note: If you use Arduino with ESP-IDF often, you can place the arduino folder into global components folder.

If you're targeting the ESP32-S2 or ESP32-S3 and you want to use USBHID classes such as `USBHID`, `USBHIDConsumerControl`, `USBHIDGamepad`, `USBHIDKeyboard`, `USBHIDMouse`, `USBHIDSystemControl`, or `USBHIDVendor`:

1. Clone these nested repos somewhere:

```
git clone https://github.com/espressif/esp32-arduino-lib-builder.git esp32-arduino-lib-
↪builder && \
git clone https://github.com/hathach/tinyusb.git esp32-arduino-lib-builder/components/
↪arduino_tinyusb/tinyusb
```

2. In the project folder, edit `CMakeLists.txt` and add the following before the `project()` line:

```
set(EXTRA_COMPONENT_DIRS <path to esp32-arduino-lib-builder/components/arduino_tinyusb>)
```

5.2.3 Configuration

Depending on one of the two following options, in the menuconfig set the appropriate settings.

Go to the section `Arduino Configuration --->`

1. For usage of `app_main()` function - Turn off `Autostart Arduino setup and loop on boot`
2. For usage of `setup()` and `loop()` functions - Turn on `Autostart Arduino setup and loop on boot`

Experienced users can explore other options in the Arduino section.

After the setup you can save and exit:

- Save [S]
- Confirm default filename [Enter]
- Close confirmation window [Enter] or [Space] or [Esc]
- Quit [Q]

Option 1. Using Arduino setup() and loop()

- In main folder rename file *main.c* to *main.cpp*.
- In main folder open file *CMakeList.txt* and change *main.c* to *main.cpp* as described below.
- Your main.cpp should be formatted like any other sketch.

```
//file: main.cpp
#include "Arduino.h"

void setup(){
  Serial.begin(115200);
  while(!Serial){
    ; // wait for serial port to connect
  }
}

void loop(){
    Serial.println("loop");
    delay(1000);
}
```

Option 2. Using ESP-IDF appmain()

In main.c or main.cpp you need to implement `app_main()` and call `initArduino();` in it.

Keep in mind that setup() and loop() will not be called in this case. Furthermore the `app_main()` is single execution as a normal function so if you need an infinite loop as in Arduino place it there.

```
//file: main.c or main.cpp
#include "Arduino.h"

extern "C" void app_main()
{
  initArduino();

  // Arduino-like setup()
  Serial.begin(115200);
  while(!Serial){
    ; // wait for serial port to connect
  }

  // Arduino-like loop()
  while(true){
    Serial.println("loop");
  }

  // WARNING: if program reaches end of function app_main() the MCU will restart.
}
```

Build, flash and monitor

- For both options use command `idf.py -p <your-board-serial-port> flash monitor`
- The project will build, upload and open the serial monitor to your board
  - Some boards require button combo press on the board: press-and-hold Boot button + press-and-release RST button, release Boot button
  - After a successful flash, you may need to press the RST button again
  - To terminate the serial monitor press [Ctrl] + [ ]]

5.2.4 Logging To Serial

If you are writing code that does not require Arduino to compile and you want your *ESP_LOGx* macros to work in Arduino IDE, you can enable the compatibility by adding the following lines:

```
#ifdef ARDUINO_ARCH_ESP32
#include "esp32-hal-log.h"
#endif
```

5.2.5 FreeRTOS Tick Rate (Hz)

The Arduino component requires the FreeRTOS tick rate *CONFIG_FREERTOS_HZ* set to 1000Hz in *make menuconfig -> Component config -> FreeRTOS -> Tick rate*.

5.2.6 Compilation Errors

As commits are made to esp-idf and submodules, the codebases can develop incompatibilities that cause compilation errors. If you have problems compiling, follow the instructions in Issue #1142 to roll esp-idf back to a different version.

