Getting Started with the ESP32 Development Board

New to ESP32? Start here! The ESP32 is a series of low-cost and low-power System on a Chip (SoC) microcontrollers developed by Espressif that include Wi-Fi and Bluetooth wireless capabilities and dual-core processor. If you're familiar with the ESP8266, the ESP32 is its successor, loaded with lots of new features.

Updated 29 September 2023

New to the ESP32? You're in the right place. This guide contains all the information you need to get started with the ESP32. Learn what is an ESP32, how to select an ESP32 board, how to get your first program working, and much more. Here's what we'll cover in this guide:

Table of Contents

Introducing the ESP32

First, to get started, **what is an ESP32**? The ESP32 is a series of chip microcontrollers developed by Espressif.

Why are they so popular? Mainly because of the following features:

- **Low-cost**: you can get an ESP32 starting at $6, which makes it easily accessible to the general public;
- **Low-power**: the ESP32 consumes very little power compared with other microcontrollers, and it supports low-power mode states like deep sleep to save power;
- **Wi-Fi capabilities**: the ESP32 can easily connect to a Wi-Fi network to connect to the internet (station mode), or create its own Wi-Fi wireless network (access point mode) so other devices can connect to it—this is essential for IoT and Home Automation projects—you can have multiple devices communicating with each other using their Wi-Fi capabilities;
- **Bluetooth**: the ESP32 supports Bluetooth classic and Bluetooth Low Energy (BLE)—which is useful for a wide variety of IoT applications;
- Dual-core: most ESP32 are dual-core— they come with 2 Xtensa 32-bit LX6 microprocessors: core 0 and core 1.

- Rich peripheral input/output interface—the ESP32 supports a wide variety of input (read data from the outside world) and output (to send commands/signals to the outside world) peripherals like capacitive touch, ADCs, DACs, UART, SPI, I2C, PWM, and much more.

- Compatible with the Arduino "programming language": those that are already familiar with programming the Arduino board, you'll be happy to know that they can program the ESP32 in the Arduino style.

- Compatible with MicroPython: you can program the ESP32 with MicroPython firmware, which is a re-implementation of Python 3 targeted for microcontrollers and embedded systems.

ESP32 Specifications

If you want to get a bit more technical and specific, you can take a look at the following detailed specifications of the ESP32 (source: http://esp32.net/)—for more details, check the datasheet):

ESP32 module: ESP-WROOM-32

- **Wireless connectivityWiFi:** 150.0 Mbps data rate with HT40
  - **Bluetooth:** BLE (Bluetooth Low Energy) and Bluetooth Classic
  - **Processor:** Tensilica Xtensa Dual-Core 32-bit LX6 microprocessor, running at 160 or 240 MHz

- **Memory**:
  - **ROM:** 448 KB (for booting and core functions)
  - **SRAM:** 520 KB (for data and instructions)

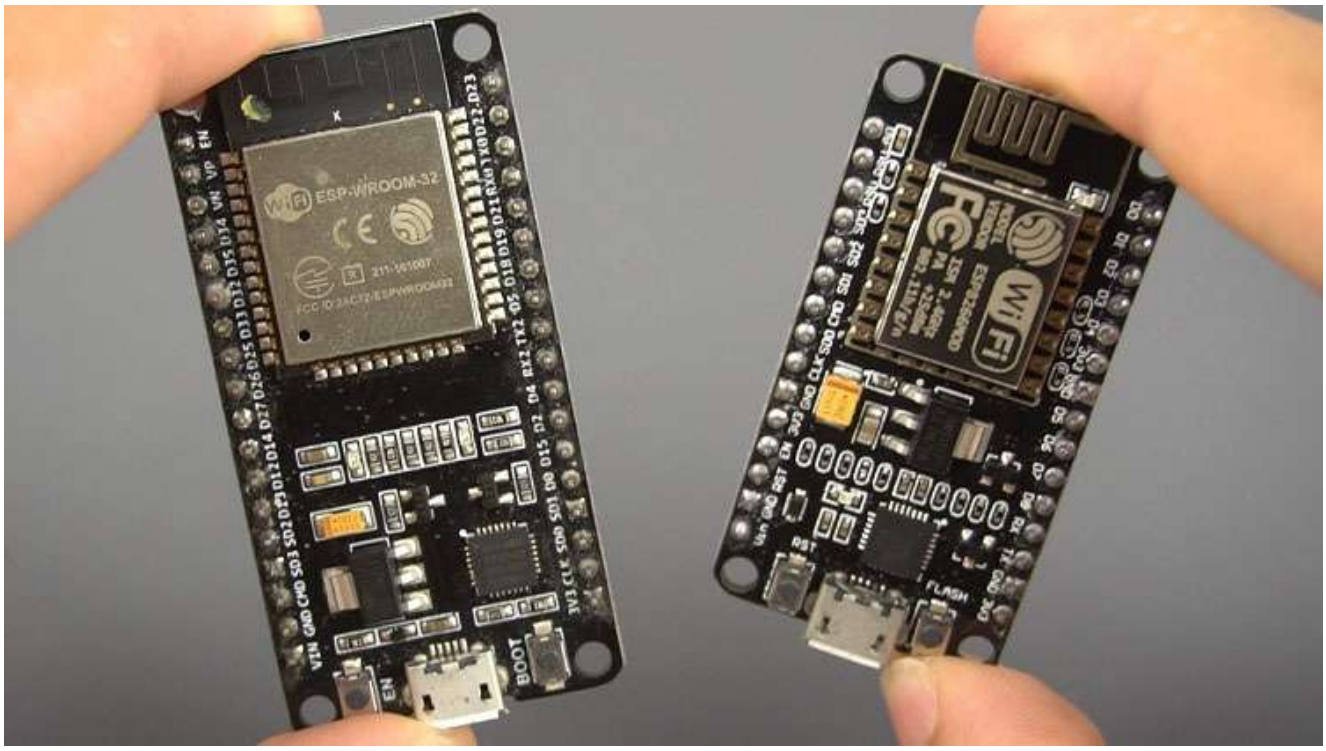
- **RTC fast SRAM**: 8 KB (for data storage and main CPU during RTC Boot from the deep-sleep mode)
- **RTC slow SRAM**: 8KB (for co-processor accessing during deep-sleep mode)
- **eFuse**: 1 Kbit (of which 256 bits are used for the system (MAC address and chip configuration) and the remaining 768 bits are reserved for customer applications, including Flash-Encryption and Chip-ID)
- **Embedded flash**: flash connected internally via IO16, IO17, SD_CMD, SD_CLK, SD_DATA_0 and SD_DATA_1 on ESP32-D2WD and ESP32-PICO-D4.
  - 0 MiB (ESP32-D0WDQ6, ESP32-D0WD, and ESP32-S0WD chips)
  - 2 MiB (ESP32-D2WD chip)
  - 4 MiB (ESP32-PICO-D4 SiP module)

- **Low Power:** ensures that you can still use ADC conversions, for example, during deep sleep.

- **Peripheral Input/Output:**
  - peripheral interface with DMA that includes capacitive touch
  - ADCs (Analog-to-Digital Converter)
  - DACs (Digital-to-Analog Converter)
  - I²C (Inter-Integrated Circuit)
  - UART (Universal Asynchronous Receiver/Transmitter)
  - SPI (Serial Peripheral Interface)
  - I²S (Integrated Interchip Sound)
  - RMII (Reduced Media-Independent Interface)
  - PWM (Pulse-Width Modulation)

- **Security:** hardware accelerators for AES and SSL/TLS

Main Differences Between ESP32 and ESP8266



Previously, we mentioned that the ESP32 is the ESP8266 successor. **What are the main differences between ESP32 and ESP8266 boards?**

The ESP32 adds an extra CPU core, faster Wi-Fi, more GPIOs, and supports Bluetooth 4.2 and Bluetooth low energy. Additionally, the ESP32 comes with touch-sensitive pins that can be used to wake up the ESP32 from deep sleep, and built-in hall effect sensor.

Both boards are cheap, but the ESP32 costs slightly more. While the ESP32 can cost around $6 to $12, the ESP8266 can cost $4 to $6 (but it really depends on where you get them and what model you're buying).

So, in summary:

- The ESP32 is faster than the ESP8266;
- The ESP32 comes with more GPIOs with multiple functions;
- The ESP32 supports analog measurements on 18 channels (analog-enabled pins) versus just one 10-bit ADC pin on the ESP8266;
- The ESP32 supports Bluetooth while the ESP8266 doesn't;
- The ESP32 is dual-core (most models), and the ESP8266 is single core;
- The ESP32 is a bit more expensive than the ESP8266.

For a more detailed analysis of the differences between those boards, we recommend reading the following article: ESP32 vs ESP8266 – Pros and Cons.

ESP32 Development Boards

ESP32 refers to the bare ESP32 chip. However, the “ESP32” term is also used to refer to ESP32 development boards. Using ESP32 bare chips is not easy or practical, especially when learning, testing, and prototyping. Most of the time, you’ll want to use an ESP32 development board.

These development boards come with all the needed circuitry to power and program the chip, connect it to your computer, pins to connect peripherals, built-in power and control LEDs, an antenna for wi-fi signal, and other useful features. Others even come with extra hardware like specific sensors or modules, displays, or a camera in the case of the ESP32-CAM.

How to Choose an ESP32 Development Board?

Once you start searching for ESP32 boards online, you’ll find there is a wide variety of boards from different vendors. While they all work in a similar way, some boards may be more suitable for some projects than others. When looking for an ESP32 development board there are several aspects you need to take into account:

- **USB-to-UART interface and voltage regulator circuit**. Most full-featured development boards have these two features. This is important to easily connect the ESP32 to your computer to upload code and apply power.

- **BOOT and RESET/EN buttons** to put the board in flashing mode or reset (restart) the board. Some boards don't have the BOOT button. Usually, these boards go into flashing mode automatically.

- **Pin configuration and the number of pins.** To properly use the ESP32 in your projects, you need to have access to the board pinout (like a map that shows which pin corresponds to which GPIO and its features). So make sure you have access to the pinout of the board you're getting. Otherwise, you may end up using the ESP32 incorrectly.

- **Antenna connector**. Most boards come with an onboard antenna for Wi-Fi signal. Some boards come with an antenna connector to optionally connect an external antenna. Adding an external antenna increases your Wi-Fi range.

- **Battery connector**. If you want to power your ESP32 using batteries, there are development boards that come with connectors for LiPo batteries—this can be handier. You can also power a "regular" ESP32 with batteries through the power pins.

- **Extra hardware features**. There are ESP32 development boards with extra hardware features. For example, some may come with a built-in OLED display, a LoRa module, a SIM800 module (for GSM and GPRS), a battery holder, a camera, or others.

What is the best ESP32 development board for beginners?

For beginners, we recommend an ESP32 board with a vast selection of available GPIOs, and without any extra hardware features. It's also important that it comes with voltage regular and USB input for power and upload code.

In most of our ESP32 projects, we use the ESP32 DEVKIT DOIT board, and that's the one we recommend for beginners. There are different versions of this board with a different number of available pins (30, 36, and 38)—all boards work in a similar way.

Where to Buy?

You can check the following link to find the ESP32 DEVKIT DOIT board in different stores:

- ESP32 DEVKIT DOIT board

Other similar boards with the features mentioned previously may also be a good option like the Adafruit ESP32 Feather, Sparkfun ESP32 Thing, NodeMCU-32S, Wemos LoLin32, etc.

ESP32 DEVKIT DOIT

In this article, we'll be using the ESP32 DEVKIT DOIT board as a reference. If you have a different board, don't worry. The information on this page is also compatible

with other ESP32 development boards.

The picture below shows the ESP32 DEVKIT DOIT V1 board, version with 36 GPIO pins.

Specifications – ESP32 DEVKIT V1 DOIT

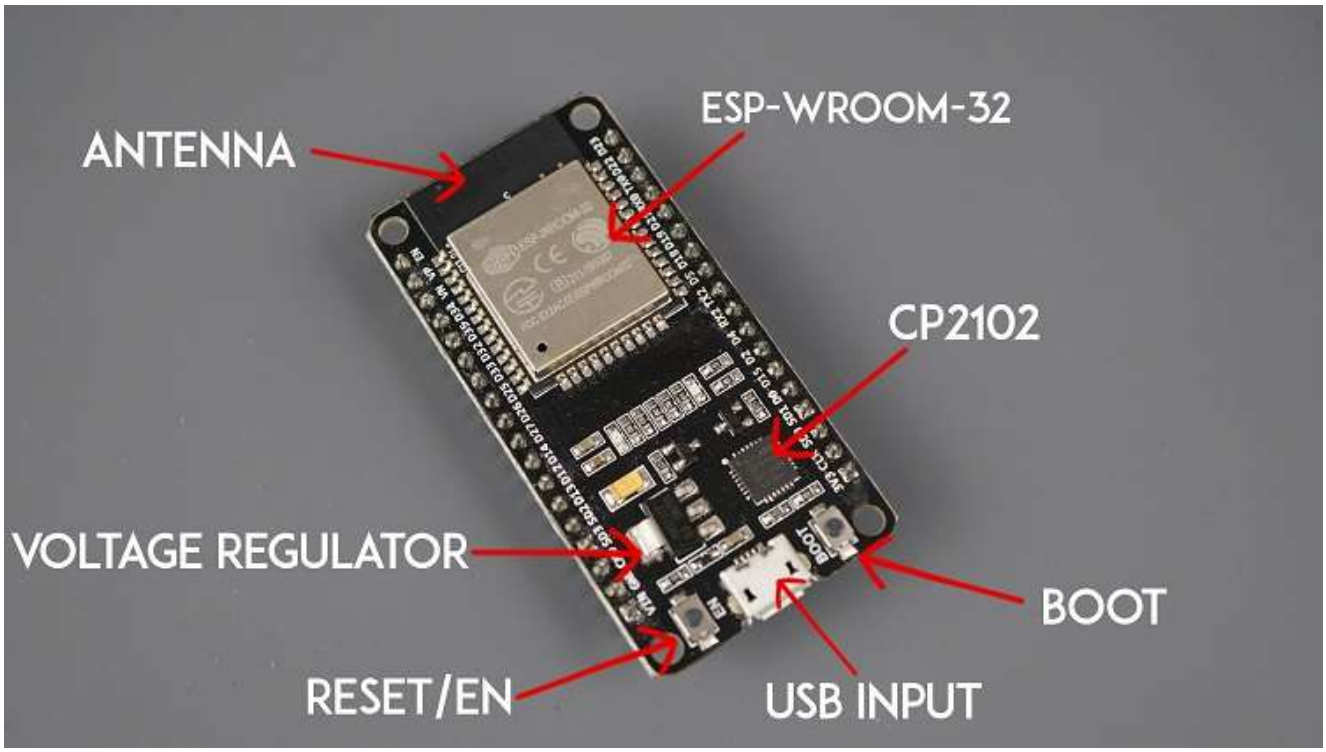
The following table shows a summary of the ESP32 DEVKIT V1 DOIT board features and specifications:

Number of cores	2 (dual core)
Wi-Fi	2.4 GHz up to 150 Mbits/s
Bluetooth	BLE (Bluetooth Low Energy) and legacy Bluetooth
Architecture	32 bits
Clock frequency	Up to 240 MHz
RAM	512 KB
Pins	30, 36, or 38 (depending on the model)

Peripherals	Capacitive touch, ADC (analog to digital converter), DAC (digital to analog converter), I2C (Inter-Integrated Circuit), UART (universal asynchronous receiver/transmitter), CAN 2.0 (Controller Area Networkr), SPI (Serial Peripheral Interface), I2S (Integrated Inter-IC Sound), RMII (Reduced Media-Independent Interface), PWM (pulse width modulation), and more.
Built-in buttons	RESET and BOOT buttons
Built-in LEDs	built-in blue LED connected to GPIO2; built-in red LED that shows the board is being powered
USB to UART bridge	CP2102

This particular ESP32 board comes with 36 pins, 18 on each side. The number of available GPIOs depends on your board model.

To learn more about the ESP32 GPIOs, read our GPIO reference guide: ESP32 Pinout Reference: Which GPIO pins should you use?

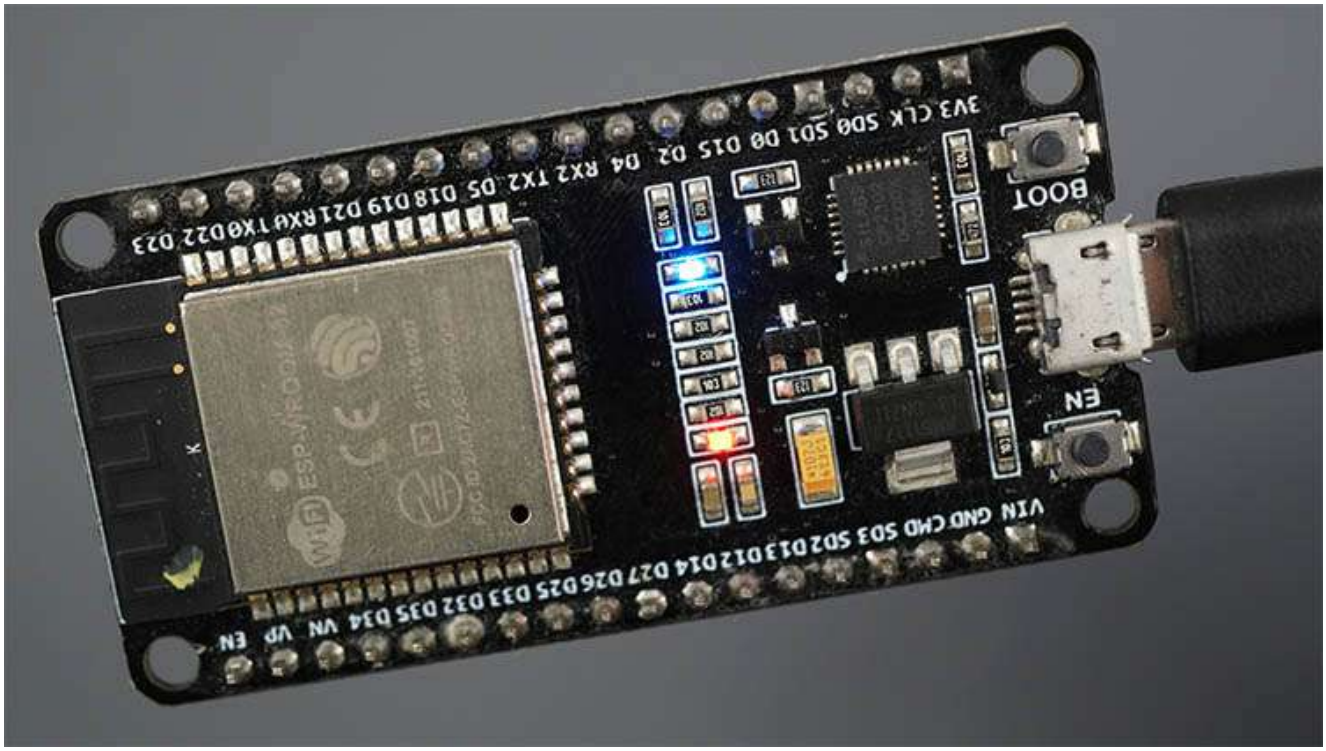


It comes with a microUSB interface that you can use to connect the board to your computer to upload code or apply power.

