Arduino-ESP32

Release 2.0.2

Espressif

Mar 02, 2022

CONTENTS:

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	No	Yes	ESP32-S3

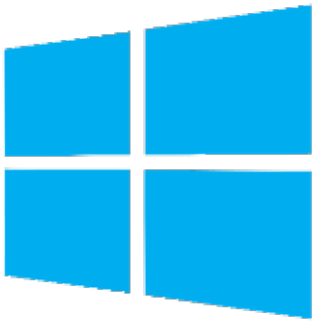
See Boards for more details about ESP32 development boards.

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
```

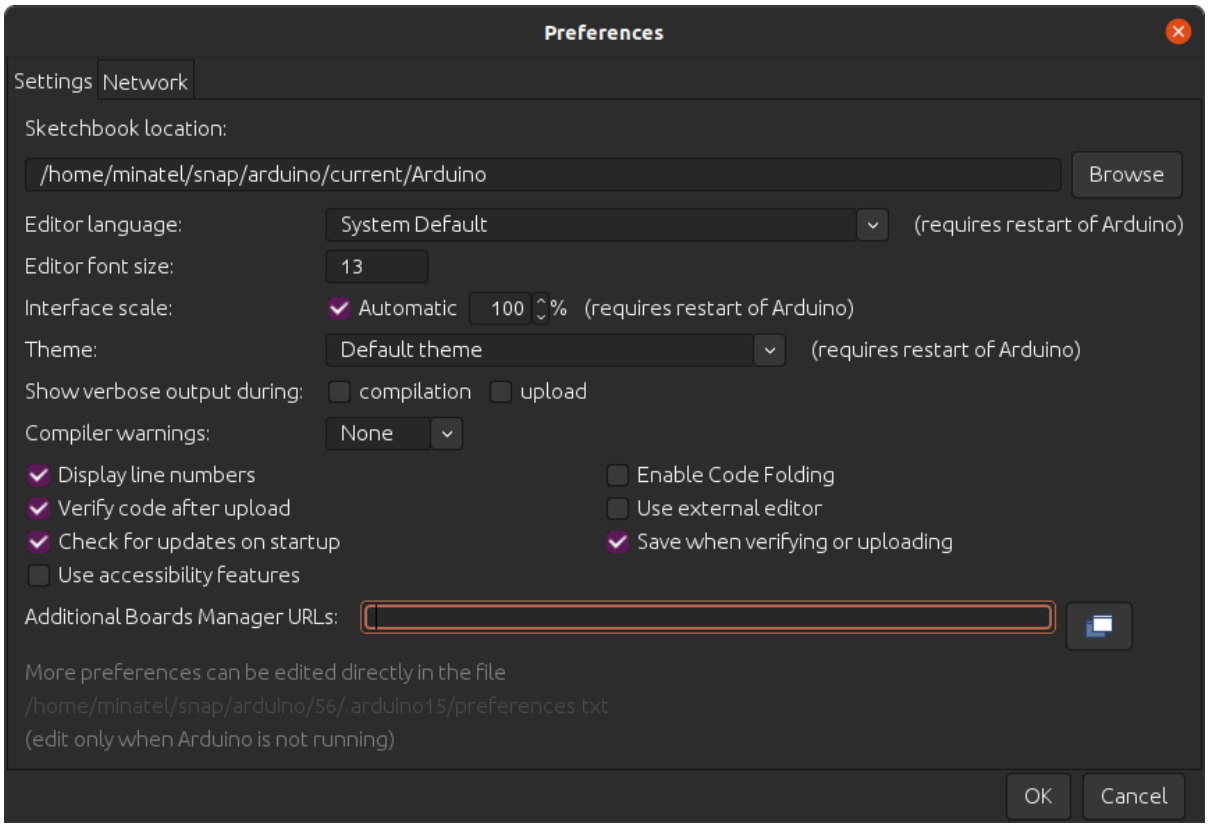
- Development release link:

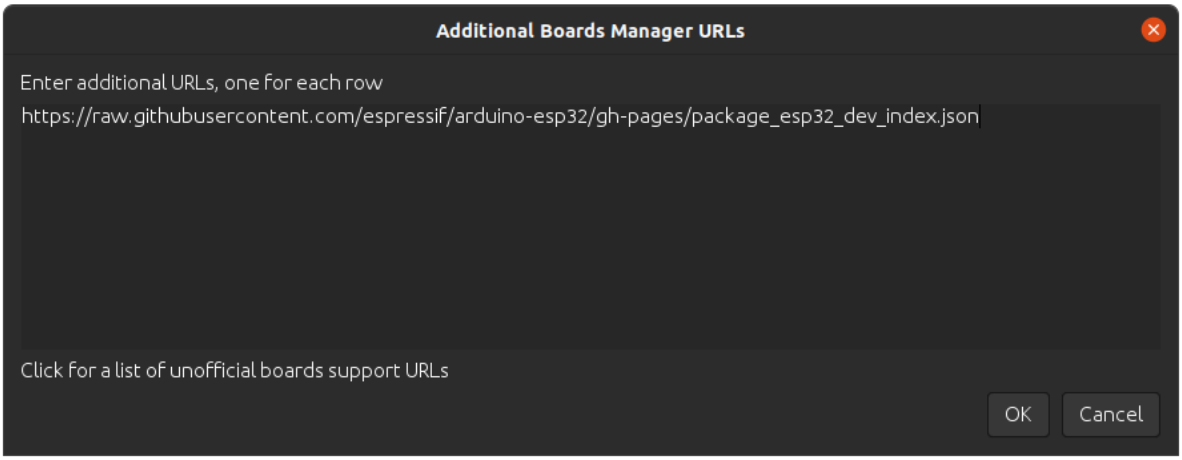
```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_
↪dev_index.json
```

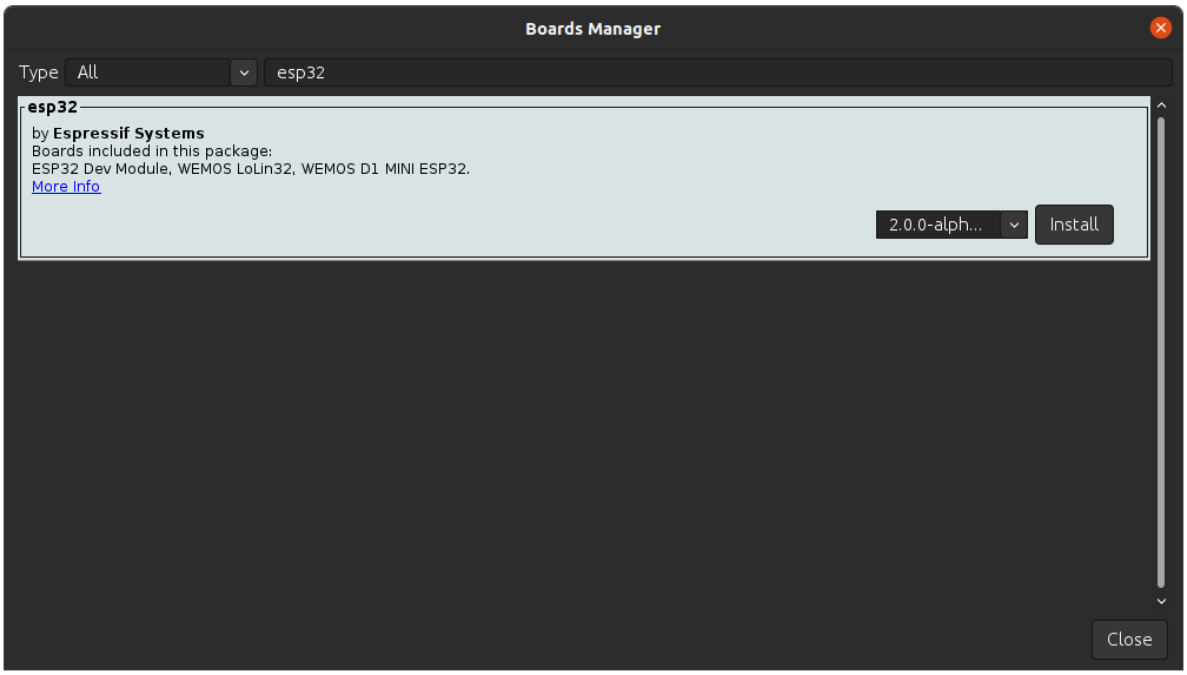
Note: Starting with the Arduino IDE version 1.6.4, Arduino allows installation of third-party platform packages using Boards Manager. We have packages available for Windows, macOS, and Linux.

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 one of most popular embedded development tool. Currently, it supports Arduino ESP32 and ESP-IDF from Espressif (other platforms are also supported).

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

To test the latest Arduino ESP32, you need to change your project *platform.ini* accordingly.

- Start a new project and select one of the available board. You can change after by changing the *platform.ini* file.
- For ESP32

```
[env:arduino-esp32]
platform = https://github.com/platformio/platform-espressif32.git#feature/arduino-
↪upstream
board = esp32dev
framework = arduino
```

(continues on next page)

(continued from previous page)

```
platform_packages =
   framework-arduinoespressif32 @ https://github.com/espressif/arduino-esp32#master
```

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

```
[env:arduino-esp32s2]
platform = https://github.com/platformio/platform-espressif32.git#feature/arduino-
↪upstream
board = esp32-s2-saola-1
framework = arduino
platform_packages =
   framework-arduinoespressif32 @ https://github.com/espressif/arduino-esp32#master
```

- For ESP32-C3 (ESP32-S3-DevKitM-1 board)