5.3 OTA Web Update

OTAWebUpdate is done with a web browser that can be useful in the following typical scenarios:

- Once the application developed and loading directly from Arduino IDE is inconvenient or not possible
- after deployment if user is unable to expose Firmware for OTA from external update server
- provide updates after deployment to small quantity of modules when setting an update server is not practicable

5.3.1 Requirements

- The ESP and the computer must be connected to the same network

5.3.2 Implementation

The sample implementation has been done using:

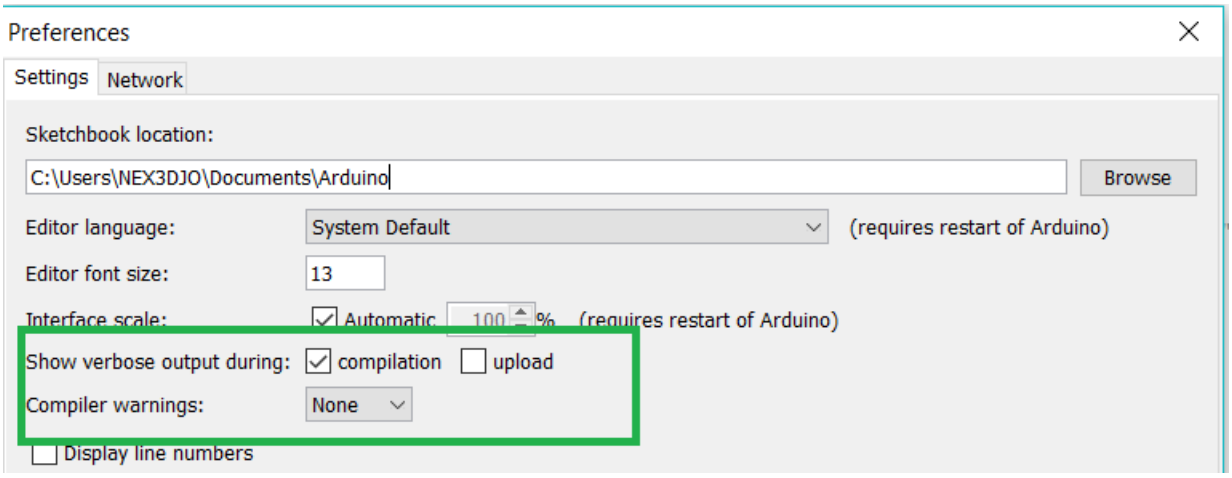
- Example sketch `OTAWebUpdater.ino`.
- ESP32 Board.

You can also use another module if it meets Flash chip size of the sketch

Before you begin, please make sure that you have the following software installed:

- Arduino IDE
- **Host software depending on O/S you use**
  - Avahi for Linux
  - Bonjour for Windows
  - Mac OSX and iOS - support is already built in / no any extra s/w is required

Prepare the sketch and configuration for initial upload with a serial port - Start Arduino IDE and load sketch OTAWebUpdater.ino available under File > Examples > OTAWebUpdater.ino - Update ssid and pass in the sketch so the module can join your Wi-Fi network - Open File > Preferences, look for "Show verbose output during:" and check out "compilation" option



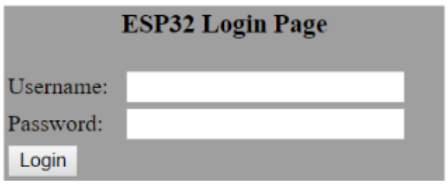
- Upload sketch (Ctrl+U)
- Now open web browser and enter the url, i.e. http://esp32.local. Once entered, browser should display a form
- username = admin
- password = admin

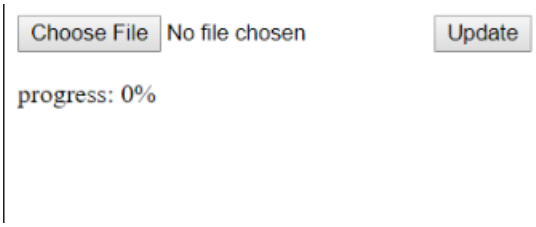
Note: *If entering "http://ESP32.local" does not work, try replacing "ESP32" with module's IP address. This workaround is useful in case the host software installed does not work.*