It uses the CP2102 chip (USB to UART) to communicate with your computer via a COM port using a serial interface. Another popular chip is the CH340. Check what's the USB to UART chip converter on your board because you'll need to install the required drivers so that your computer can communicate with the board (more information about this later in this guide).

This board also comes with a RESET button (may be labeled EN) to restart the board and a BOOT button to put the board in flashing mode (available to receive code). Note that some boards may not have a BOOT button.

It also comes with a built-in blue LED that is internally connected to GPIO 2. This LED is useful for debugging to give some sort of visual physical output. There's also a red LED that lights up when you provide power to the board.



ESP32 GPIOs Pinout Guide

The ESP32 chip comes with 48 pins with multiple functions. Not all pins are exposed in all ESP32 development boards, and some pins should not be used. The ESP32 DEVKIT V1 DOIT board usually comes with 36 exposed GPIOs that you can use to connect peripherals.

Power Pins

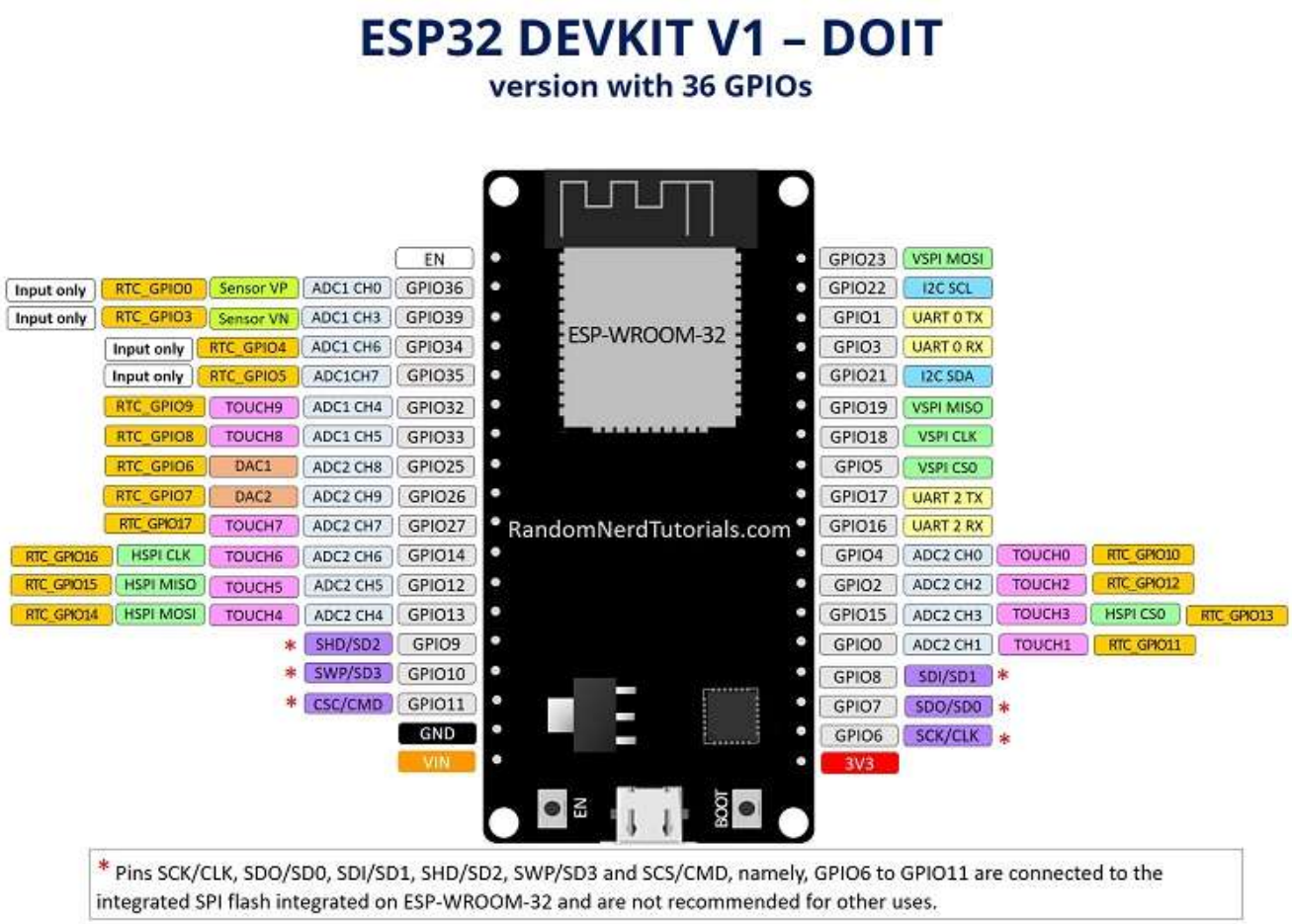
Usually, all boards come with power pins: 3V3, GND, and VIN. You can use these pins to power the board (if you're not providing power through the USB port), or to get power for other peripherals (if you're powering the board using the USB port).

General Purpose Input Output Pins (GPIOS)

Almost all GPIOs have a number assigned and that's how you should refer to them—by their number.

With the ESP32 you can decide which pins are UART, I2C, or SPI – you just need to set that on the code. This is possible due to the ESP32 chip's multiplexing feature that allows to assign multiple functions to the same pin.

If you don't set them on the code, the pins will be configured by default as shown in the figure below (the pin location can change depending on the manufacturer). Additionally, there are pins with specific features that make them suitable or not for a particular project.



We have a detailed guide dedicated to the ESP32 GPIOs that we recommend you read: ESP32 Pinout Reference Guide. It shows how to use the ESP32 GPIOs and explains what are the best GPIOs to use depending on your project.

The placement of the GPIOs might be different depending on your board model. However, usually, each specific GPIO works in the same way regardless of the development board you're using (with some exceptions). For example, regardless of the board, usually GPIO5 is always the VSPI CS0 pin, GPIO 23 always corresponds to VSPI MOSI for SPI communication, etc.

How to Program the ESP32?

The ESP32 can be programmed using different firmware and programming languages. You can use:

- Arduino C/C++ using the Arduino core for the ESP32
- Espressif IDF (IoT Development Framework)
- Micropython
- JavaScript
- LUA
- …

Our preferred method to program the ESP32 is with C/C++ "Arduino programming language". We also have some guides and tutorials using MicroPython firmware.

Throughout this guide, we'll cover programming the ESP32 using the Arduino core for the ESP32 board. If you prefer using MicroPython, please refer to this guide: Getting Started with MicroPython on ESP32.

Programming ESP32 with Arduino IDE

To program your boards, you need an IDE to write your code. For beginners, we recommend using Arduino IDE. While it's not the best IDE, it works well and is simple and intuitive to use for beginners. After getting familiar with Arduino IDE and you start creating more complex projects, you may find it useful to use VS Code with the Platformio extension instead.

If you're just getting started with the ESP32, start with Arduino IDE. At the time of writing this tutorial, **we recommend using the legacy version (1.8.19)** with the ESP32. While version 2 works well with Arduino, there are still some bugs and some features that are not supported yet for the ESP32.

Installing Arduino IDE

To run Arduino IDE, you need JAVA installed on your computer. If you don't, go to the following website to download and install the latest version: http://java.com/download.

Downloading Arduino IDE

To download the Arduino IDE, visit the following URL:

- https://www.arduino.cc/en/Main/Software

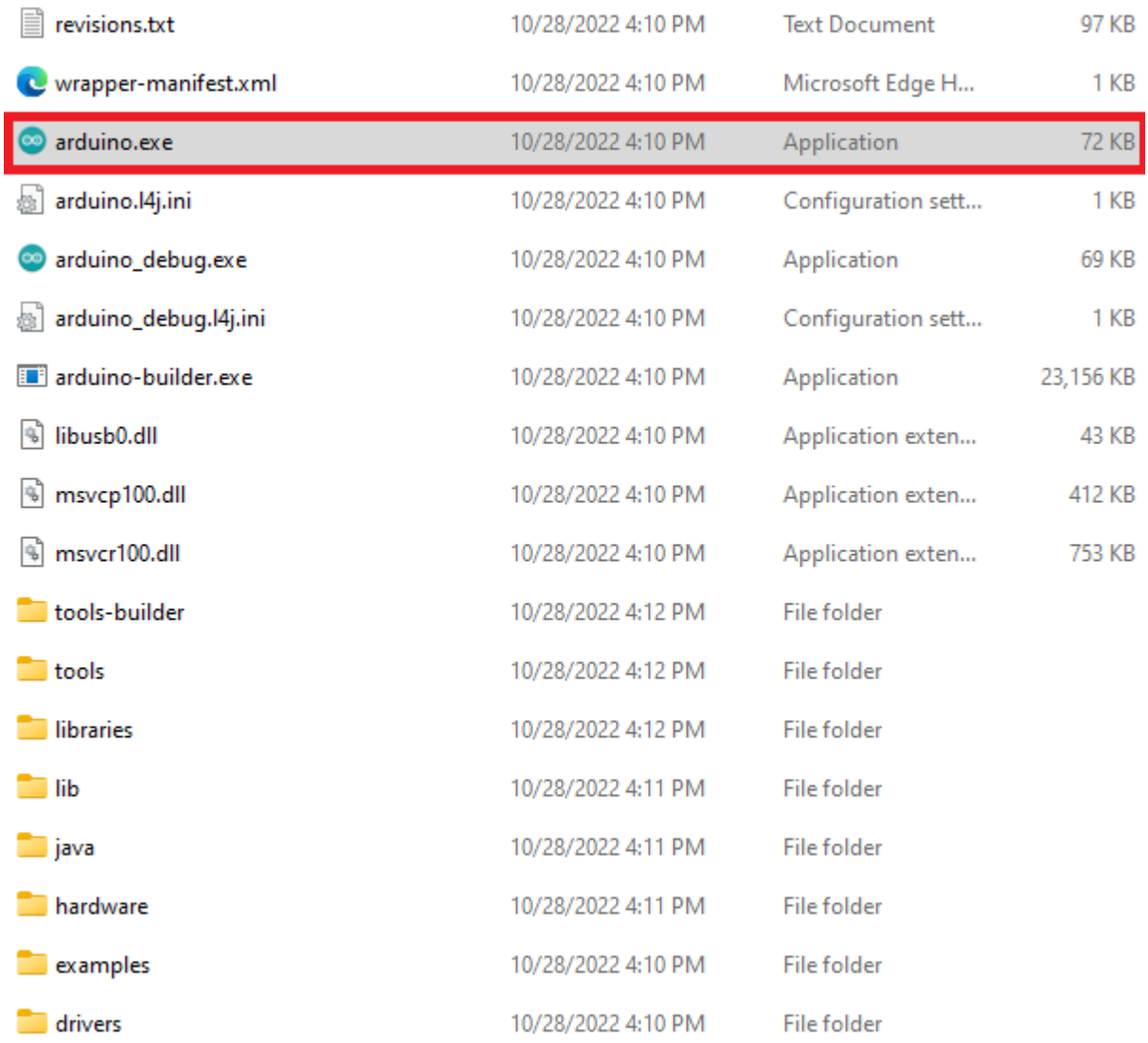
Don't install the 2.0 version. At the time of writing this tutorial, **we recommend using the legacy version (1.8.19)** with the ESP32. While version 2 works well with Arduino, there are still some bugs and some features that are not supported yet for the ESP32.

Scroll down until you find the legacy version section.

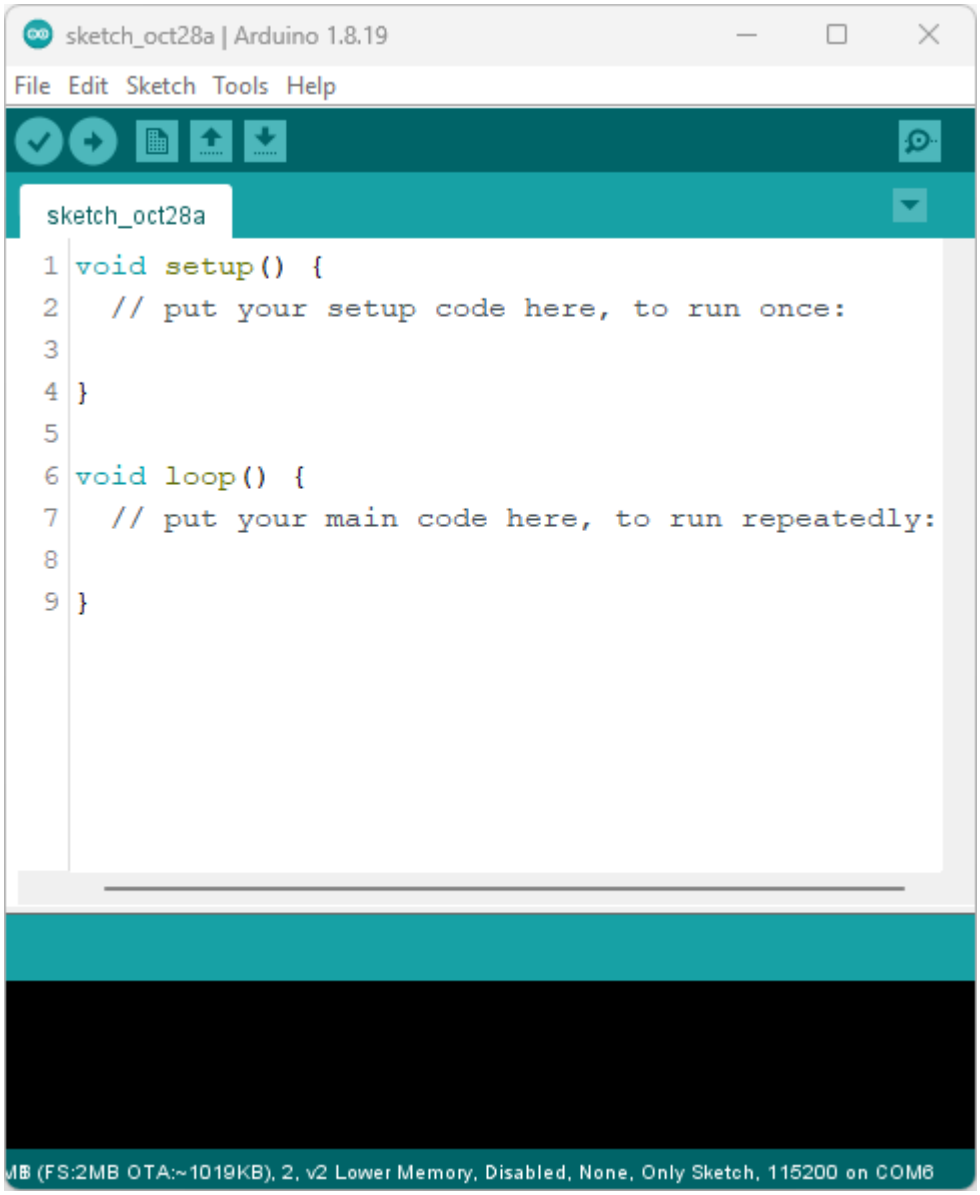
Select your operating system and download the software. For Windows, we recommend downloading the "**Windows ZIP file**".

Running Arduino IDE

Grab the folder you've just downloaded and unzip it. Run the executable file called *arduino.exe* (highlighted below).