```
[env:arduino-esp32c3]
platform = https://github.com/platformio/platform-espressif32.git#feature/arduino-
↪upstream
board = esp32-c3-devkitm-1
framework = arduino
platform_packages =
   framework-arduinoespressif32 @ https://github.com/espressif/arduino-esp32#master
```

Now you're able to use the latest Arduino ESP32 support directly from Espressif GitHub repository.

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.

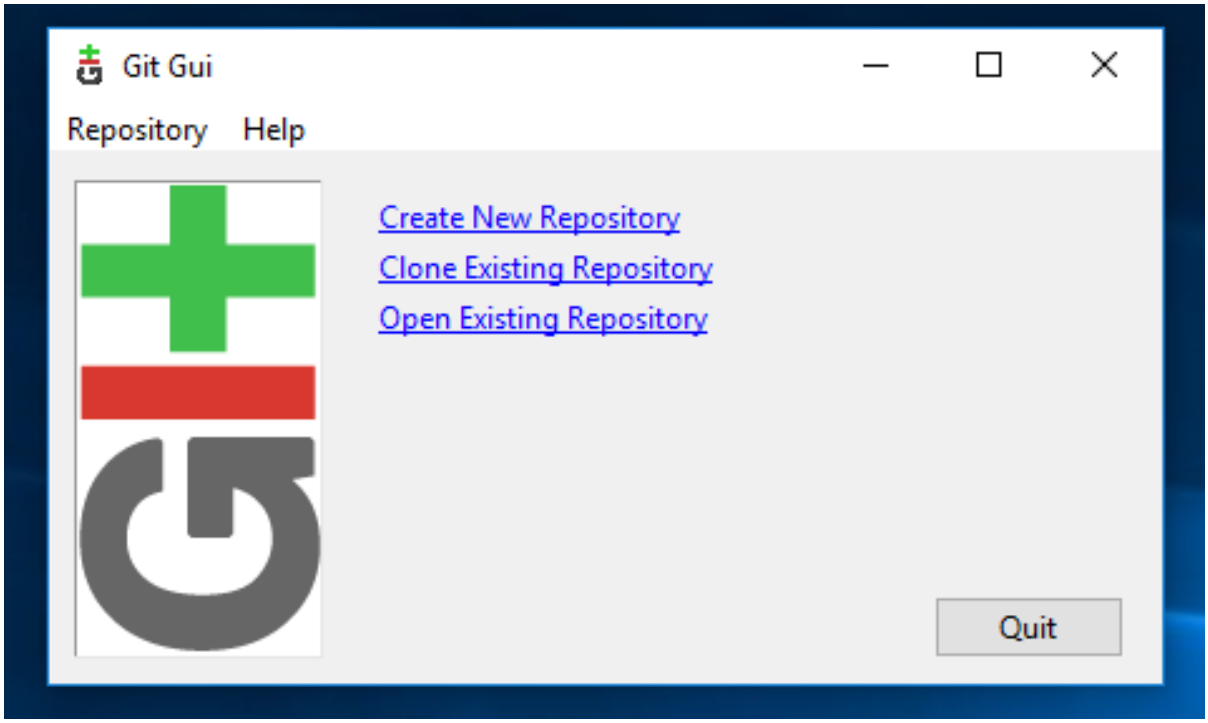
The Arduino ESP32 v1.0.6 still works on WIN32. You might want to install python 3.8.x because it is the latest release supported by Windows 7.

Steps to install Arduino ESP32 support on Windows:

Step 1

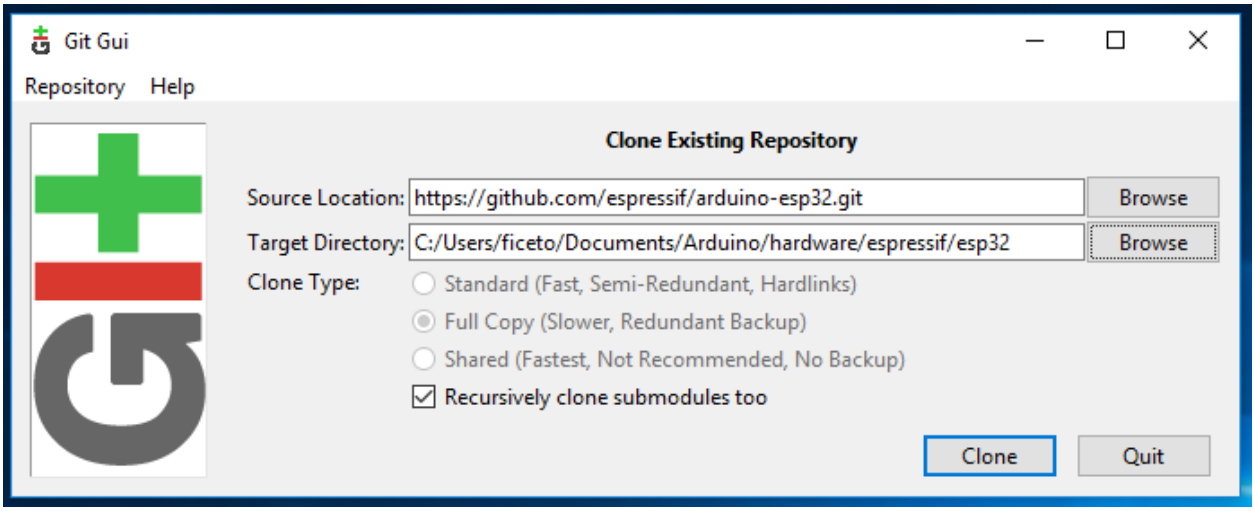
1. Download and install the latest Arduino IDE `Windows Installer` from [arduino.cc](https://www.arduino.cc/en/Main/Software)
2. Download and install Git from [git-scm.com](https://git-scm.com/download/win)
3. Start `Git GUI` and do the following steps:

- Select `Clone Existing Repository`
- **Select source and destination**



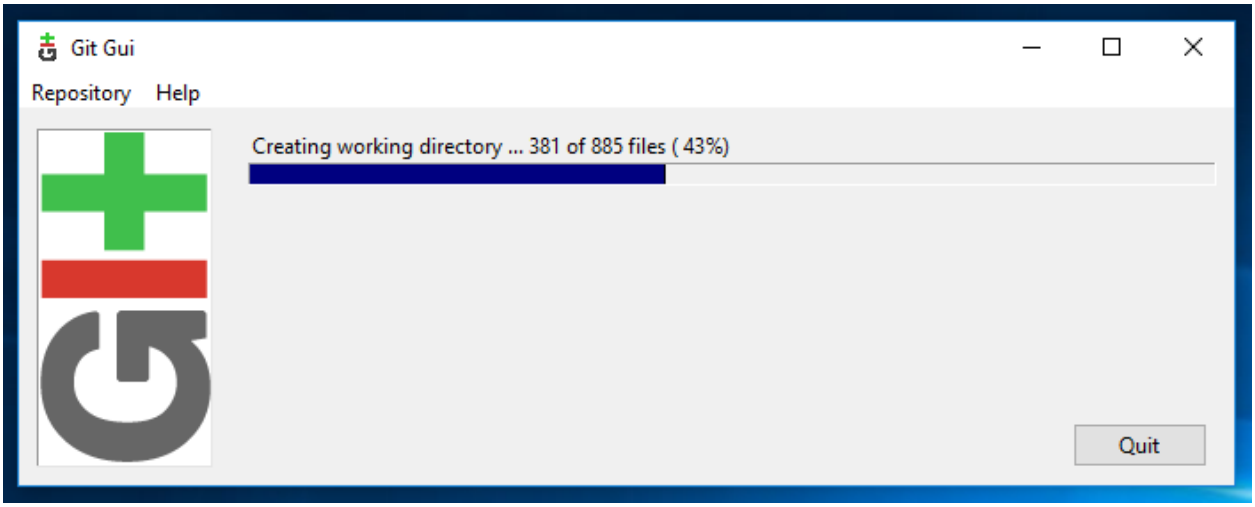
- Sketchbook Directory: Usually `C:/Users/[YOUR_USER_NAME]/Documents/Arduino` and is listed underneath the "Sketchbook location" in Arduino preferences.
- Source Location: `https://github.com/espressif/arduino-esp32.git`
- Target Directory: `[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32`