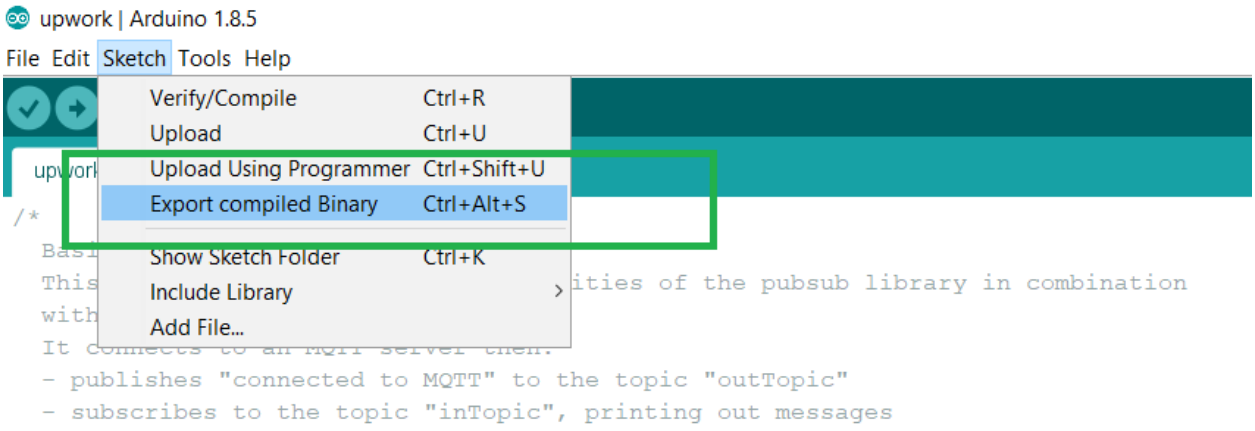
Now click on the Login button and browser will display an upload form

For Uploading the New Firmware, you need to provide the Binary File of your Code.

Exporting Binary file of the Firmware (Code) - Open up the Arduino IDE - Open up the Code, for Exporting up Binary file - Now go to Sketch > export compiled Binary


ESP32 Login Page
Username:
Password:
Login


Choose File
No file chosen
Update
progress: 0%


upwork | Arduino 1.8.5
File Edit Sketch Tools Help
Verify/Compile Ctrl+R
Upload Ctrl+U
Upload Using Programmer Ctrl+Shift+U
Export compiled Binary Ctrl+Alt+S
Show Sketch Folder Ctrl+K
Include Library
Add File...
/*
- publishes "connected to MQTT" to the topic "outTopic"
- subscribes to the topic "inTopic", printing out messages

- Binary file is exported to the same Directory where your code is present

Once you are comfortable with this procedure, go ahead and modify OTAWebUpdater.ino sketch to print some additional messages and compile it. Then, export the new binary file and upload it using web browser to see entered changes on a Serial Monitor.

5.4 makeEspArduino

The makeEspArduino is a generic makefile for any ESP8266/ESP32 Arduino project. Using it instead of the Arduino IDE makes it easier to do automated and production builds.

CHAPTER

SIX

FREQUENTLY ASKED QUESTIONS

6.1 How to modify an sdkconfig option in Arduino?

Arduino-esp32 project is based on ESP-IDF. While ESP-IDF supports configuration of various compile-time options (known as "Kconfig options" or "sdkconfig options") via a "menuconfig" tool, this feature is not available in Arduino IDE.

To use the arduino-esp32 core with a modified sdkconfig option, you need to use ESP-IDF to compile Arduino libraries. Please see *Arduino as an ESP-IDF component* and *Library Builder* for the two solutions available.

Note that modifying `sdkconfig` or `sdkconfig.h` files found in the arduino-esp32 project tree **does not** result in changes to these options. This is because ESP-IDF libraries are included into the arduino-esp32 project tree as pre-built libraries.

6.2 How to compile libs with different debug level?

The short answer is `esp32-arduino-lib-builder/configs/defconfig.common:44`. A guide explaining the process can be found here <guides/core_debug>

CHAPTER

SEVEN

TROUBLESHOOTING

7.1 Common Issues

Here are some of the most common issues around the ESP32 development using Arduino.

Note: Please consider contributing if you have found any issues with the solution here.

7.1.1 Installing

Here are the common issues during the installation.

7.1.2 Building

Missing Python: "python": executable file not found in $PATH

You are trying to build your sketch using Ubuntu and this message appears:

```
"exec: "python": executable file not found in $PATH
Error compiling for board ESP32 Dev Module"
```

Solution

To avoid this error, you can install the `python-is-python3` package to create the symbolic links.

```
sudo apt install python-is-python3
```

If you are not using Ubuntu, you can check if you have the Python correctly installed or the presence of the symbolic links/environment variables.

7.1.3 Flashing

Why is my board not flashing/uploading when I try to upload my sketch?

To be able to upload the sketch via serial interface, the ESP32 must be in the download mode. The download mode allows you to upload the sketch over the serial port and to get into it, you need to keep the **GPIO0** in LOW while a resetting (**EN** pin) cycle. If you are trying to upload a new sketch and your board is not responding, there are some possible reasons.