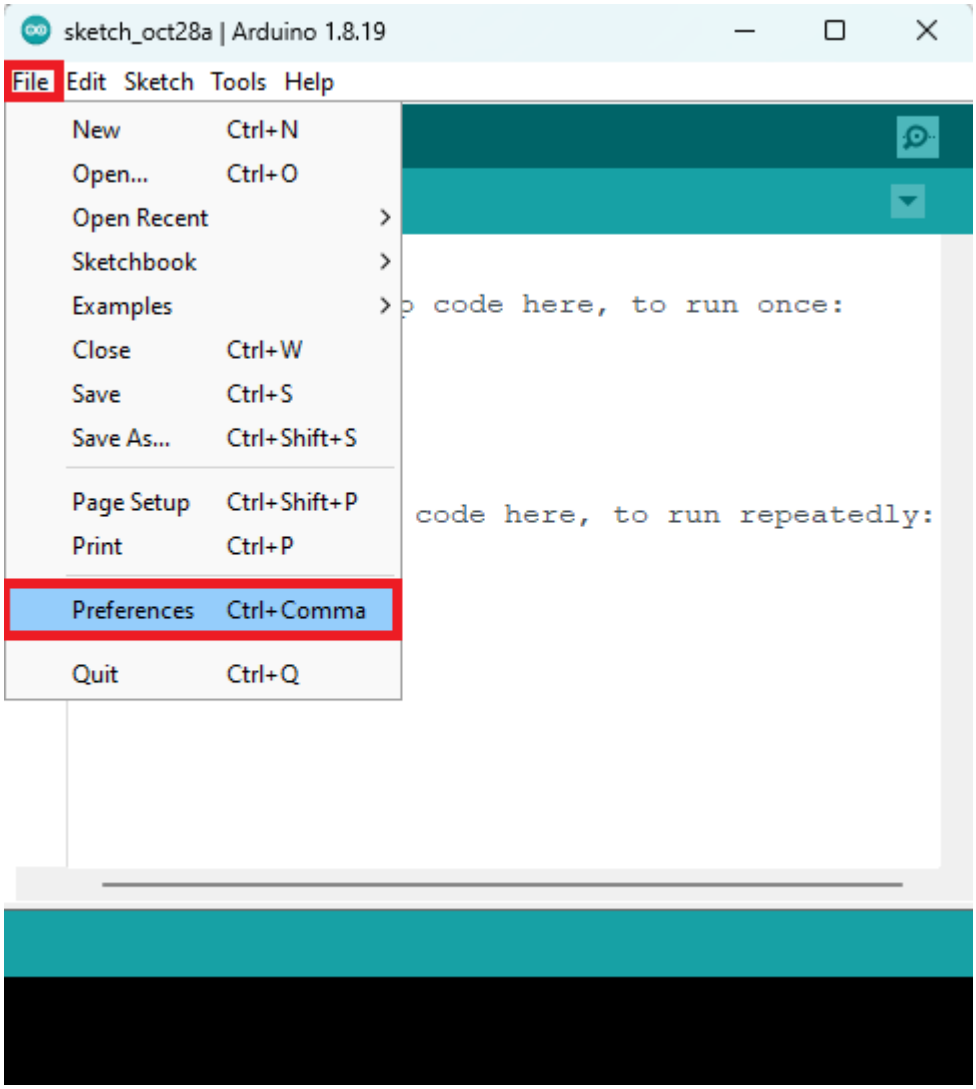
The Arduino IDE window should open.



Installing the ESP32 in Arduino IDE

To be able to program the ESP32 using Arduino IDE, you need to add support for the ESP32 boards. Follow the next steps:

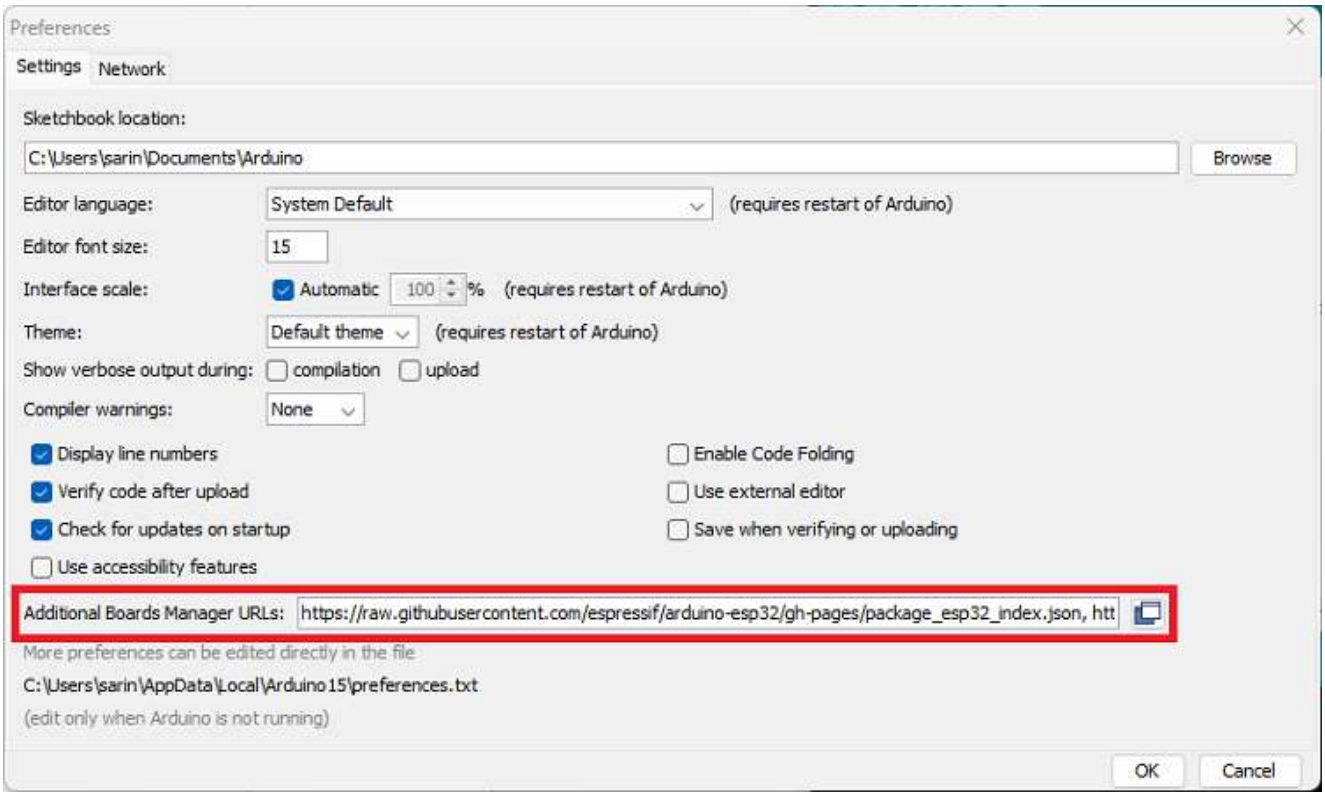
1. Go to **File** > **Preferences**.



2. Enter the following into the "*Additional Board Manager URLs*" field. This will add support for ESP32 and ESP8266 boards as well.

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json,
http://arduino.esp8266.com/stable/package_esp8266com_index.json
```

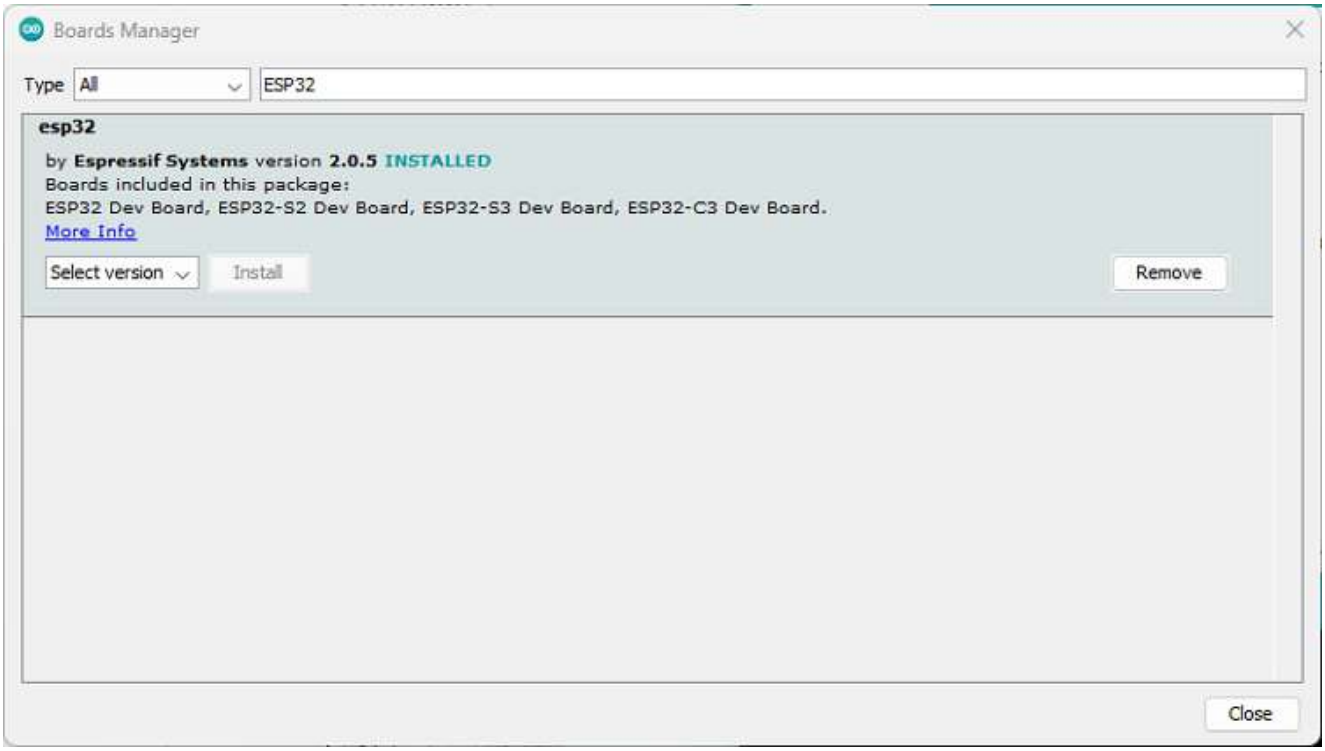
See the figure below. Then, click the "**OK**" button.



3. Open the **Boards Manager**. Go to **Tools** > **Board** >**Boards Manager…**

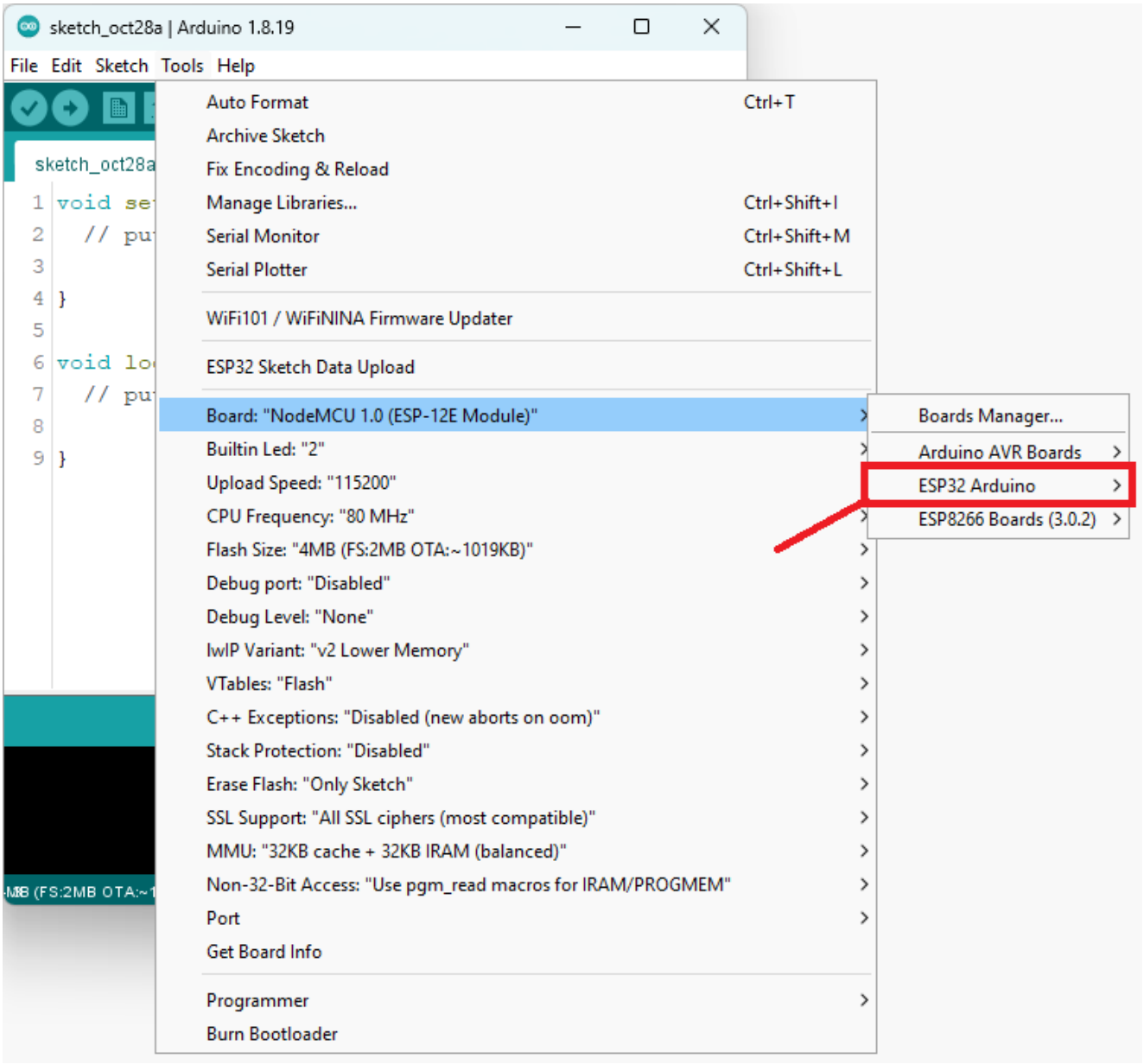
4. Search for **ESP32** and install the “**ESP32 by Espressif Systems**“:

That’s it. It will be installed after a few seconds.



After this, restart your Arduino IDE.

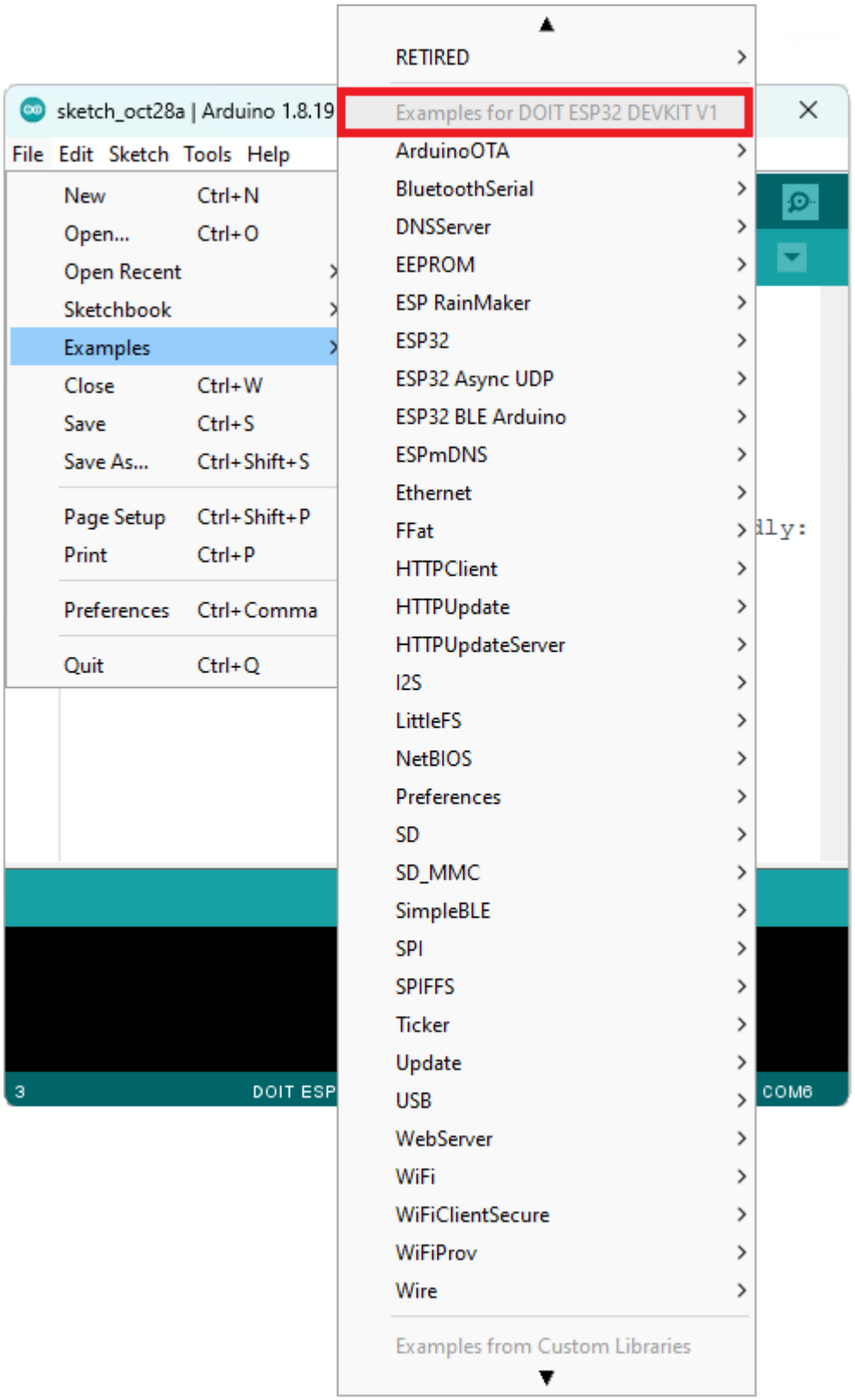
Then, go to **Tools** > **Board** and check that you have ESP32 boards available.



Now, you're ready to start programming your ESP32 using Arduino IDE.

ESP32 Examples

In your Arduino IDE, you can find multiple examples for the ESP32. First, make sure you have an ESP32 board selected in **Tools** > **Board**. Then, simply go to **File** > **Examples** and check out the examples under the ESP32 section.



Update the ESP32 Core in Arduino IDE

Once in a while, it's a good idea to check if you have the latest version of the ESP32 boards add-on installed.

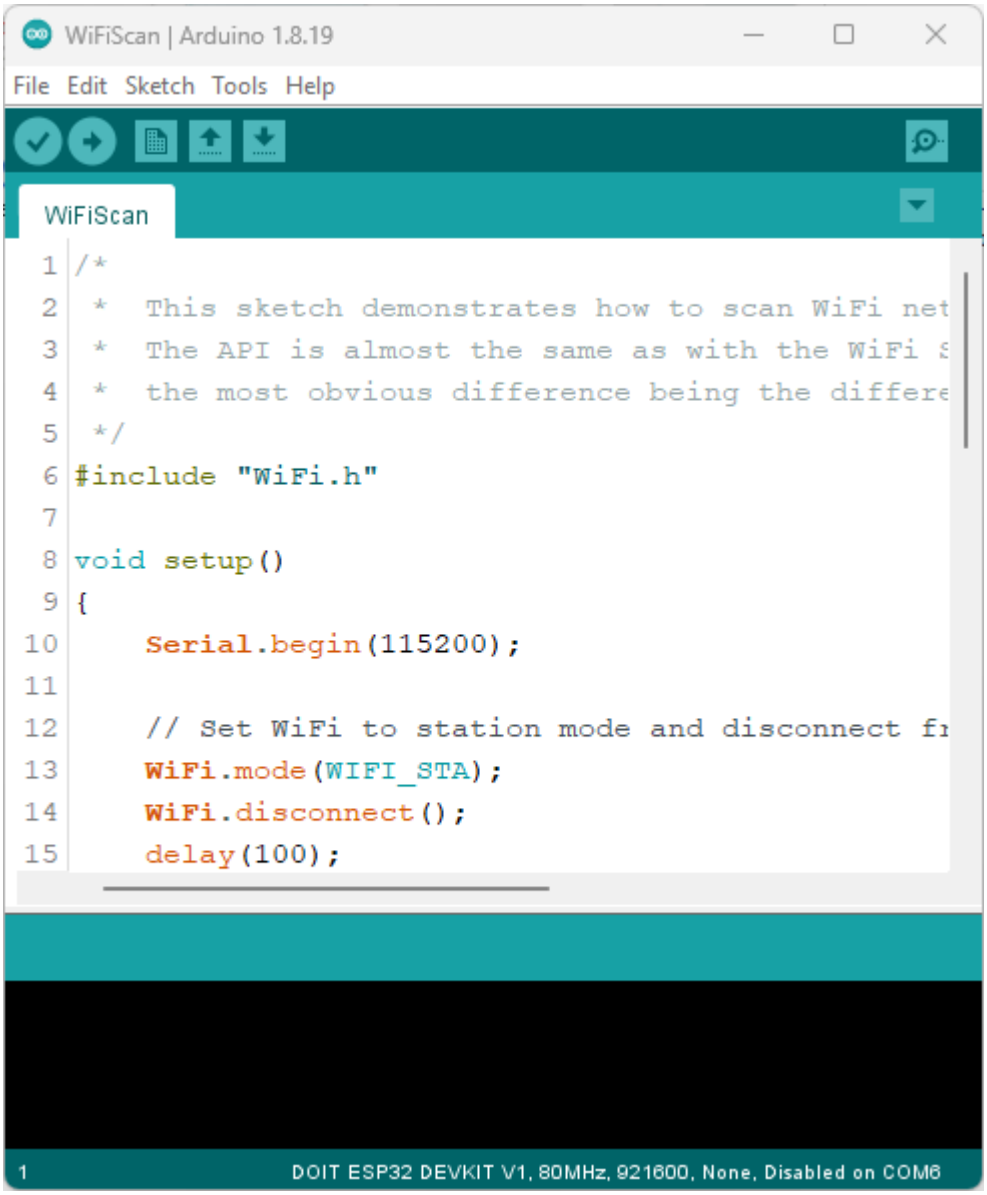
You just need to go to **Tools** > **Board > Boards Manager**, search for **ESP32**, and check the version that you have installed. If there is a more recent version available, select that version to install it.

Upload Code to the ESP32 using Arduino IDE

To show you how to upload code to your ESP32 board, we'll try a simple example available in the Arduino IDE examples for the ESP32.

First, make sure you have an ESP32 selected in **Tools** > **Board**. Then, go to **File** > **Examples** > **WiFi** > **WiFiScan**.

This will load a sketch that scans Wi-Fi networks within the range of your ESP32 board.



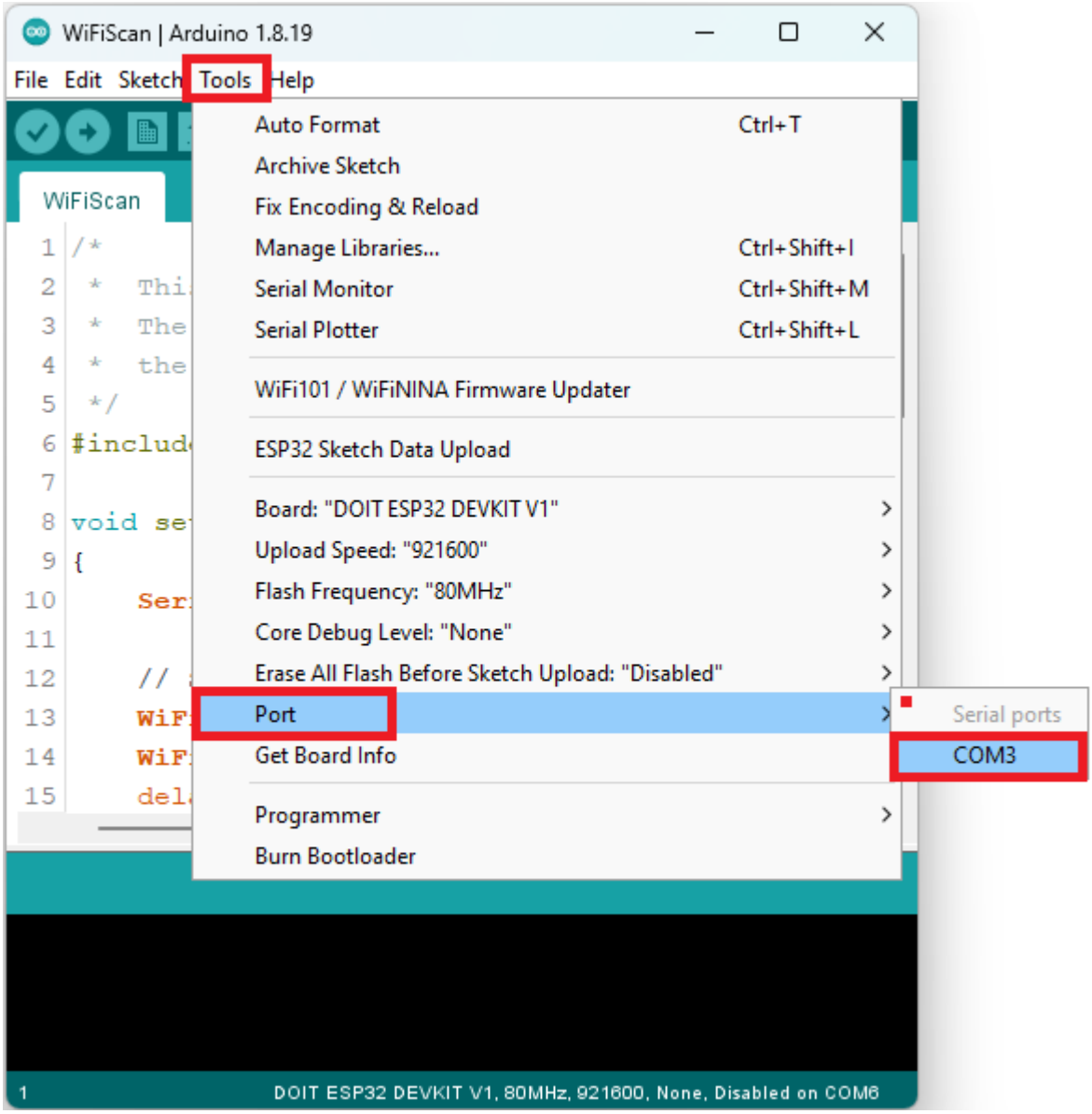
Connect your ESP32 development board to your computer using a USB cable. If you have an ESP32 DEVKIT DOIT board, the built-in red LED will turn on. This indicates the board is receiving power.

> **Important:** you must use a USB cable with data wires. Some USB cables from chargers or power banks are power only and they don't transfer data—these won't work.

Now, follow the next steps to upload the code.

1) Go to **Tools** > **Board**, scroll down to the ESP32 section and select the name of your ESP32 board. In my case, it's the DOIT ESP32 DEVKIT V1 board.

2) Go to **Tools** > **Port** and select a COM port available. If the COM port is grayed out, this means you don't have the required USB drivers. Check the section Installing USB Drivers before proceeding.



3) Press the upload button.

Some boards will automatically go into flashing mode and the code will be successfully uploaded straight away.

Other boards don't go into flashing mode automatically, so you may end up getting the following error.