- Click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

Step 2



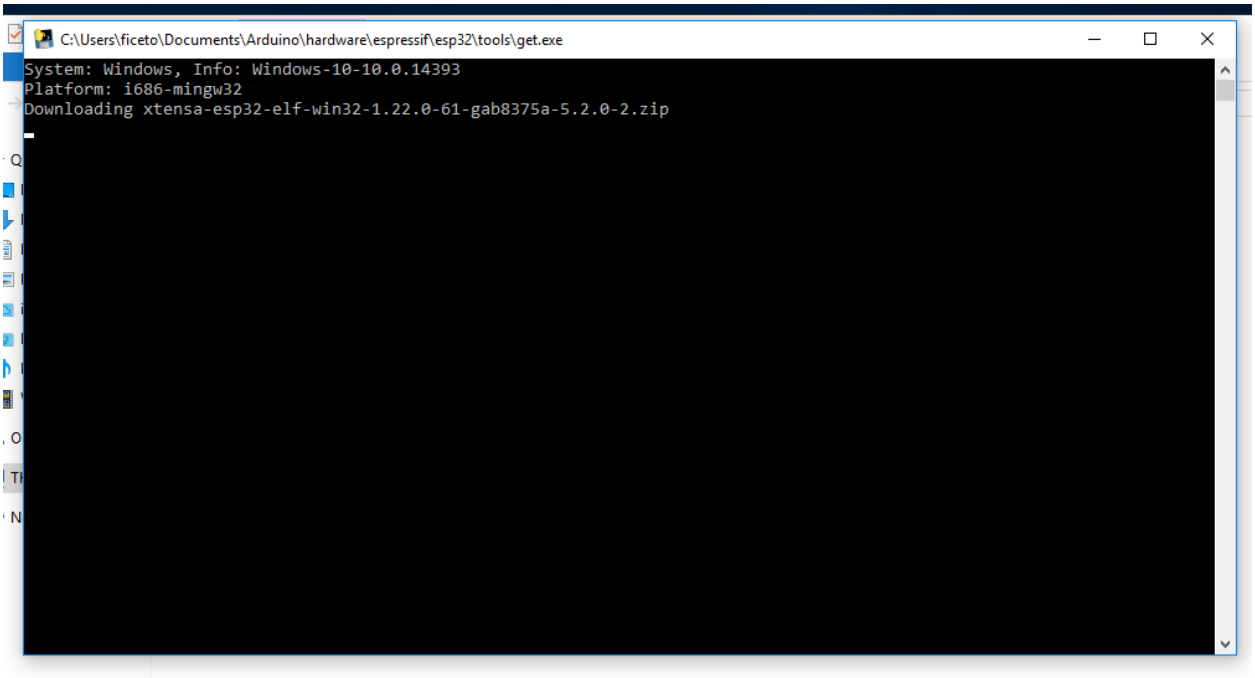
Step 3

- 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`

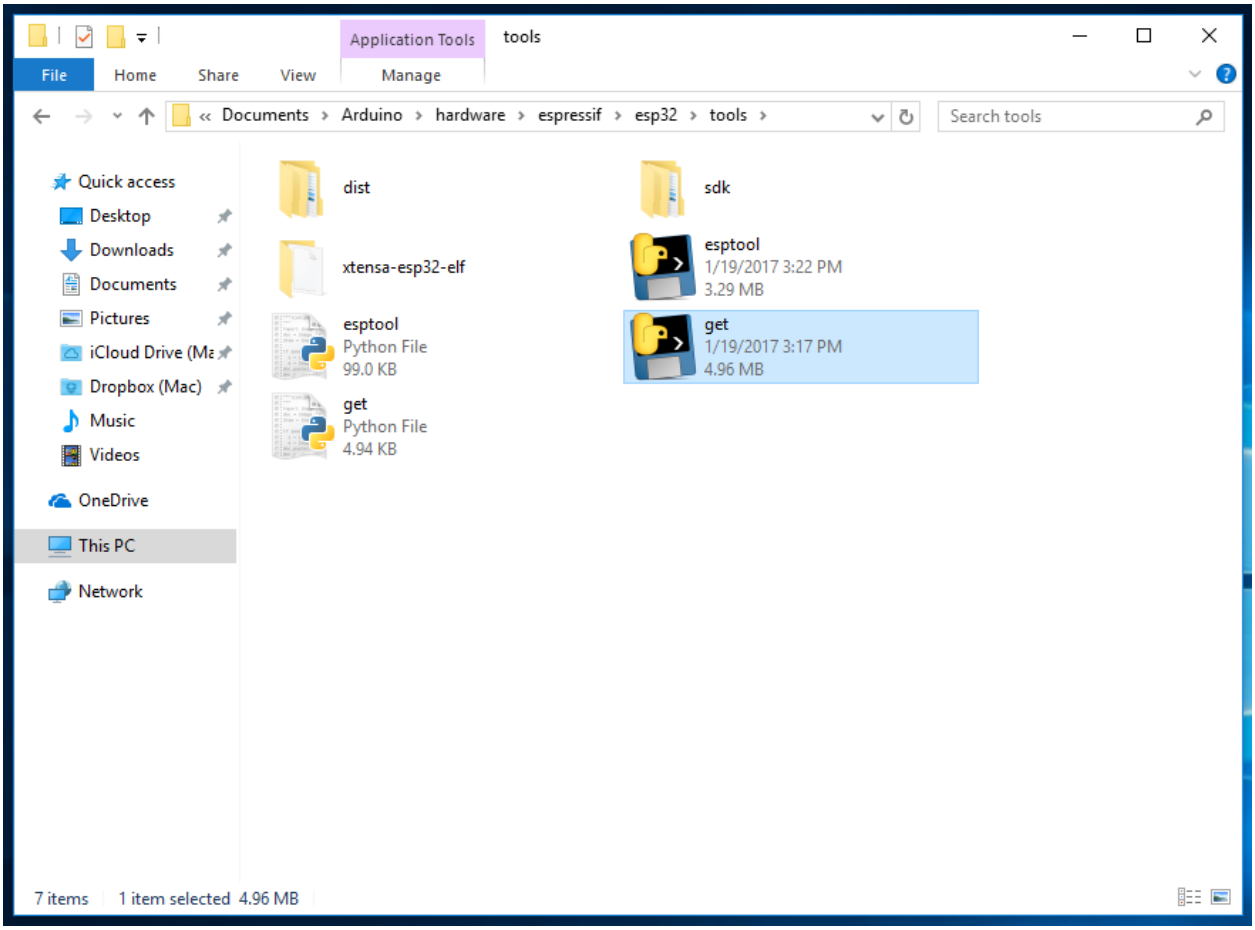
Step 4

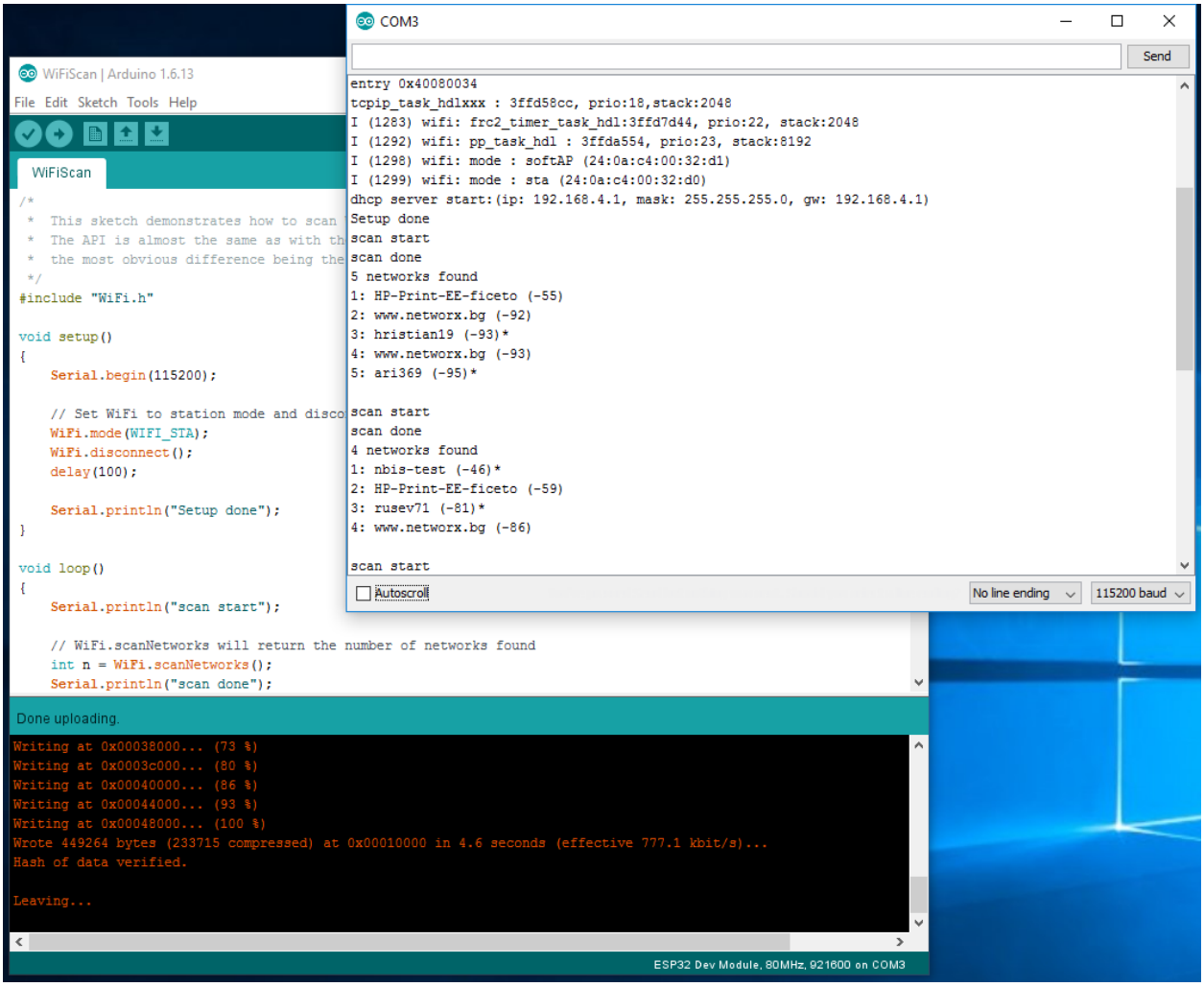


- When `` `get.exe` `` finishes, you should see the following files in the directory

Step 5

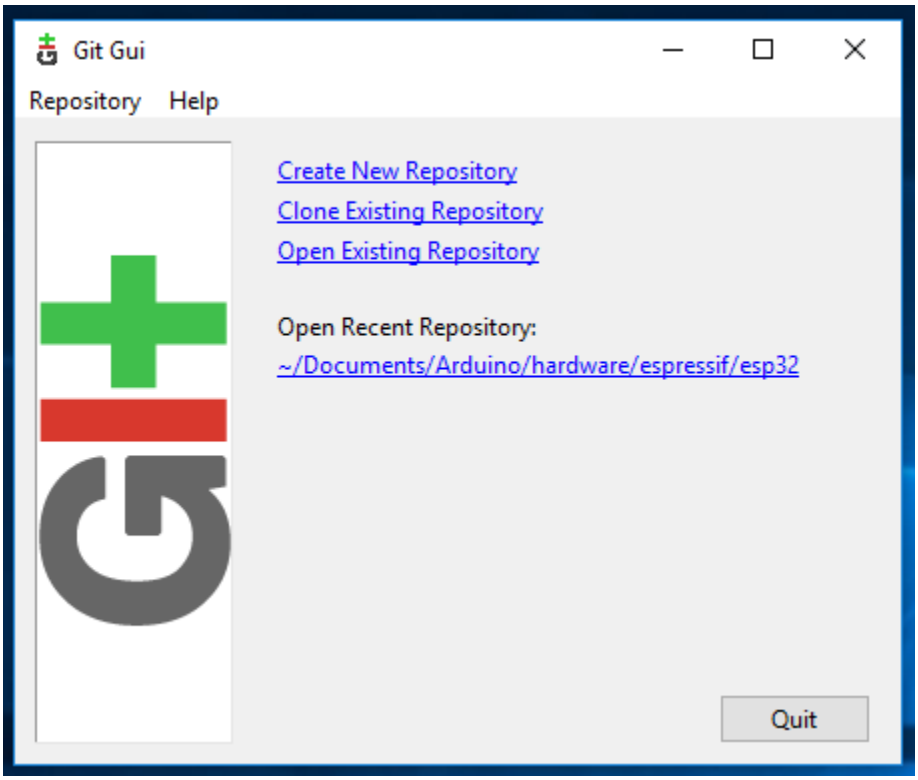
1. Plug your ESP32 board and wait for the drivers to install (or install manually any that might be required)
2. Start Arduino IDE
3. Select your board in `Tools > Board` menu
4. Select the COM port that the board is attached to
5. Compile and upload (You might need to hold the boot button while uploading)


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

How to update to the latest code

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



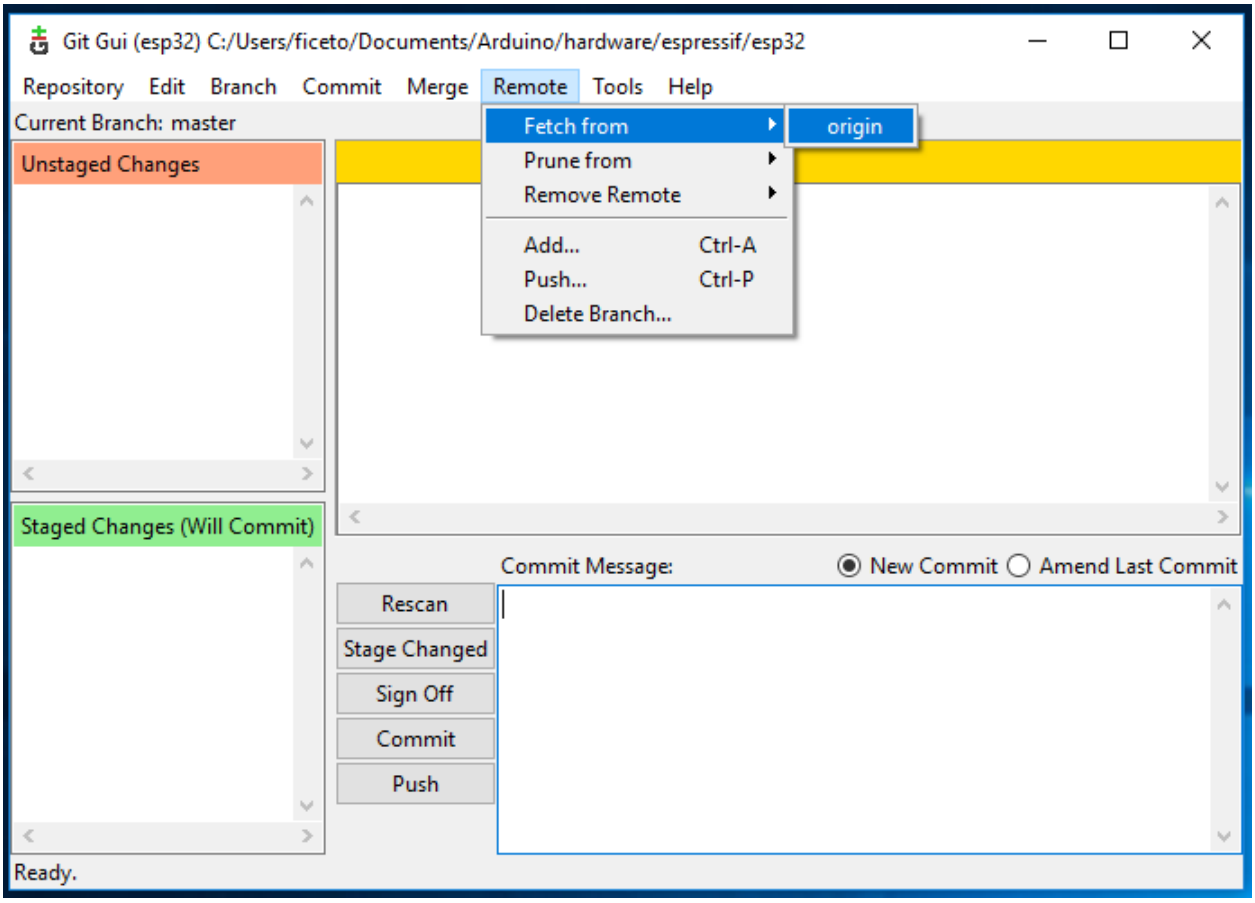
1. From menu `Remote` select `Fetch from` > `origin`
1. Wait for git to pull any changes and close `Git GUI`
2. Open `[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32/tools` and double-click `get.exe`

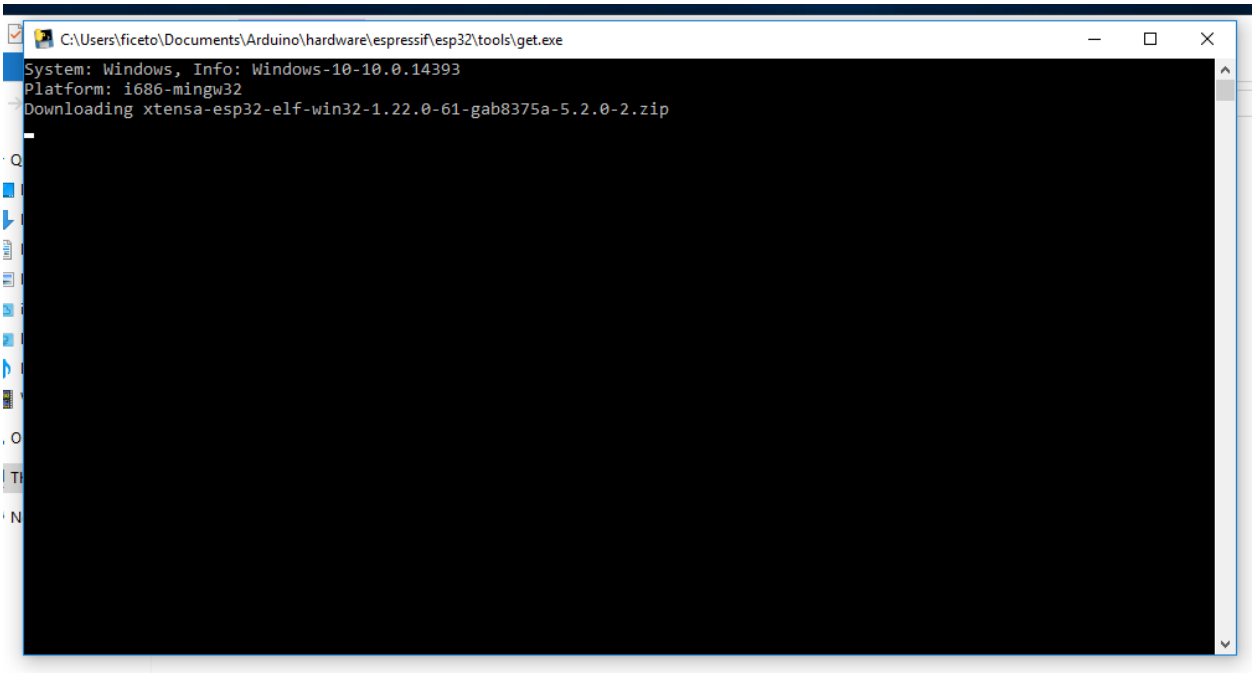
Linux

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


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


C:\Users\ficeto\Documents\Arduino\hardware\espressif\esp32\tools\get.exe