Possible fatal error message from the Arduino IDE:

> *A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header*

Solution

Here are some steps that you can try to:

- Check your USB cable and try a new one.
- Change the USB port.
- Check your power supply.
- In some instances, you must keep **GPIO0** LOW during the uploading process via the serial interface.
- Hold down the **"BOOT"** button in your ESP32 board while uploading/flashing.

In some development boards, you can try adding the reset delay circuit, as described in the *Power-on Sequence* section on the ESP32 Hardware Design Guidelines in order to get into the download mode automatically.

7.1.4 Hardware

Why is my computer not detecting my board?

If your board is not being detected after connecting to the USB, you can try to:

Solution

- Check if the USB driver is missing. - USB Driver Download Link
- Check your USB cable and try a new one.
- Change the USB port.
- Check your power supply.
- Check if the board is damaged or defective.

7.1.5 Wi-Fi

Why does the board not connect to WEP/WPA-"encrypted" Wi-Fi?

Please note that WEP/WPA has significant security vulnerabilities and its use is strongly discouraged. The support may therefore be removed in the future. Please migrate to WPA2 or newer.

Solution

Nevertheless, it may be necessary to connect to insecure networks. To do this, the security requirement of the ESP32 must be lowered to an insecure level by using:

```
WiFi.setMinSecurity(WIFI_AUTH_WEP); // Lower min security to WEP.
// or
WiFi.setMinSecurity(WIFI_AUTH_WPA_PSK); // Lower min security to WPA.
```

Why does the board not connect to WPA3-encrypted Wi-Fi?

WPA3 support is resource intensive and may not be compiled into the used SDK.

Solution

- Check WPA3 support by your SDK.
- Compile your custom SDK with WPA3 support.

Sample code to check SDK WPA3 support at compile time:

```
#ifndef CONFIG_ESP32_WIFI_ENABLE_WPA3_SAE
#warning "No WPA3 support."
#endif
```

CHAPTER

EIGHT

CONTRIBUTIONS GUIDE

We welcome contributions to the Arduino ESP32 project!

8.1 How to Contribute

Contributions to the Arduino ESP32 (fixing bugs, adding features, adding documentation) are welcome. We accept contributions via Github Pull Requests.

8.2 Before Contributing

Before sending us a Pull Request, please consider this:

- Is the contribution entirely your own work, or is it already licensed under an LGPL 2.1 compatible Open Source License? If not, cannot accept it.
- Is the code adequately commented and can people understand how it is structured?
- Is there documentation or examples that go with code contributions?
- Are comments and documentation written in clear English, with no spelling or grammar errors?
- Example contributions are also welcome.
  - If you are contributing by adding a new example, please use the Arduino style guide and the example guideline below.
- If the contribution contains multiple commits, are they grouped together into logical changes (one major change per pull request)? Are any commits with names like "fixed typo" squashed into previous commits?

If you're unsure about any of these points, please open the Pull Request anyhow and then ask us for feedback.

8.3 Pull Request Process

After you open the Pull Request, there will probably be some discussion in the comments field of the request itself.

Once the Pull Request is ready to merge, it will first be merged into our internal git system for "in-house" automated testing.

If this process passes, it will be merged into the public GitHub repository.

8.4 Example Contribution Guideline

8.4.1 Checklist

- Check if your example proposal has no similarities to the project (**already existing examples**)
- Use the Arduino style guide
- Add the header to all source files
- Add the *README.md* file
- Add inline comments if needed
- Test the example

8.4.2 Header

All the source files must include the header with the example name and license, if applicable. You can change this header as you wish, but it will be reviewed by the community and may not be accepted.

Ideally, you can add some description about the example, links to the documentation, or the author's name. Just have in mind to keep it simple and short.

Header Example

8.4.3 README file

The **README.md** file should contain the example details.

Please see the recommended **README.md** file in the example template folder.

8.4.4 Inline Comments

Inline comments are important if the example contains complex algorithms or specific configurations that the user needs to change.

Brief and clear inline comments are really helpful for the example understanding and it's fast usage.

Example

See the FTM example as a reference.

and

8.4.5 Testing

Be sure you have tested the example in all the supported targets. If the example works only with specific targets, add this information in the **README.md** file on the **Supported Targets** and in the example code as an inline comment.

Example

and

8.4.6 Example Template

The example template can be found here and can be used as a reference.

8.5 Legal Part

Before a contribution can be accepted, you will need to sign our contributor agreement. You will be prompted for this automatically as part of the Pull Request process.