```
Failed to connect to ESP32: Timed out... Connecting...
```

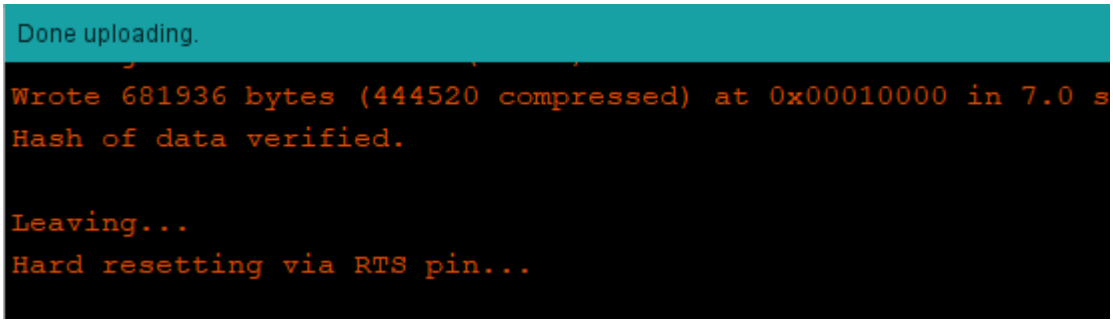
Or something like:

```
A fatal error occurred: Failed to connect to ESP32: Wrong boot
mode detected (0x13)! The chip needs to be in download mode.
```

This means the ESP32 was not in flashing mode when you tried to upload the code. In this situation, you should long press the board **BOOT** button, when you start seeing the "**Connecting….**" message on the debugging window.

Note: in some boards, a simple trick can make the ESP32 go into flashing mode automatically. Check it out on the following tutorial: [SOLVED] Failed to connect to ESP32: Timed out waiting for packet header.

Now, the code should be successfully uploaded to the board. You should get a "**Done uploading** "message".

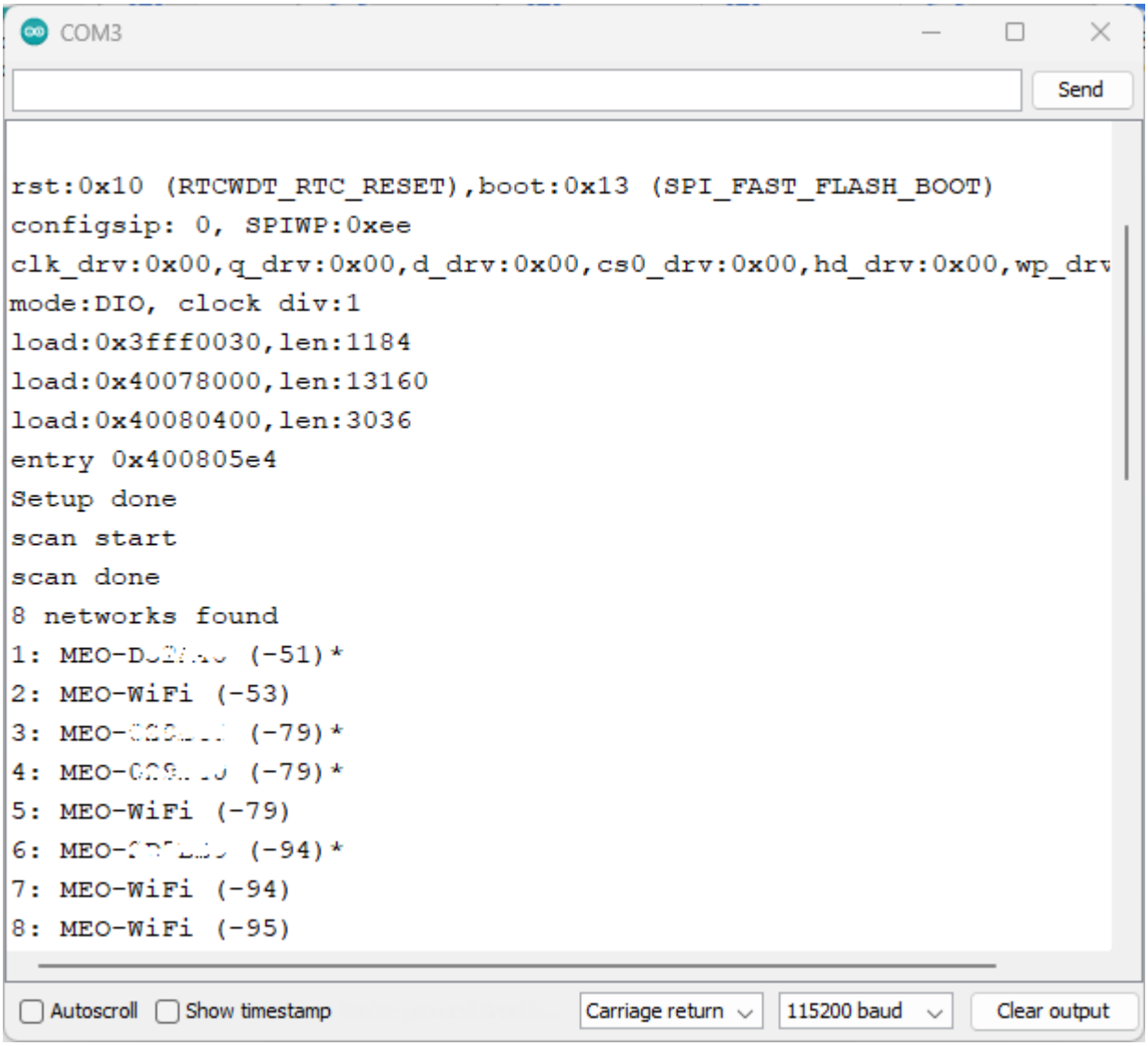


Demonstration

To see if the code is working as expected, open the Serial Monitor at a baud rate of 115200.

Press the ESP32 RST or EN button to restart the board and start running the newly uploaded code.

You should get a list of nearby wi-fi networks.



This means everything went as expected.

Installing ESP32 USB Drivers

After connecting the ESP32 board to your computer, if the COM port in Arduino IDE is grayed out, it means you don't have the necessary USB drivers installed on your computer.

Most ESP32 boards either use the CP2101 or CH340 drivers. Check the USB to UART converter on your board, and install the corresponding drivers.

You'll easily find instructions with a quick google search. For example "install CP2101 drivers Windows".

Wrapping Up

We hope you've found this getting started guide useful. I think we've included all the required information for you to get started. You learned what is an ESP32, how to choose an ESP32 development board, and how to upload new code to the ESP32 using Arduino IDE.

Want to learn more? We recommend the following tutorials to get started:

- ESP32 Digital Inputs and Digital Outputs (Arduino IDE)
- ESP32 Web Server Tutorial

Also, don't forget to take a look at the ESP32 pinout to learn how to use its GPIOs:

- ESP32 Pinout Reference: Which GPIO pins should you use?

If you're serious about learning about the ESP32, we recommend taking a look at our best-selling eBook:

- **Learn ESP32 with Arduino IDE eBook**

You can also check all our free ESP32 tutorials and guides on the following link:

- More ESP32 Projects

If you like ESP32 make sure you subscribe to our blog, so you don't miss upcoming projects.

Do you have any questions? Leave a comment down below!

Thanks for reading.

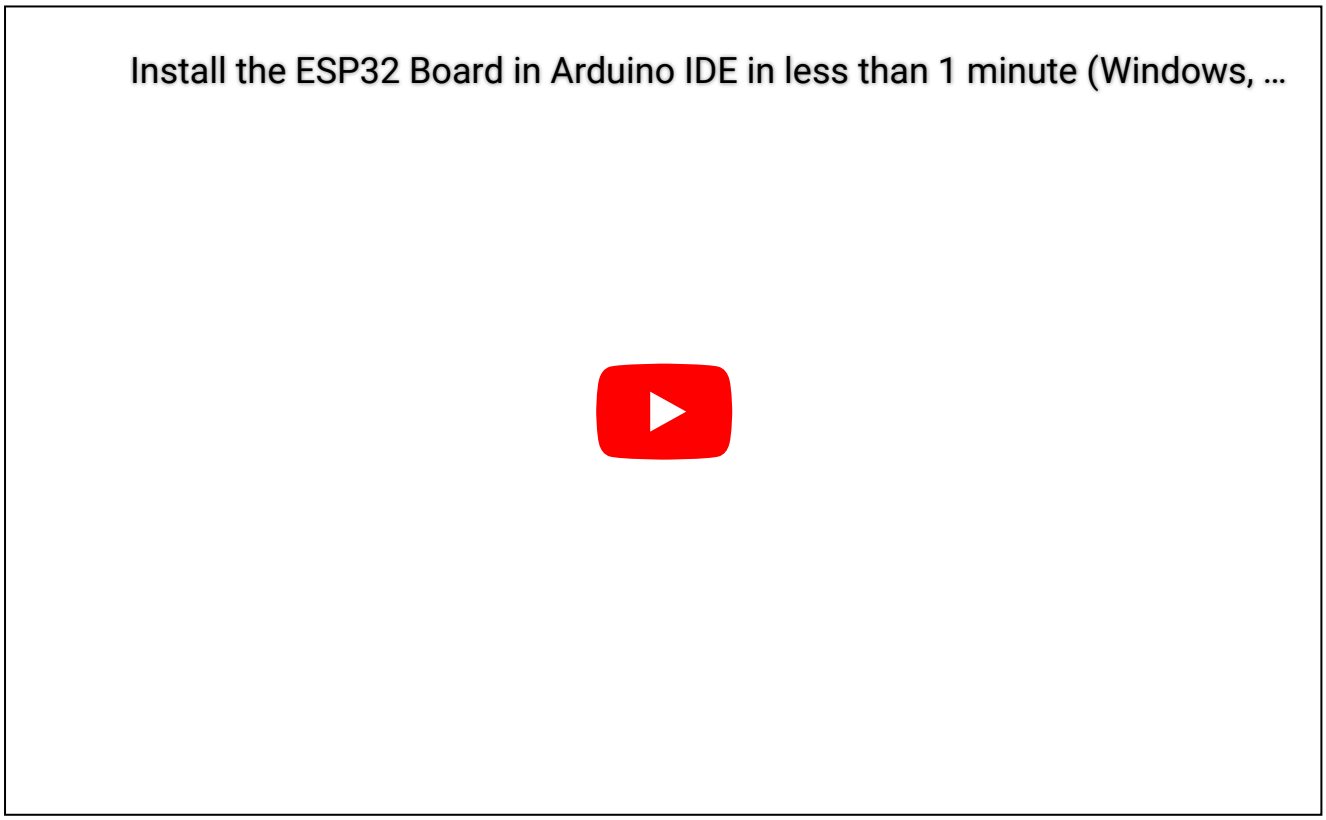
Installing the ESP32 Board in Arduino IDE (Windows, Mac OS X, Linux)

There's an add-on for the Arduino IDE that allows you to program the ESP32 using the Arduino IDE and its programming language. In this tutorial we'll show you how to install the ESP32 board in Arduino IDE whether you're using Windows, Mac OS X or Linux.

> Using Arduino 2.0? Follow this tutorial instead: Installing ESP32 Board in Arduino IDE 2.0

Watch the Video Tutorial

This tutorial is available in video format (watch below) and in written format (continue reading this page).



If you have any problems during the installation procedure, take a look at the ESP32 Troubleshooting Guide.

If you like the ESP32, enroll in our course: Learn ESP32 with Arduino IDE.

Prerequisites: Arduino IDE Installed

Before starting this installation procedure, you need to have Arduino IDE installed on your computer. There are two versions of the Arduino IDE you can install: version 1 and version 2.

You can download and install Arduino IDE by clicking on the following link: arduino.cc/en/Main/Software

Which Arduino IDE version do we recommend? At the moment, there are some plugins for the ESP32 (like the SPIFFS Filesystem Uploader Plugin) that are not yet supported on Arduino 2. So, if you intend to use the SPIFFS plugin in the future, we recommend installing the legacy version 1.8.X. You just need to scroll down on the Arduino software page to find it.

If you'll use Arduino 2, you can follow this tutorial instead:

- Installing ESP32 Board in Arduino IDE 2.0

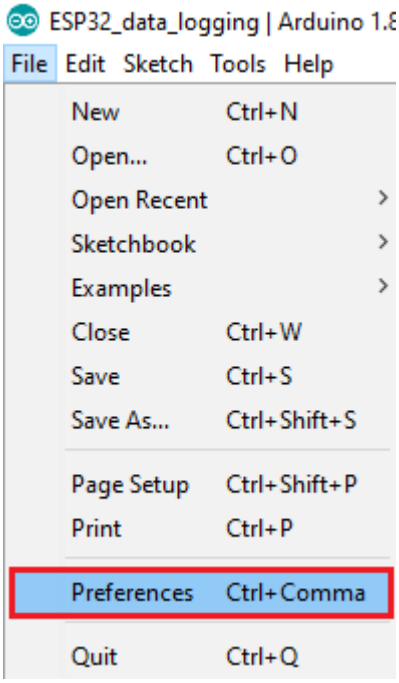
If later on, you need to install the SPIFFS plugin, you can install Arduino 1.8.X and have both versions installed on your computer.

Do you need an ESP32 board? You can buy it here.

Installing ESP32 Add-on in Arduino IDE

To install the ESP32 board in your Arduino IDE, follow these next instructions:

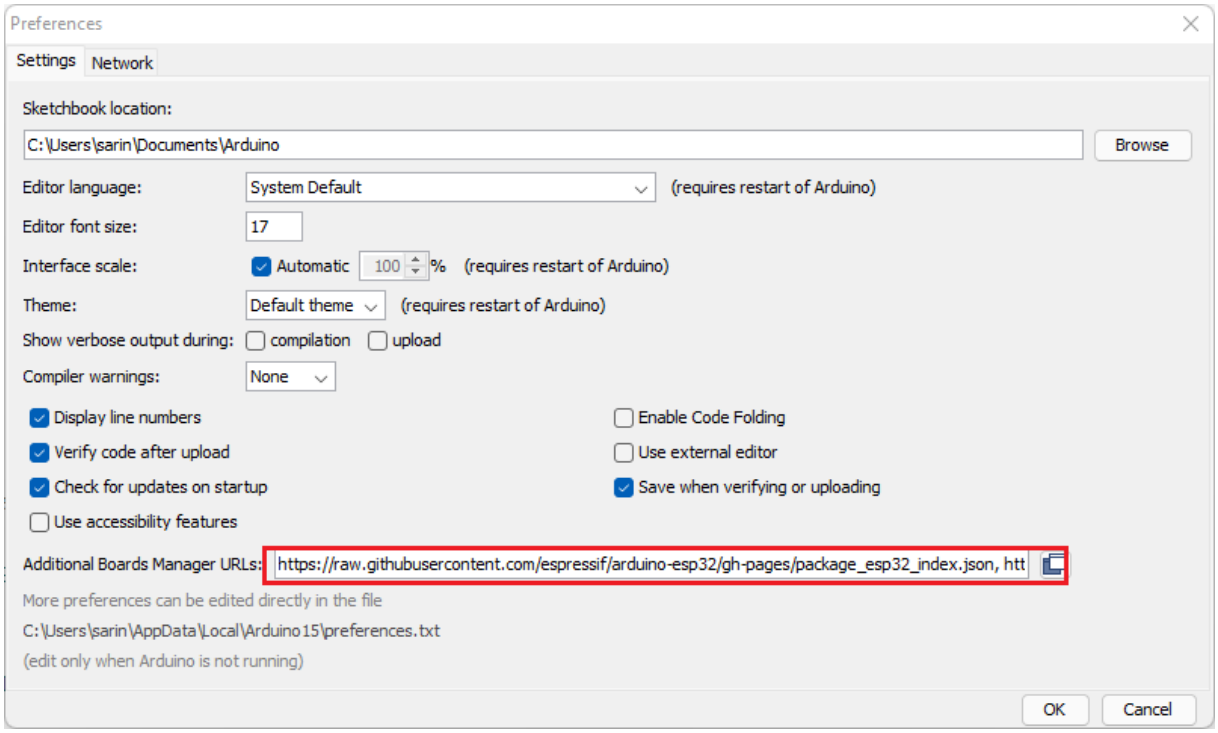
1. In your Arduino IDE, go to **File**> **Preferences**



2. Enter the following into the “Additional Board Manager URLs” field:

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json
```