System: Windows, Info: Windows-10-10.0.14393
Platform: i686-mingw32
Downloading xtensa-esp32-elf-win32-1.22.0-61-gab8375a-5.2.0-2.zip

- 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 boardsto help you to save money and time.

One ESP32 to rule them all!

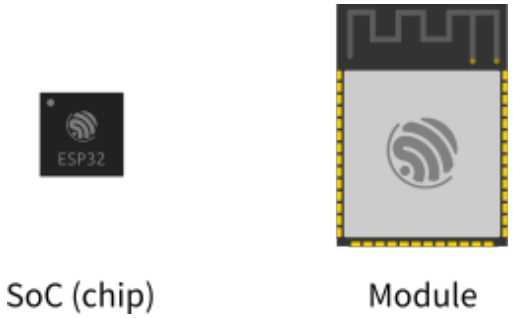
One important information that usually bring about some confusion is regarding the different models of the ESP32 SoC and modules.

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.

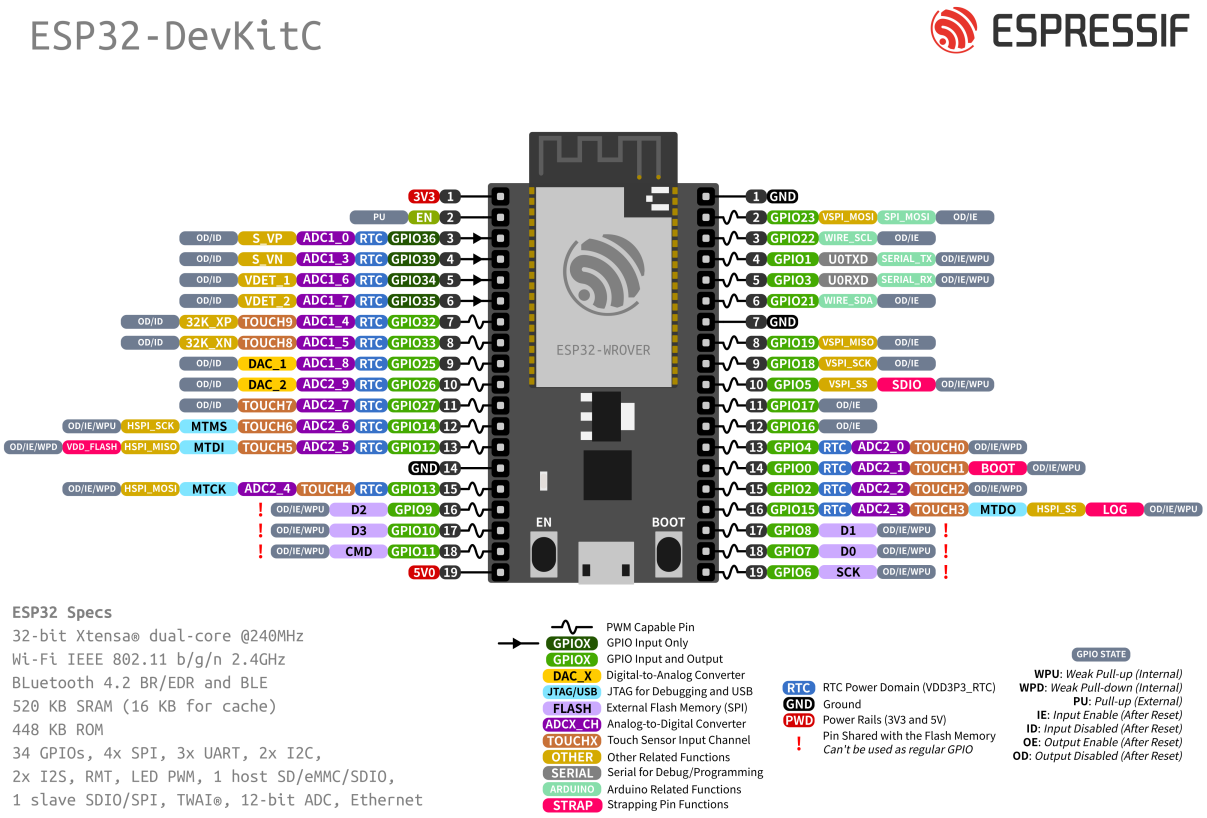
Pin Layout

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.

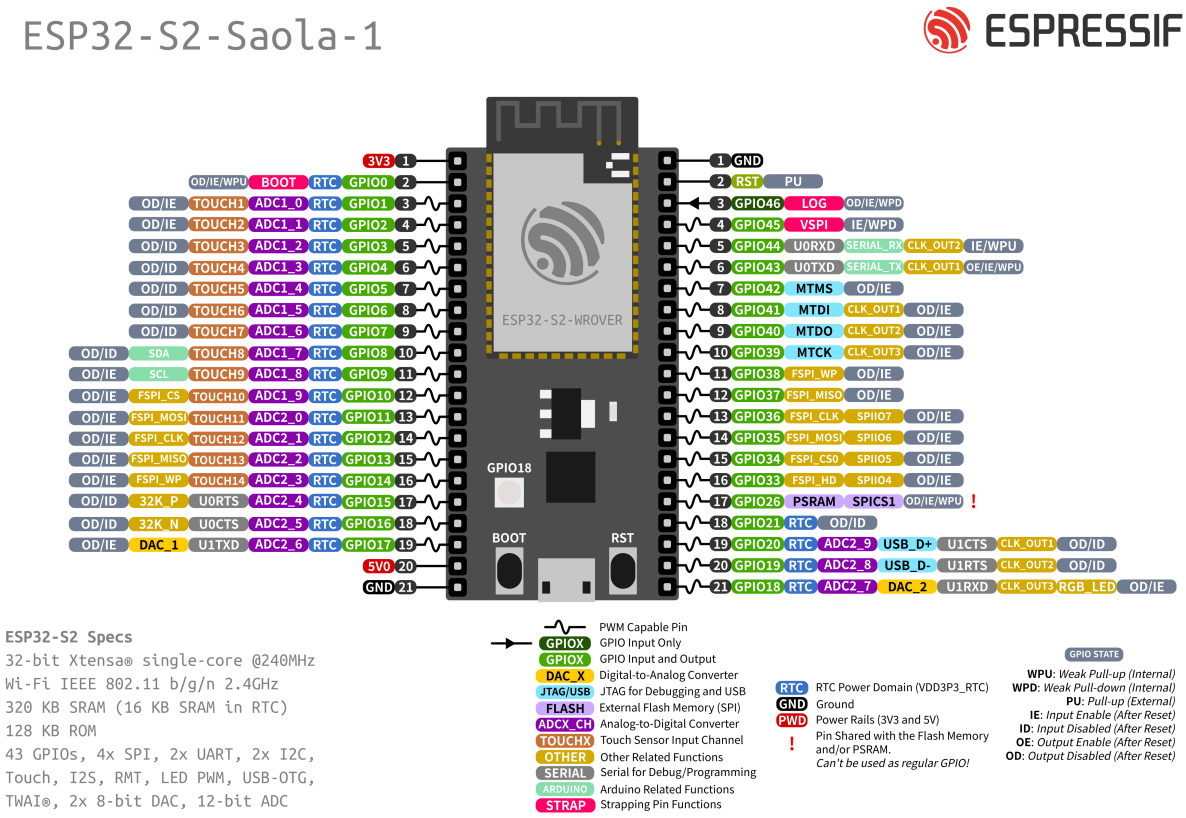
Pin Layout

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

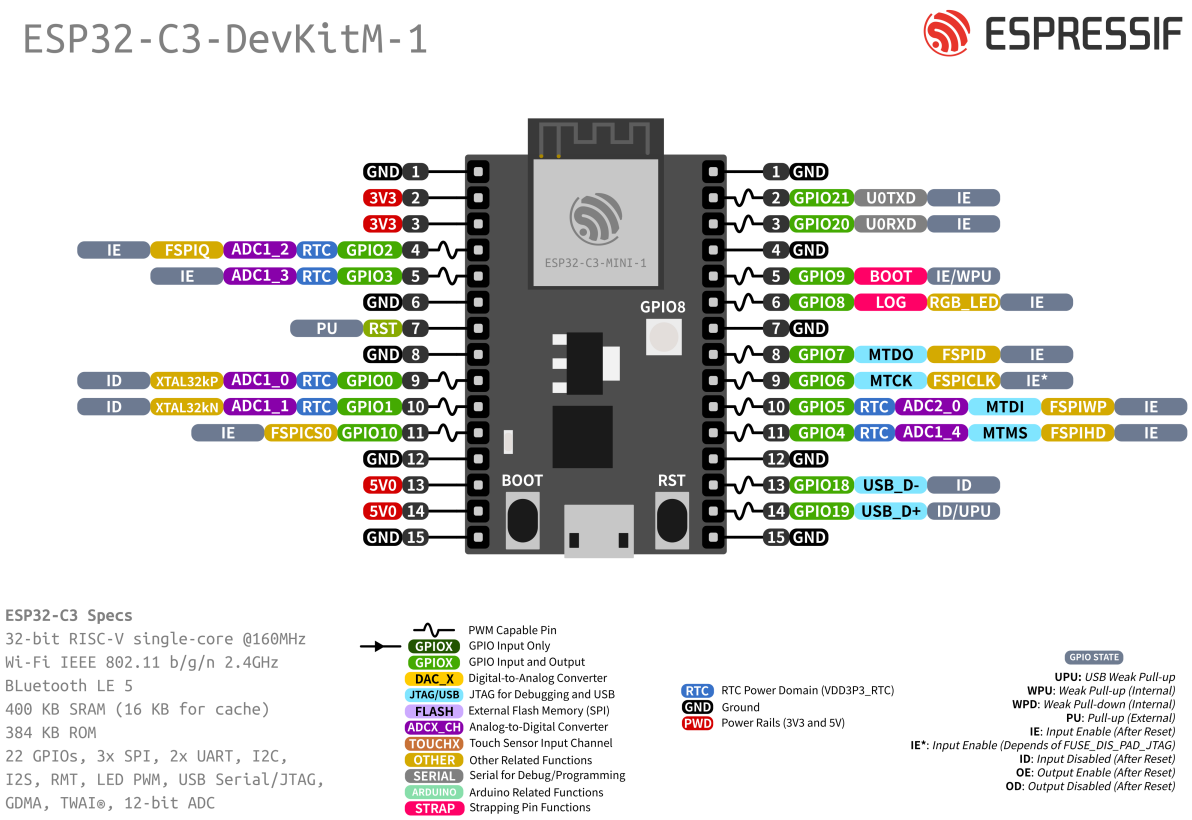
J3

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

ESP32-S2

ESP32-C3

ESP32-S3

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

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

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

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 Bluetooth API

2.3.3 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.

2.3.4 Deep Sleep

2.3.5 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.

Example

Resources

- ESP-NOW (User Guide)

2.3.6 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);
    }
}
```