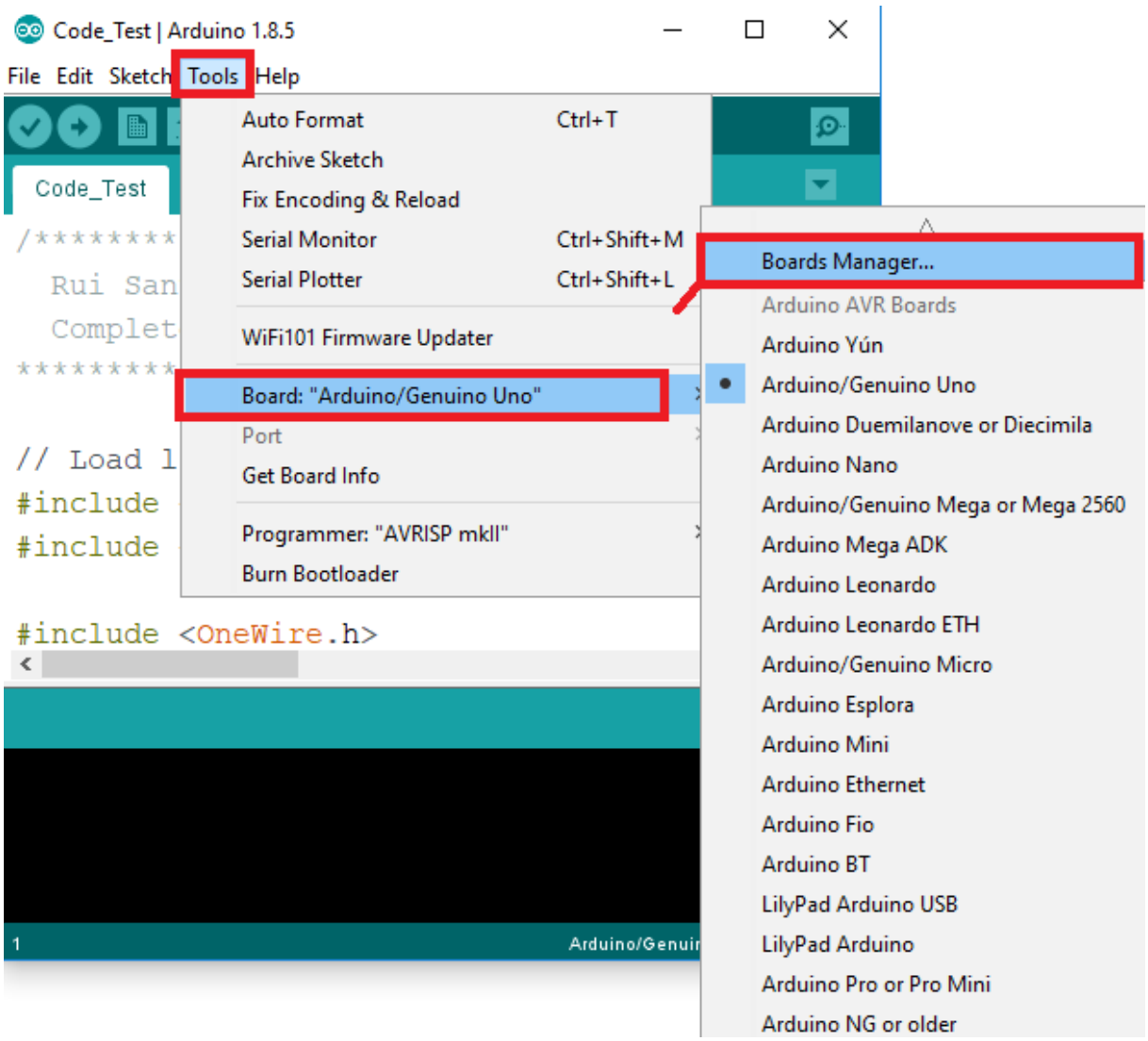
Then, click the “OK” button:



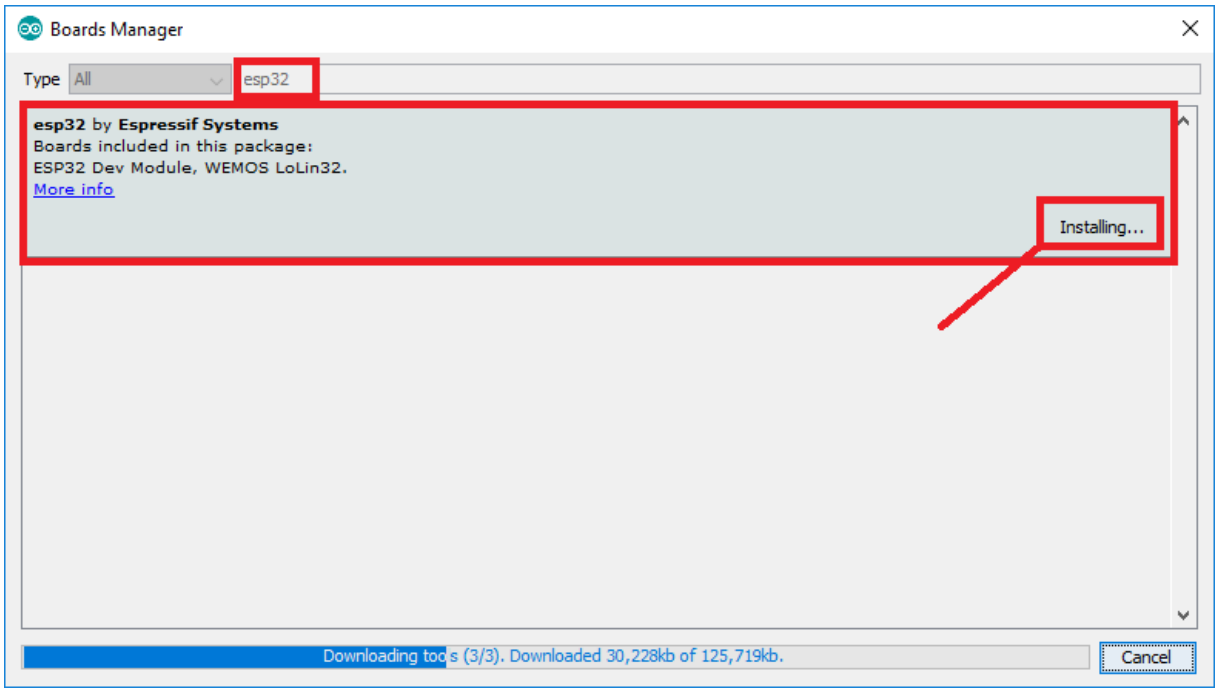
Note: if you already have the ESP8266 boards URL, you can separate the URLs with a comma as follows:

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json,
http://arduino.esp8266.com/stable/package_esp8266com_index.json
```

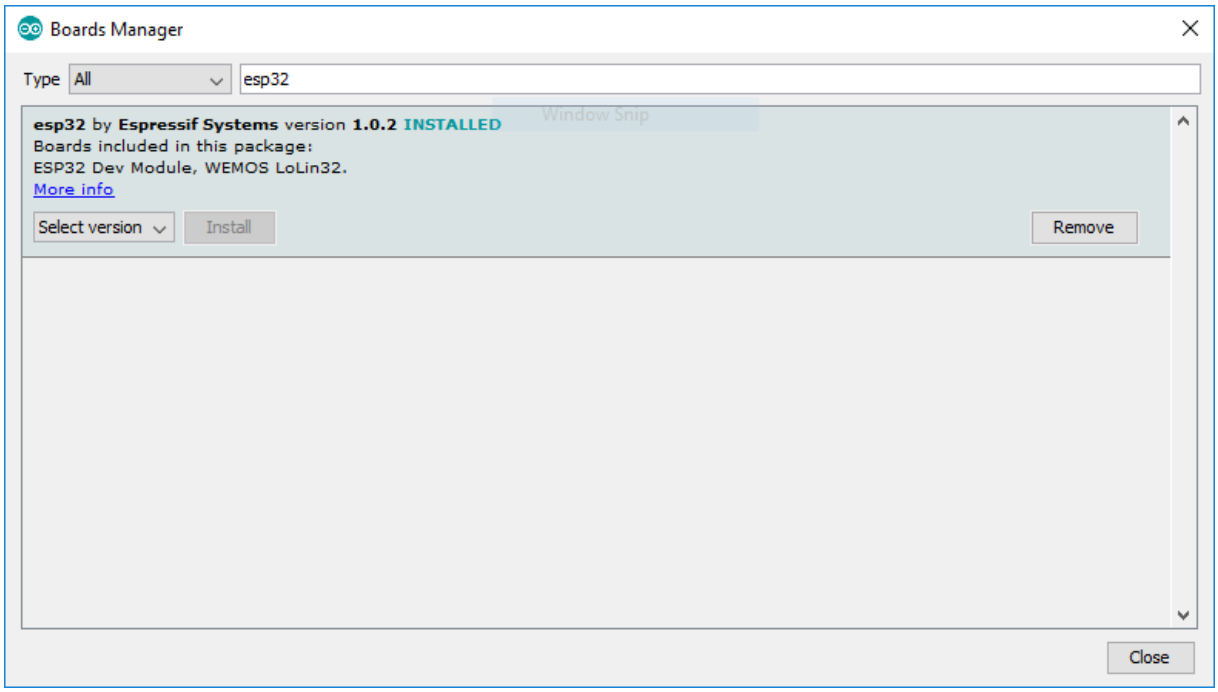
3. Open the Boards Manager. Go to **Tools** > **Board** > **Boards Manager…**



4. Search for **ESP32** and press install button for the “**ESP32 by Espressif Systems**“:



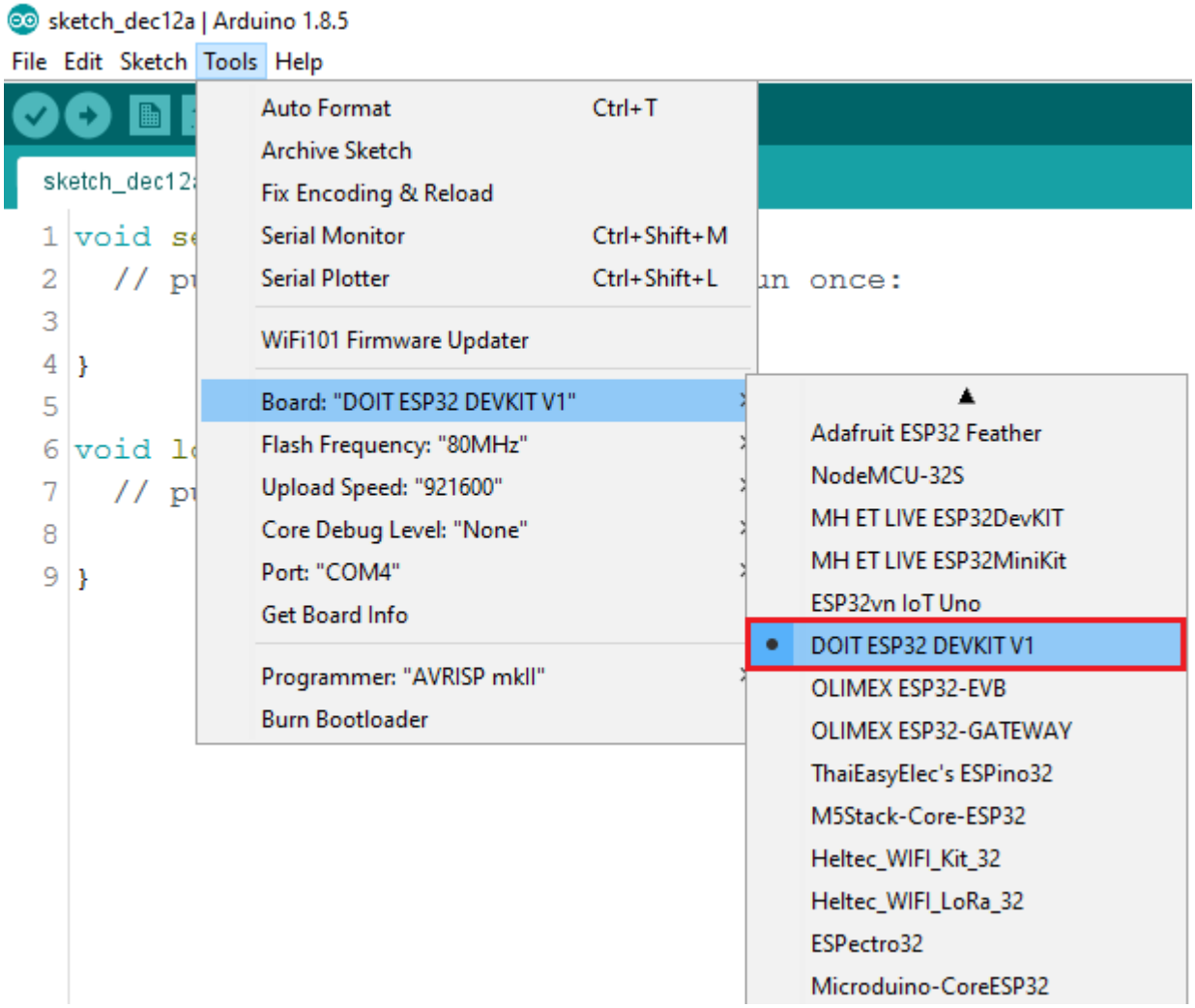
5. That's it. It should be installed after a few seconds.



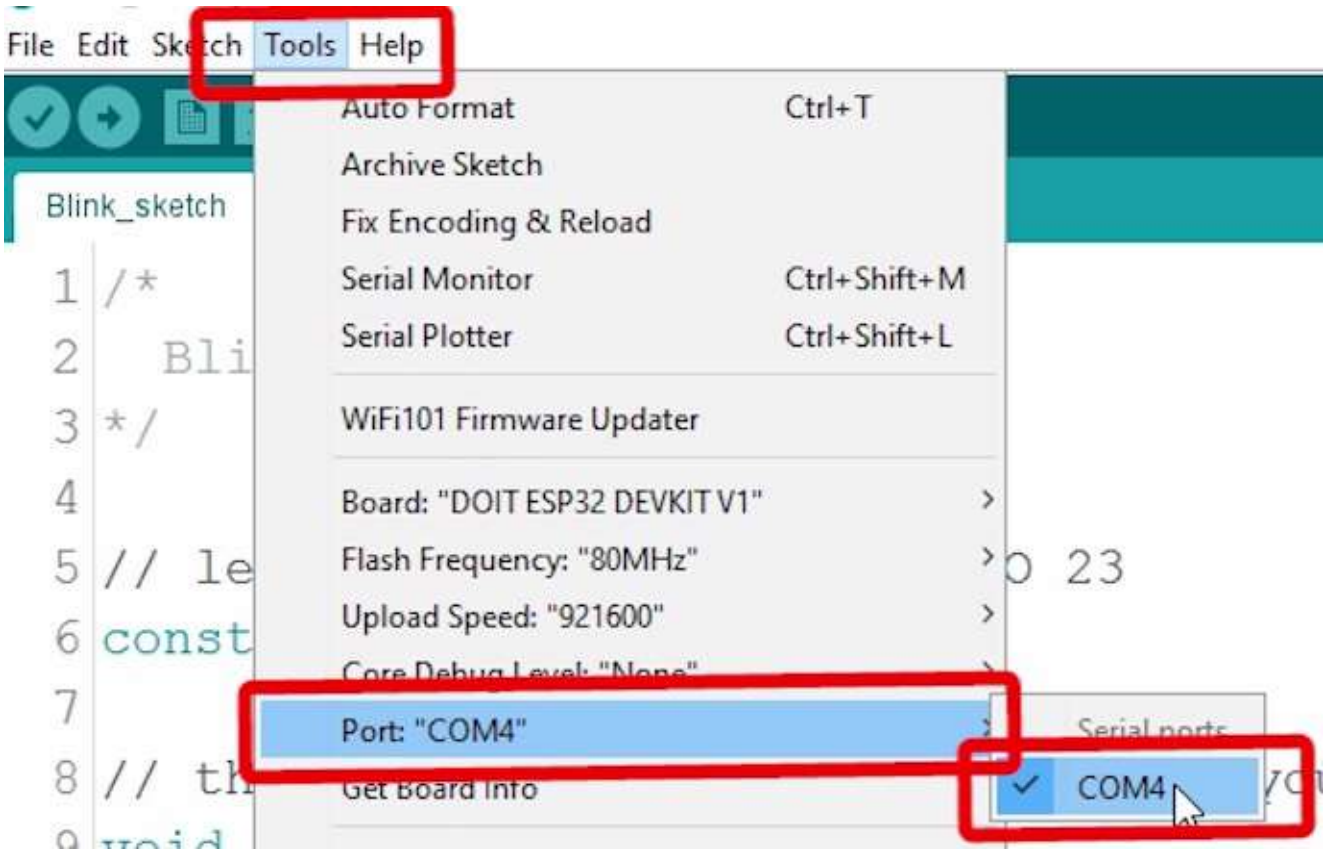
Testing the Installation

Plug the ESP32 board to your computer. With your Arduino IDE open, follow these steps:

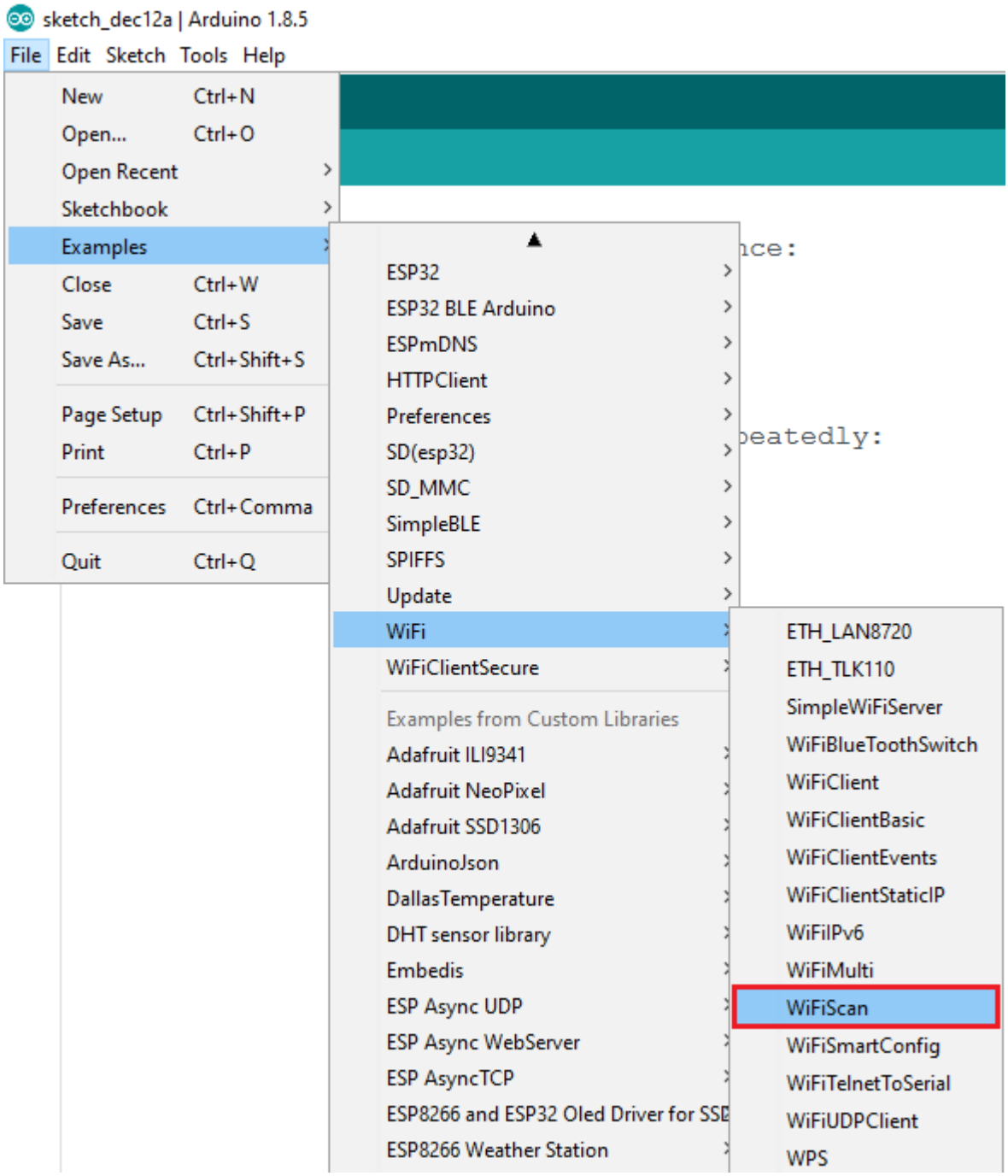
1. Select your Board in **Tools** > **Board** menu (in my case it's the **DOIT ESP32 DEVKIT V1**)



2. Select the Port (if you don't see the COM Port in your Arduino IDE, you need to install the CP210x USB to UART Bridge VCP Drivers):



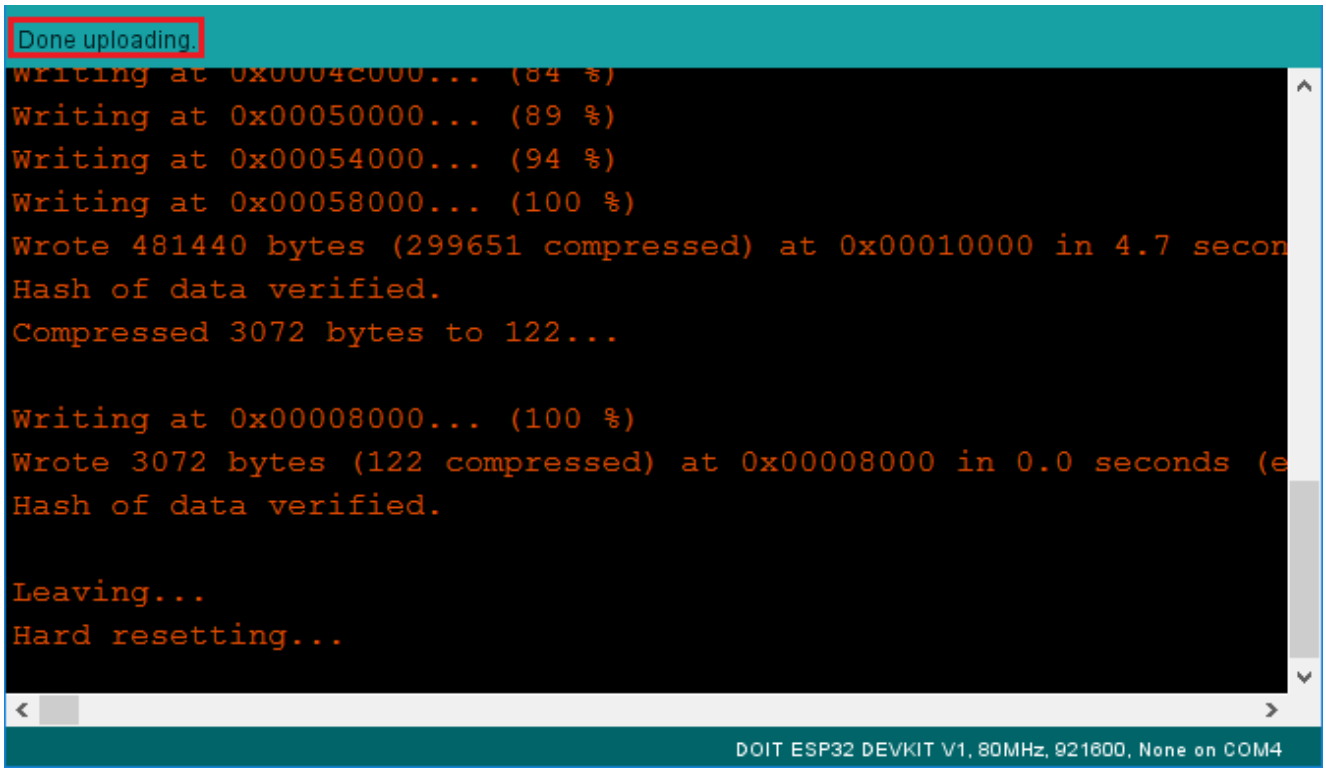
3. Open the following example under **File** > **Examples** > **WiFi (ESP32)** > **WiFiScan**



4. A new sketch opens in your Arduino IDE:

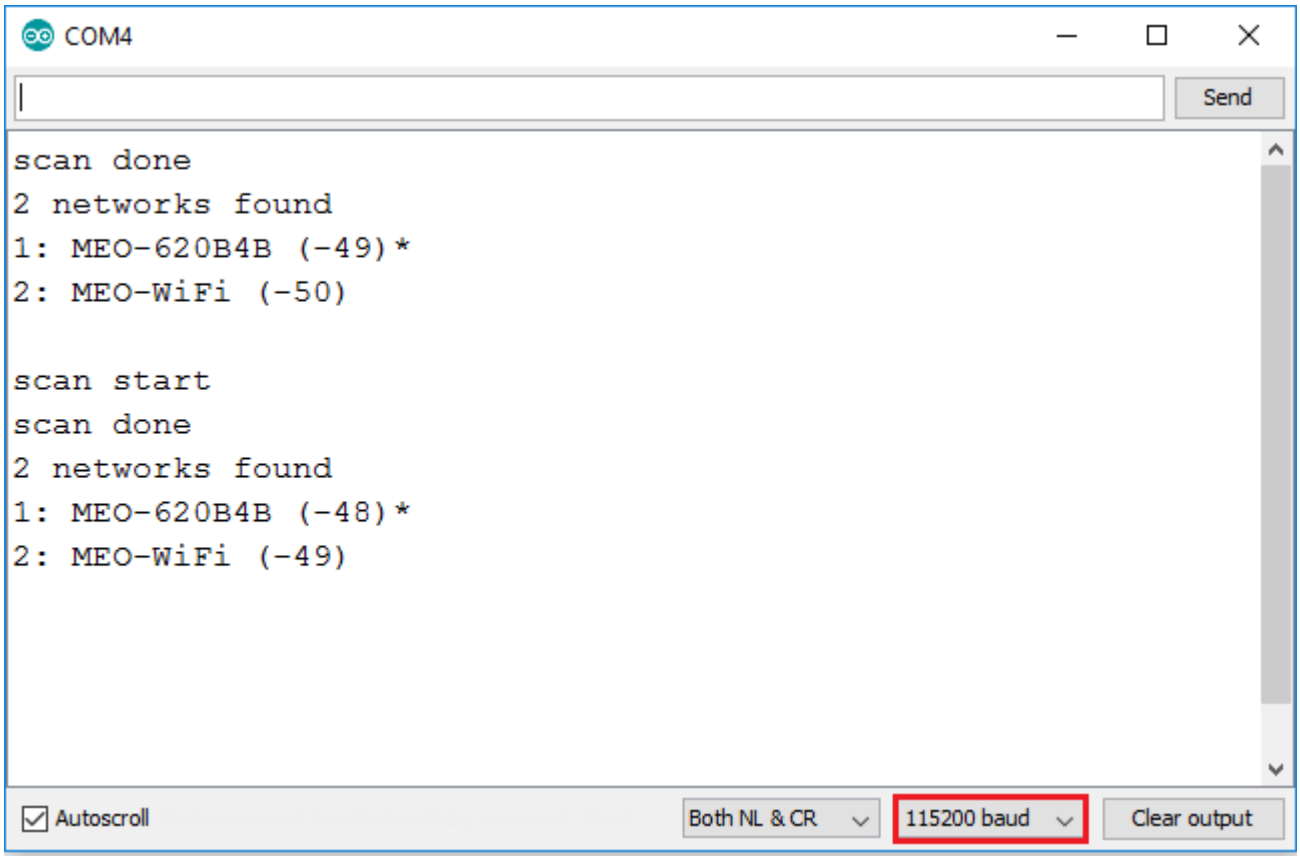
5. Press the **Upload** button in the Arduino IDE. Wait a few seconds while the code compiles and uploads to your board.

6. If everything went as expected, you should see a “**Done uploading.**” message.



7. Open the Arduino IDE Serial Monitor at a baud rate of 115200:

8. Press the ESP32 on-board **Enable** button and you should see the networks available near your ESP32:

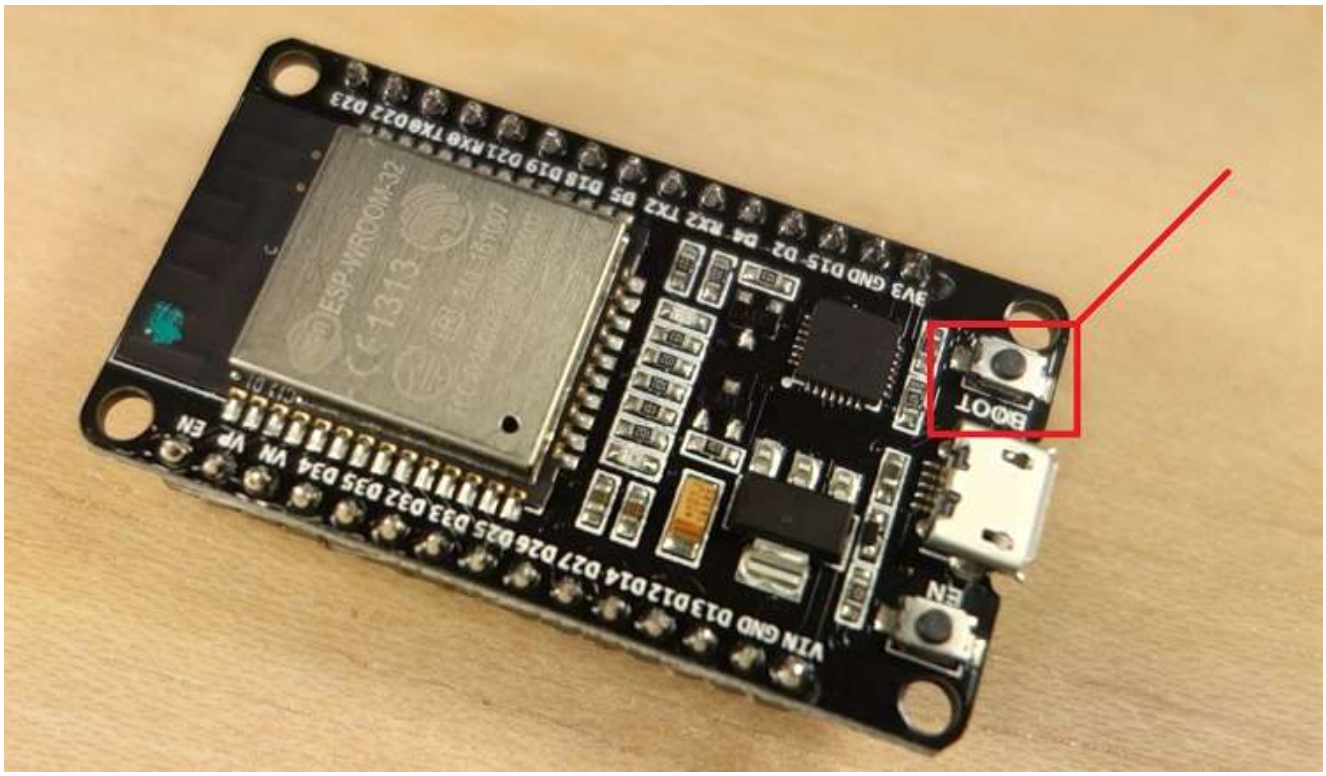


Troubleshooting

1) If you try to upload a new sketch to your ESP32 and you get this error message "*A fatal error occurred: Failed to connect to ESP32: Timed out… Connecting…*". It means that your ESP32 is not in flashing/uploading mode.

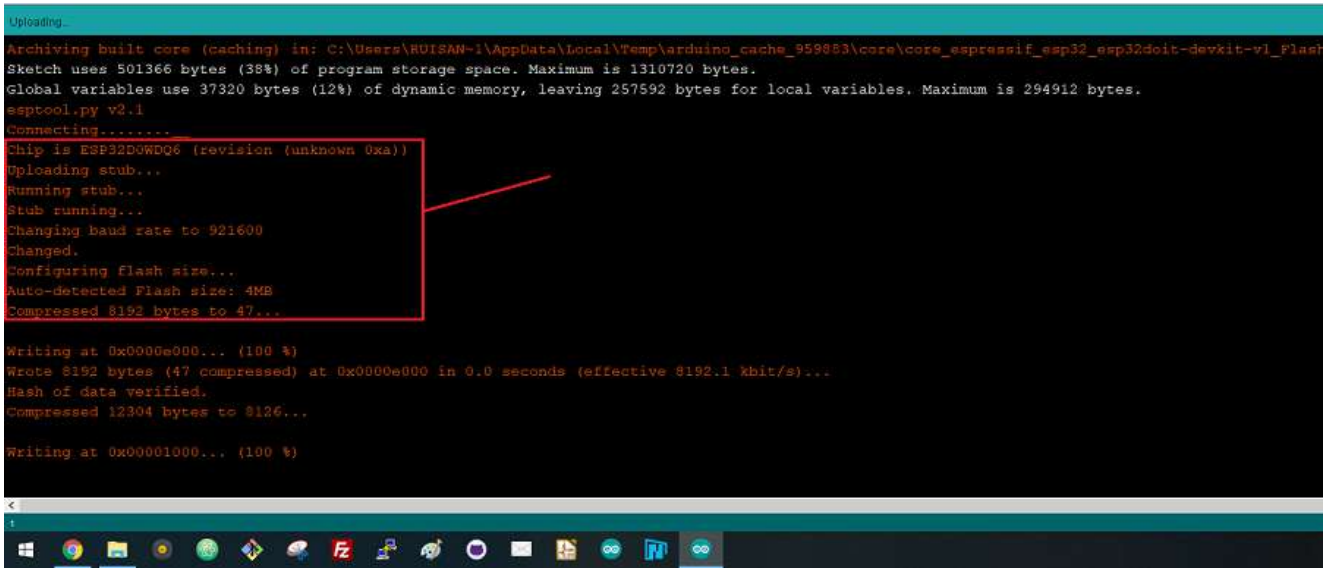
Having the right board name and COM port selected, follow these steps:

- Hold-down the "**BOOT**" button in your ESP32 board



- Press the "**Upload**" button in the Arduino IDE to upload your sketch:

- After you see the "**Connecting….**" message in your Arduino IDE, release the finger from the "**BOOT**" button:



- After that, you should see the "**Done uploading**" message

That's it. Your ESP32 should have the new sketch running. Press the "**ENABLE**" button to restart the ESP32 and run the new uploaded sketch.

You'll also have to repeat that button sequence every time you want to upload a new sketch. But if you want to solve this issue once for all without the need to press the **BOOT** button, follow the suggestions in the next guide:

- [SOLVED] Failed to connect to ESP32: Timed out waiting for packet header

2) If you get the error "COM Port not found/not available", you might need to install the CP210x Drivers:

- Install USB Drivers – CP210x USB to UART Bridge (Windows PC)
- Install USB Drivers – CP210x USB to UART Bridge (Mac OS X)

If you experience any problems or issues with your ESP32, take a look at our in-depth ESP32 Troubleshooting Guide.

Wrapping Up

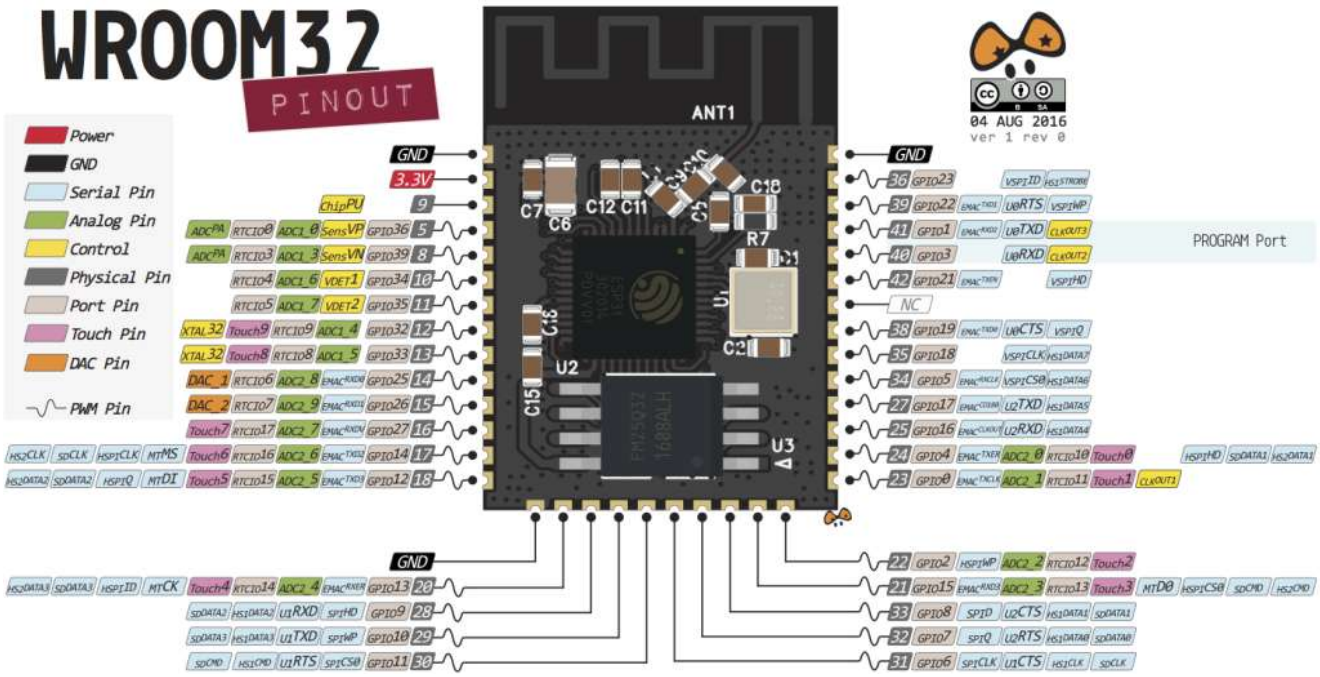
This is a quick guide that illustrates how to prepare your Arduino IDE for the ESP32 on a Windows PC, Mac OS X, or Linux computer. If you encounter any issues during the installation procedure, take a look at the ESP32 troubleshooting guide.

ESP32 Pinout Reference: Which GPIO pins should you use?

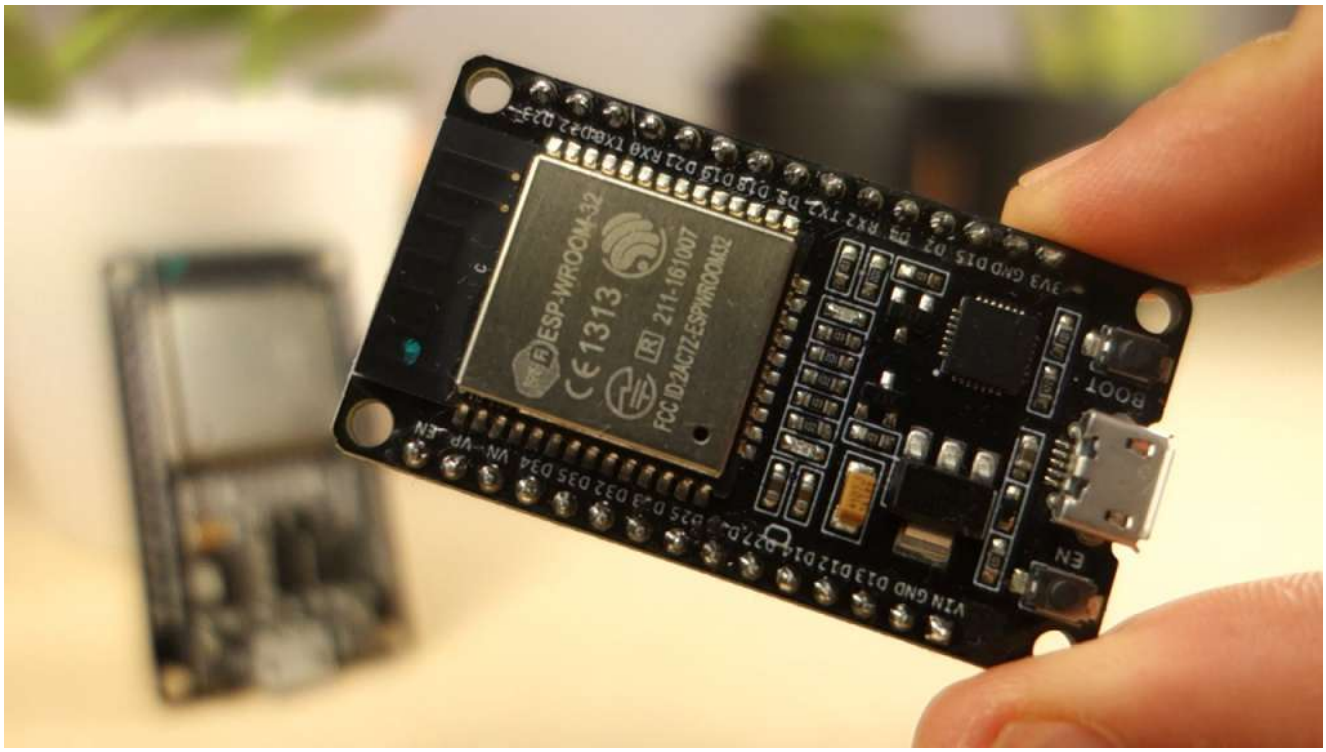
The ESP32 chip comes with 48 pins with multiple functions. Not all pins are exposed in all ESP32 development boards, and some pins cannot be used.

There are many questions on how to use the ESP32 GPIOs. What pins should you use? What pins should you avoid using in your projects? This post aims to be a simple and easy-to-follow reference guide for the ESP32 GPIOs.

The figure below illustrates the ESP-WROOM-32 pinout. You can use it as a reference if you're using an **ESP32 bare chip** to build a custom board:



Note: not all GPIOs are accessible in all development boards, but each specific GPIO works in the same way regardless of the development board you're using. If you're just getting started with the ESP32, we recommend reading our guide: Getting Started with the ESP32 Development Board.



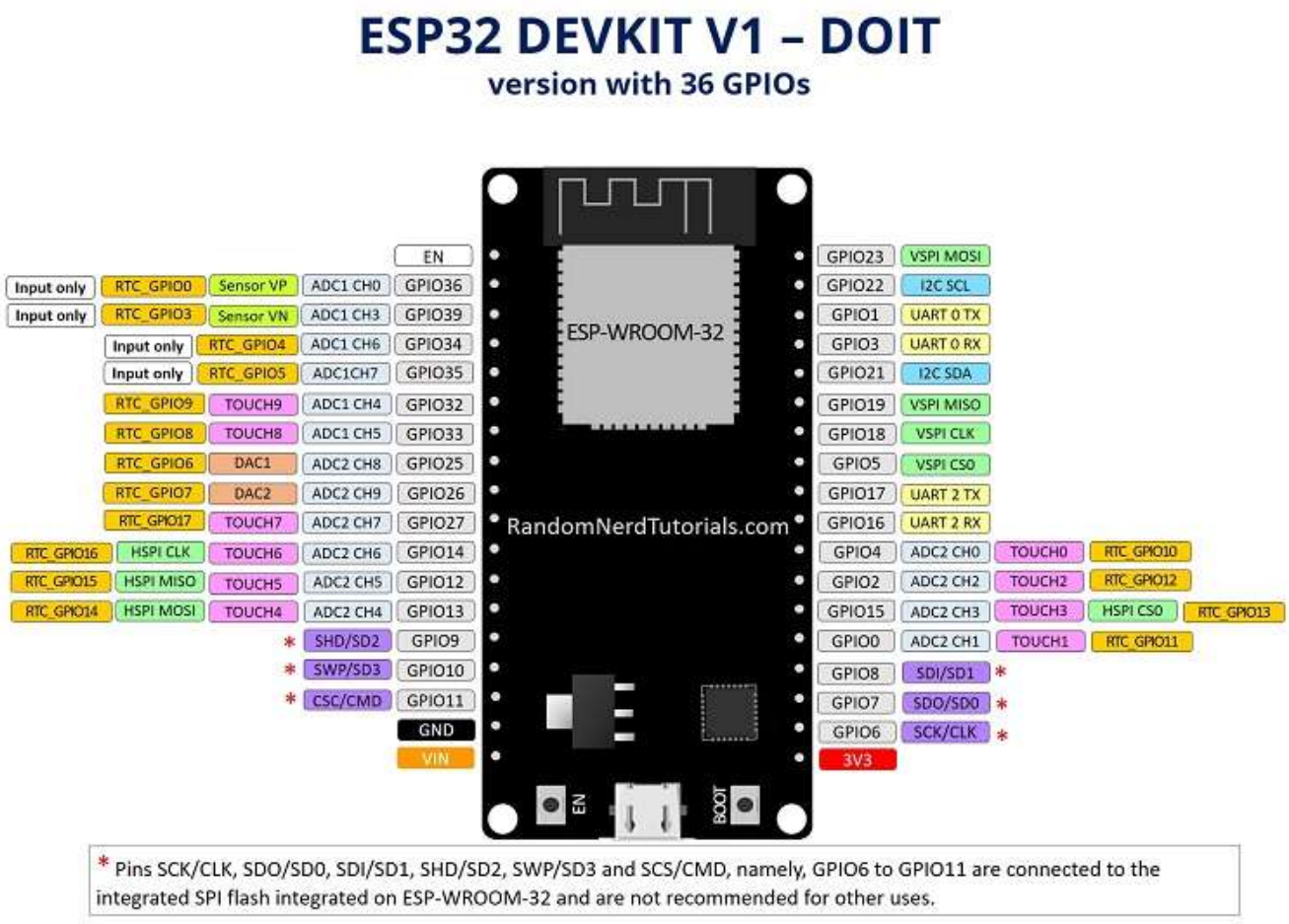
ESP32 Peripherals

The ESP32 peripherals include:

- 18 Analog-to-Digital Converter (ADC) channels
- 3 SPI interfaces
- 3 UART interfaces
- 2 I2C interfaces
- 16 PWM output channels
- 2 Digital-to-Analog Converters (DAC)
- 2 I2S interfaces
- 10 Capacitive sensing GPIOs

The ADC (analog to digital converter) and DAC (digital to analog converter) features are assigned to specific static pins. However, you can decide which pins are UART, I2C, SPI, PWM, etc – you just need to assign them in the code. This is possible due to the ESP32 chip's multiplexing feature.

Although you can define the pins properties on the software, there are pins assigned by default as shown in the following figure (this is an example for the ESP32 DEVKIT V1 DOIT board with 36 pins – the pin location can change depending on the manufacturer).



Additionally, there are pins with specific features that make them suitable or not for a particular project. The following table shows what pins are best to use as inputs, outputs and which ones you need to be cautious.

The pins highlighted in green are OK to use. The ones highlighted in yellow are OK to use, but you need to pay attention because they may have an unexpected behavior mainly at boot. The pins highlighted in red are not recommended to use as inputs or outputs.