2.3.7 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.

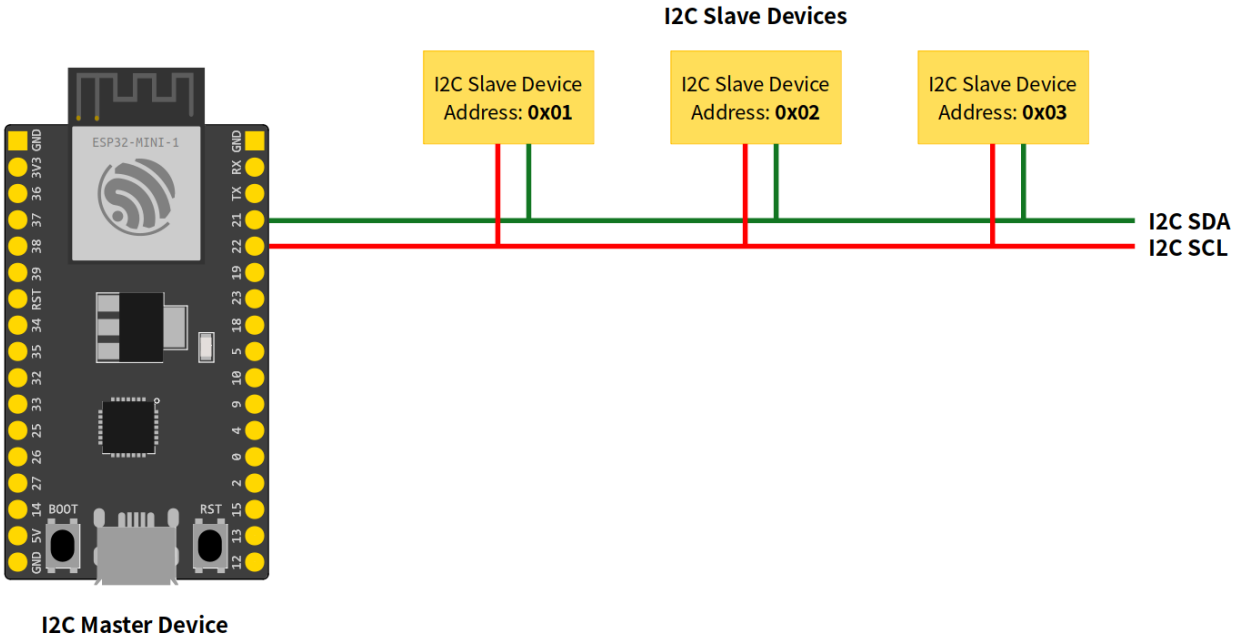
This serial communication is considered as a low-speed bus, and multiple devices can be connected on the same two-wires bus, each with a unique 7-bits address (up to 128 devices). These two wires are called SDA (serial data line) and SCL (serial clock line).

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.

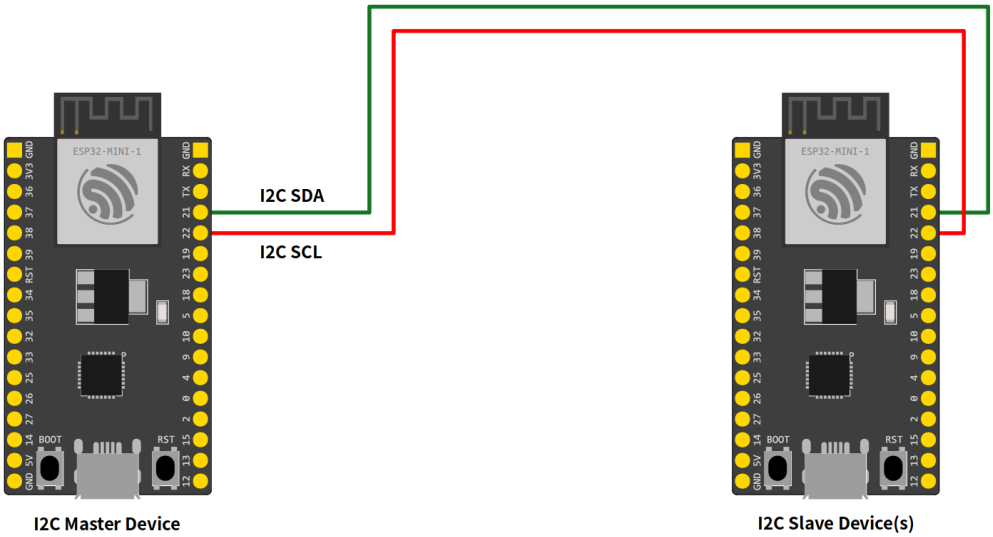
I2C Modes

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
  error = Wire.requestFrom(I2C_DEV_ADDR, 16);
  Serial.printf("requestFrom: %u\n", error);
  if(error){
    uint8_t temp[error];
    Wire.readBytes(temp, error);
    log_print_buf(temp, error);
  }
}
```

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() {

}
```

2.3.8 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.

```
double ledcSetup(uint8_t channel, double 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.

```
double 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.

```
double ledcWriteTone(uint8_t chan, double 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.

```
double 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.

```
double ledcChangeFrequency(uint8_t chan, double 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).

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 13 bit precission for LEDC timer
#define LEDC_TIMER_13_BIT  13

// 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, 8191 from 2 ^ 13 - 1
  uint32_t duty = (8191 / 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_13_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;
```

(continues on next page)

(continued from previous page)

```
  // 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.

// 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);
}
```

(continues on next page)

(continued from previous page)

```
// 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
 }

}

// 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;
```

(continues on next page)

(continued from previous page)

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

2.3.9 RainMaker

2.3.10 Reset Reason

2.3.11 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.

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

- `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);
}
```

2.3.12 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.
- `resolution` select timer resolution.
  - range is 1-14 bits (1-20 bits for ESP32).

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
}
```

2.3.13 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_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

```
#include "USB.h"

#if ARDUINO_USB_CDC_ON_BOOT
#define HWSerial Serial0
#define USBSerial Serial
#else
#define HWSerial Serial
USBCDC USBSerial;
#endif
```

(continues on next page)

(continued from previous page)

```
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;

      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;

      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);
  }
}
```

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

```
#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;

      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;
```

(continues on next page)

(continued from previous page)

```
    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
}
```

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.

2.3.14 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 (WPA, WPA2, WEP, etc.)
- Scanning for access points

Working as AP

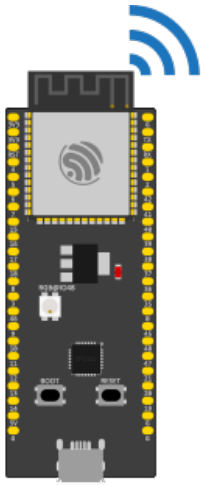
In this mode, the ESP32 is configured as an Access Point (AP) and it's capable of receiving incoming connections from other devices (stations) by providing a Wi-Fi network.

This mode can be used for serving an HTTP or HTTPS server inside the ESP32, for example.

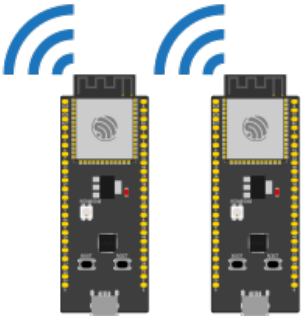
Working as STA

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.

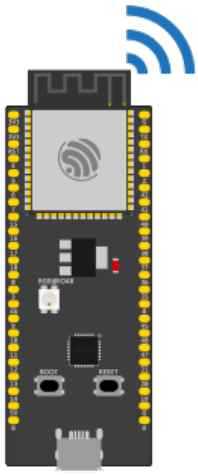


ESP32 as Wi-Fi Access Point (AP)

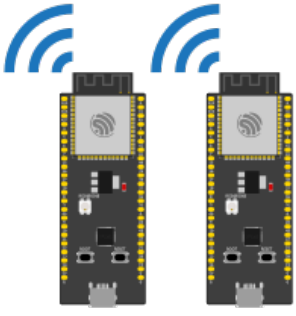


ESP32' as Wi-Fi Station (STA)

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 used to set the automatic connection.

```
bool setAutoConnect(bool autoConnect);
```

Where:

- `bool autoConnect` is set to `true` to enable this option.

getAutoConnect

Function used to get the automatic connection setting value.

```
bool getAutoConnect();
```

The function will return `true` if the setting is enabled.

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. .. code-block:: arduino

bool getAutoReconnect();

The function will return `true` if this setting is enabled.

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

```
/*
 *  This sketch sends data via HTTP GET requests to data.sparkfun.com service.
 *
 *  You need to get streamId and privateKey at data.sparkfun.com and paste them
 *  below. Or just customize this script to talk to other HTTP servers.
 *
 */

#include <WiFi.h>

const char* ssid     = "your-ssid";
const char* password = "your-password";

const char* host = "data.sparkfun.com";
const char* streamId   = "....................";
const char* privateKey = "....................";

void setup()
{
    Serial.begin(115200);
    delay(10);

    // We start by connecting to a WiFi network

    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());
}

int value = 0;

void loop()
{
    delay(5000);
    ++value;

    Serial.print("connecting to ");
```

(continues on next page)

(continued from previous page)

```
    Serial.println(host);

    // Use WiFiClient class to create TCP connections
    WiFiClient client;
    const int httpPort = 80;
    if (!client.connect(host, httpPort)) {
        Serial.println("connection failed");
        return;
    }

    // We now create a URI for the request
    String url = "/input/";
    url += streamId;
    url += "?private_key=";
    url += privateKey;
    url += "&value=";
    url += value;

    Serial.print("Requesting URL: ");
    Serial.println(url);

    // This will send the request to the server
    client.print(String("GET ") + url + " HTTP/1.1\r\n" +
                 "Host: " + host + "\r\n" +
                 "Connection: close\r\n\r\n");
    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.println();
    Serial.println("closing connection");
}
```

References

CHAPTER

THREE

GUIDES

3.1 Arduino IDE Tools Menu

3.1.1 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.

3.1.2 Arduino IDE

The Arduino IDE is widely used for ESP32 on Arduino development and offers a wide variety of configurations.

3.1.3 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.

3.1.4 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.

Port

This option is used to select the serial port to be used on the flashing and monitor.

3.1.5 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.

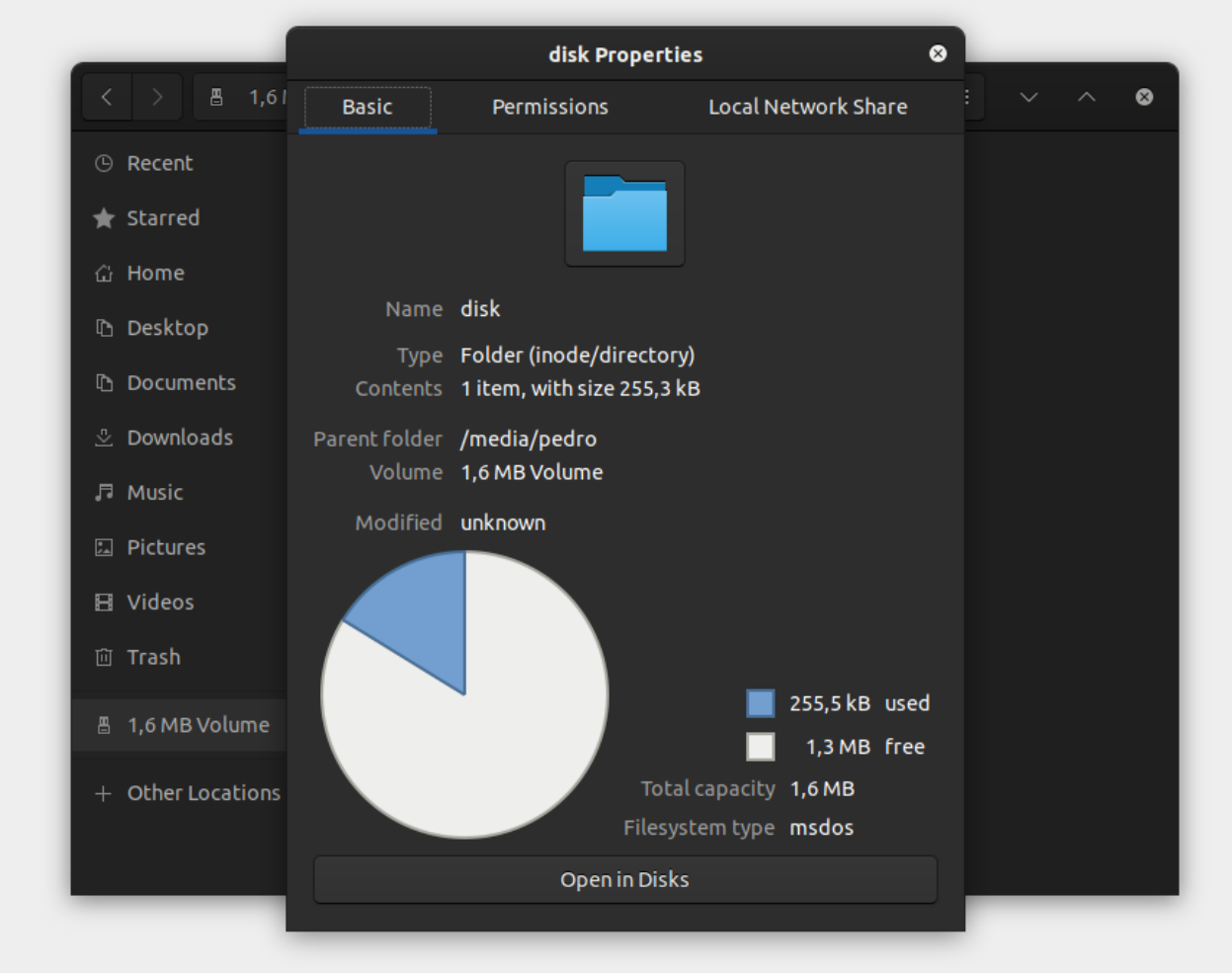
This option can be used as well for debugging via the `Serial Monitor` using **CDC** instead of the **UART0**.

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.

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

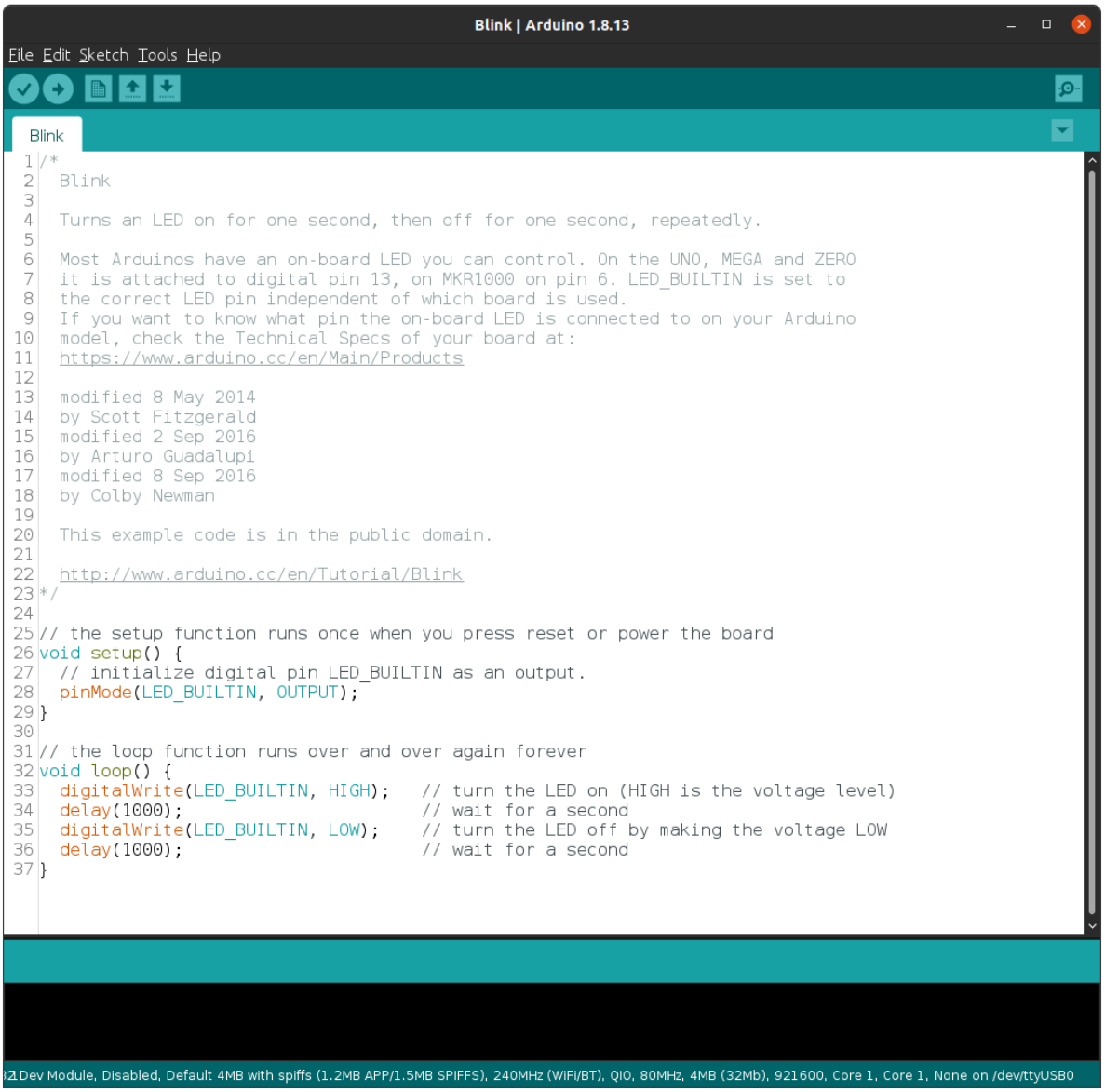
Listing 1: Blink.ino

```
/*
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 DFU

Note: DFU is only supported by the ESP32-S2.

4.4 GPIO Matrix and Pin Mux

4.4.1 Introduction

This is a basic introduction on 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.

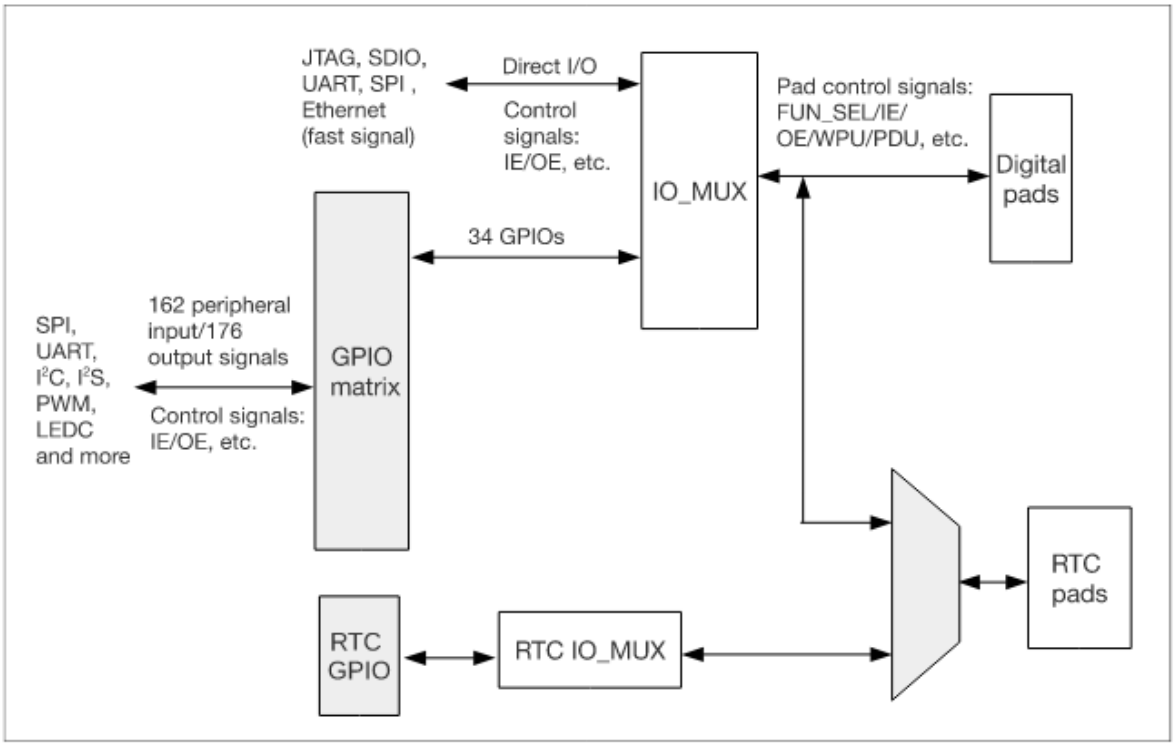
For example.

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

void setup()
{
    Wire.setPins(sda_pin, scl_pin); // Set the I2C pins before begin
```

(continues on next page)

(continued from previous page)

```
    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

CHAPTER

FIVE

ADVANCED UTILITIES

5.1 Library Builder

5.1.1 How to Use Library Builder

Espressif has provided a tool to simplify building your own compiled libraries for use in Arduino IDE (or your favorite IDE). To generate custom libraries, follow these steps:

- Step 1 - Clone the ESP32 Arduino lib builder:

```
git clone https://github.com/espressif/esp32-arduino-lib-builder
```

- Step 2 - Go to the `esp32-arduino-lib-builder` folder:

```
cd esp32-arduino-lib-builder
```

- Step 3 - Run the `update-components` script:

```
./tools/update-components.sh`
```

- Step 4 - Run `install-esp-idf` installation script (if you already have an `$IDF_PATH` defined, it will use your local copy of the repository):

```
./tools/install-esp-idf.sh
```

- Step 5 - Copy the configuration (recommended) or directly edit sdkconfig using `idf.py menuconfig`:

```
cp sdkconfig.esp32s2 sdkconfig
```

- Step 6 - Build:

```
idf.py build
```

The script automates the process of building Arduino as an ESP-IDF component. Once it is complete, you can cherry pick the needed libraries from `out/tools/sdk/lib`, or run `tools/copy-to-arduino.sh` to copy the entire built system. `tools/config.sh` contains a number of variables that control the process, particularly the `$IDF_BRANCH` variable. You can adjust this to try building against newer versions, but there are absolutely no guarantees that any components will work or even successfully compile against a newer IDF.

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
```

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");
  }
```

(continues on next page)

(continued from previous page)

```
  // 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)

You might notice that Arduino-esp32's *delay()* function will only work in multiples of 10ms. That is because, by default, esp-idf handles task events 100 times per second. To fix that behavior, you need to set FreeRTOS tick rate 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:

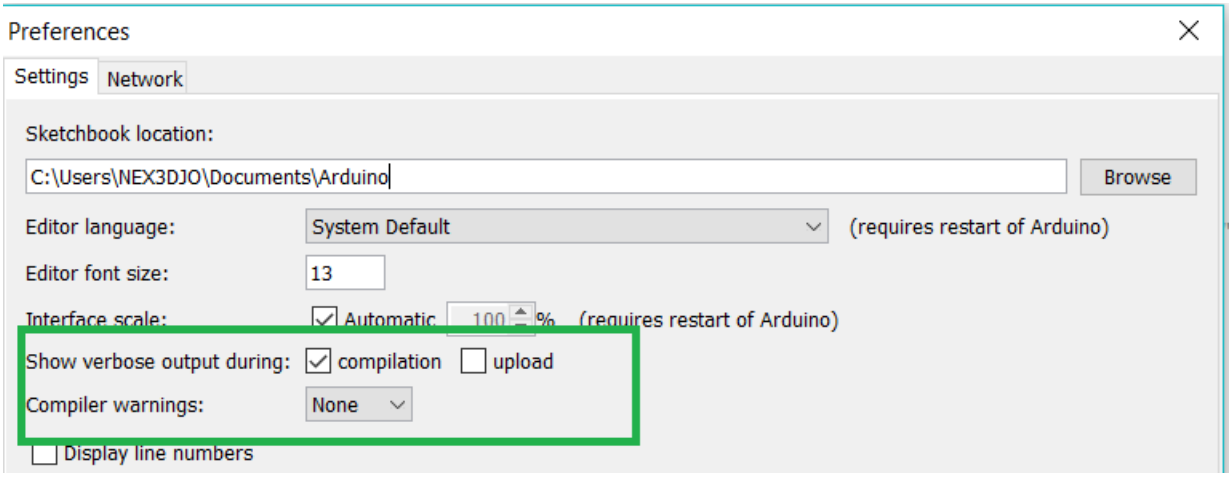
- Example sketch `OTAWebUpdater.ino`.
- ESP32 Board.

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

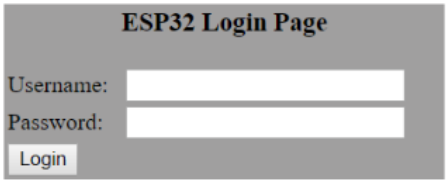
Prepare the sketch and configuration for initial upload with a serial port - Start Arduino IDE and load sketch OTAWebUpdater.ino available under File > Examples > OTAWebUpdater.ino - Update ssid and pass in the sketch so the module can join your Wi-Fi network - Open File > Preferences, look for "Show verbose output during:" and check out "compilation" option

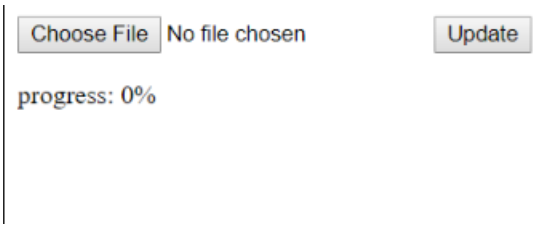


- Upload sketch (Ctrl+U)
- Now open web browser and enter the url, i.e. http://esp32.local. Once entered, browser should display a form
- username = admin
- password = admin

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.*

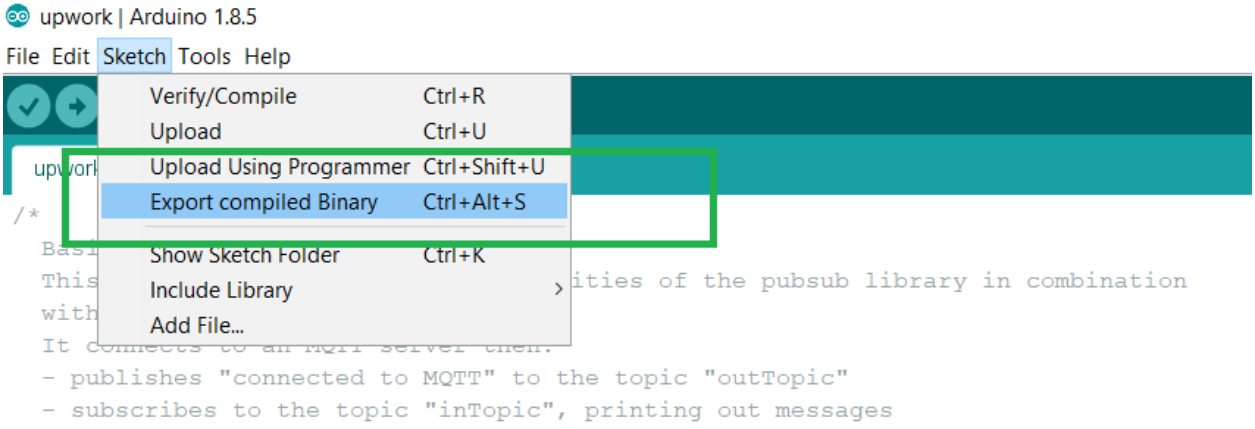
Now click on the Login button and browser will display an upload form





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



- 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

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.

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, we unfortunately cannot accept it.
- Is the code adequately commented for people to 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.
- 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 onto the public github repository.

8.4 Legal Part

Before a contribution can be accepted, you will need to sign our contributor-agreement. You will be prompted for this automatically as part of the Pull Request process.