GPIO	Input	Output	Notes
0	pulled up	OK	outputs PWM signal at boot, must be LOW to enter flashing mode
1	TX pin	OK	debug output at boot
2	OK	OK	connected to on-board LED, must be left floating or LOW to enter flashing mode

3	OK	RX pin	HIGH at boot
4	OK	OK	
5	OK	OK	outputs PWM signal at boot, strapping pin
6	x	x	connected to the integrated SPI flash
7	x	x	connected to the integrated SPI flash
8	x	x	connected to the integrated SPI flash
9	x	x	connected to the integrated SPI flash
10	x	x	connected to the integrated SPI flash
11	x	x	connected to the integrated SPI flash
12	OK	OK	boot fails if pulled high, strapping pin
13	OK	OK	
14	OK	OK	outputs PWM signal at boot
15	OK	OK	outputs PWM signal at boot, strapping pin
16	OK	OK	
17	OK	OK	
18	OK	OK	
19	OK	OK	
21	OK	OK	
22	OK	OK	
23	OK	OK	
25	OK	OK	
26	OK	OK	

27	OK	OK	
32	OK	OK	
33	OK	OK	
34	OK		input only
35	OK		input only
36	OK		input only
39	OK		input only

Continue reading for a more detail and in-depth analysis of the ESP32 GPIOs and its functions.

Input only pins

GPIOs 34 to 39 are GPIs – input only pins. These pins don't have internal pull-up or pull-down resistors. They can't be used as outputs, so use these pins only as inputs:

- GPIO 34
- GPIO 35
- GPIO 36
- GPIO 39

SPI flash integrated on the ESP-WROOM-32

GPIO 6 to GPIO 11 are exposed in some ESP32 development boards. However, these pins are connected to the integrated SPI flash on the ESP-WROOM-32 chip and are not recommended for other uses. So, don't use these pins in your projects:

- GPIO 6 (SCK/CLK)
- GPIO 7 (SDO/SD0)
- GPIO 8 (SDI/SD1)
- GPIO 9 (SHD/SD2)

- GPIO 10 (SWP/SD3)
- GPIO 11 (CSC/CMD)

Capacitive touch GPIOs

The ESP32 has 10 internal capacitive touch sensors. These can sense variations in anything that holds an electrical charge, like the human skin. So they can detect variations induced when touching the GPIOs with a finger. These pins can be easily integrated into capacitive pads and replace mechanical buttons. The capacitive touch pins can also be used to wake up the ESP32 from deep sleep.

Those internal touch sensors are connected to these GPIOs:

- T0 (GPIO 4)
- T1 (GPIO 0)
- T2 (GPIO 2)
- T3 (GPIO 15)
- T4 (GPIO 13)
- T5 (GPIO 12)
- T6 (GPIO 14)
- T7 (GPIO 27)
- T8 (GPIO 33)
- T9 (GPIO 32)

Learn how to use the touch pins with Arduino IDE: ESP32 Touch Pins with Arduino IDE

Analog to Digital Converter (ADC)

The ESP32 has 18 x 12 bits ADC input channels (while the ESP8266 only has 1x 10 bits ADC). These are the GPIOs that can be used as ADC and respective channels:

- ADC1_CH0 (GPIO 36)
- ADC1_CH1 (GPIO 37)
- ADC1_CH2 (GPIO 38)
- ADC1_CH3 (GPIO 39)
- ADC1_CH4 (GPIO 32)
- ADC1_CH5 (GPIO 33)

- ADC1_CH6 (GPIO 34)
- ADC1_CH7 (GPIO 35)
- ADC2_CH0 (GPIO 4)
- ADC2_CH1 (GPIO 0)
- ADC2_CH2 (GPIO 2)
- ADC2_CH3 (GPIO 15)
- ADC2_CH4 (GPIO 13)
- ADC2_CH5 (GPIO 12)
- ADC2_CH6 (GPIO 14)
- ADC2_CH7 (GPIO 27)
- ADC2_CH8 (GPIO 25)
- ADC2_CH9 (GPIO 26)

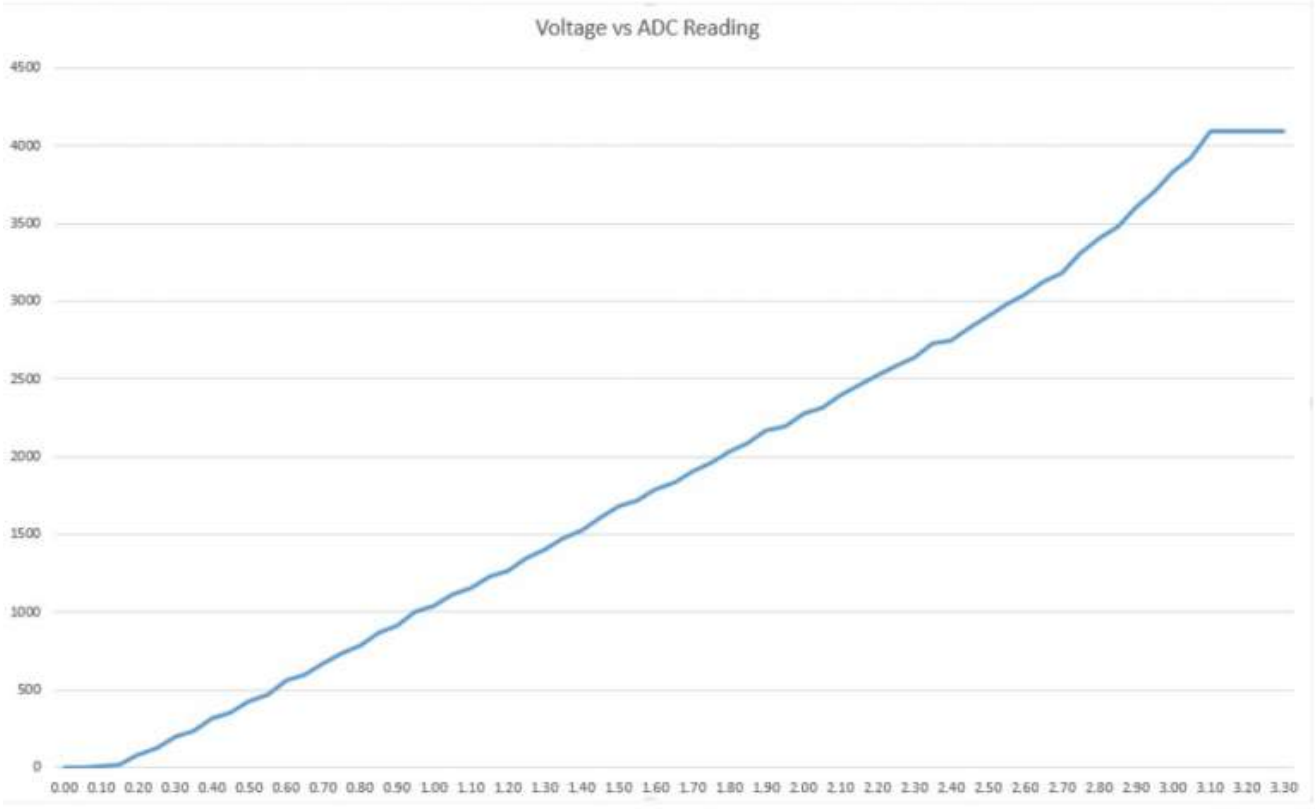
Learn how to use the ESP32 ADC pins:

- ESP32 ADC Pins with Arduino IDE
- ESP32 ADC Pins with MicroPython

> **Note:** ADC2 pins cannot be used when Wi-Fi is used. So, if you're using Wi-Fi and you're having trouble getting the value from an ADC2 GPIO, you may consider using an ADC1 GPIO instead. That should solve your problem.

The ADC input channels have a 12-bit resolution. This means that you can get analog readings ranging from 0 to 4095, in which 0 corresponds to 0V and 4095 to 3.3V. You can also set the resolution of your channels on the code and the ADC range.

The ESP32 ADC pins don't have a linear behavior. You'll probably won't be able to distinguish between 0 and 0.1V, or between 3.2 and 3.3V. You need to keep that in mind when using the ADC pins. You'll get a behavior similar to the one shown in the following figure.



View source

Digital to Analog Converter (DAC)

There are 2 x 8 bits DAC channels on the ESP32 to convert digital signals into analog voltage signal outputs. These are the DAC channels:

- DAC1 (GPIO25)
- DAC2 (GPIO26)

RTC GPIOs

There is RTC GPIO support on the ESP32. The GPIOs routed to the RTC low-power subsystem can be used when the ESP32 is in deep sleep. These RTC GPIOs can be used to wake up the ESP32 from deep sleep when the Ultra Low Power (ULP) co-processor is running. The following GPIOs can be used as an external wake up source.

- RTC_GPIO0 (GPIO36)
- RTC_GPIO3 (GPIO39)
- RTC_GPIO4 (GPIO34)
- RTC_GPIO5 (GPIO35)
- RTC_GPIO6 (GPIO25)

- RTC_GPIO7 (GPIO26)
- RTC_GPIO8 (GPIO33)
- RTC_GPIO9 (GPIO32)
- RTC_GPIO10 (GPIO4)
- RTC_GPIO11 (GPIO0)
- RTC_GPIO12 (GPIO2)
- RTC_GPIO13 (GPIO15)
- RTC_GPIO14 (GPIO13)
- RTC_GPIO15 (GPIO12)
- RTC_GPIO16 (GPIO14)
- RTC_GPIO17 (GPIO27)

Learn how to use the RTC GPIOs to wake up the ESP32 from deep sleep: ESP32 Deep Sleep with Arduino IDE and Wake Up Sources

PWM

The ESP32 LED PWM controller has 16 independent channels that can be configured to generate PWM signals with different properties. All pins that can act as outputs can be used as PWM pins (GPIOs 34 to 39 can't generate PWM).

To set a PWM signal, you need to define these parameters in the code:

- Signal's frequency;
- Duty cycle;
- PWM channel;
- GPIO where you want to output the signal.

Learn how to use ESP32 PWM with Arduino IDE: ESP32 PWM with Arduino IDE

I2C

The ESP32 has two I2C channels and any pin can be set as SDA or SCL. When using the ESP32 with the Arduino IDE, the default I2C pins are:

- GPIO 21 (SDA)
- GPIO 22 (SCL)

If you want to use other pins when using the wire library, you just need to call:

```
Wire.begin(SDA, SCL);
```

Learn more about I2C communication protocol with the ESP32 using Arduino IDE: ESP32 I2C Communication (Set Pins, Multiple Bus Interfaces and Peripherals)

SPI

By default, the pin mapping for SPI is:

SPI	MOSI	MISO	CLK	CS
VSPI	GPIO 23	GPIO 19	GPIO 18	GPIO 5
HSPI	GPIO 13	GPIO 12	GPIO 14	GPIO 15

Learn more about SPI communication protocol with the ESP32 using Arduino IDE: ESP32 SPI Communication: Set Pins, Multiple SPI Bus Interfaces, and Peripherals (Arduino IDE)

Interrupts

All GPIOs can be configured as interrupts.

Learn how to use interrupts with the ESP32:

- ESP32 interrupts with Arduino IDE
- ESP32 interrupts with MicroPython

Strapping Pins

The ESP32 chip has the following strapping pins:

- GPIO 0 (must be LOW to enter boot mode)
- GPIO 2 (must be floating or LOW during boot)
- GPIO 4
- GPIO 5 (must be HIGH during boot)
- GPIO 12 (must be LOW during boot)

- GPIO 15 (must be HIGH during boot)

These are used to put the ESP32 into bootloader or flashing mode. On most development boards with built-in USB/Serial, you don't need to worry about the state of these pins. The board puts the pins in the right state for flashing or boot mode. More information on the ESP32 Boot Mode Selection can be found here.

However, if you have peripherals connected to those pins, you may have trouble trying to upload new code, flashing the ESP32 with new firmware, or resetting the board. If you have some peripherals connected to the strapping pins and you are getting trouble uploading code or flashing the ESP32, it may be because those peripherals are preventing the ESP32 from entering the right mode. Read the Boot Mode Selection documentation to guide you in the right direction. After resetting, flashing, or booting, those pins work as expected.

Pins HIGH at Boot

Some GPIOs change their state to HIGH or output PWM signals at boot or reset. This means that if you have outputs connected to these GPIOs you may get unexpected results when the ESP32 resets or boots.

- GPIO 1
- GPIO 3
- GPIO 5
- GPIO 6 to GPIO 11 (connected to the ESP32 integrated SPI flash memory – not recommended to use).
- GPIO 14
- GPIO 15

Enable (EN)

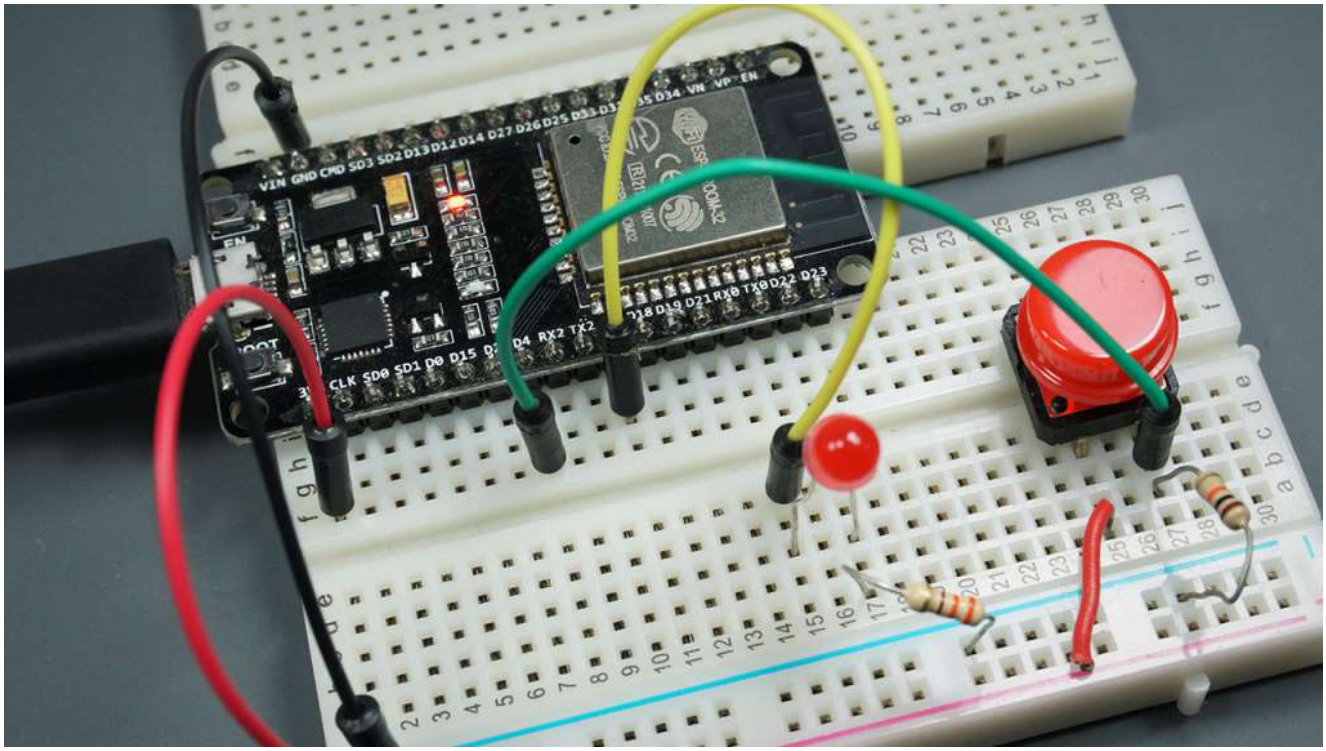
Enable (EN) is the 3.3V regulator's enable pin. It's pulled up, so connect to ground to disable the 3.3V regulator. This means that you can use this pin connected to a pushbutton to restart your ESP32, for example.

GPIO current drawn

The absolute maximum current drawn per GPIO is 40mA according to the "Recommended Operating Conditions" section in the ESP32 datasheet.

ESP32 Digital Inputs and Digital Outputs (Arduino IDE)

In this getting started guide you'll learn how to read digital inputs like a button switch and control digital outputs like an LED using the ESP32 with Arduino IDE.



Prerequisites

We'll program the ESP32 using Arduino IDE. So, make sure you have the ESP32 boards add-on installed before proceeding:

- Installing the ESP32 Board in Arduino IDE (Windows, Mac OS X, Linux)

ESP32 Control Digital Outputs

First, you need set the GPIO you want to control as an `OUTPUT`. Use the `pinMode()` function as follows:

```
pinMode(GPIO, OUTPUT);
```

To control a digital output you just need to use the `digitalWrite()` function, that accepts as arguments, the GPIO (int number) you are referring to, and the state, either `HIGH` or `LOW`.

```
digitalWrite(GPIO, STATE);
```

All GPIOs can be used as outputs except GPIOs 6 to 11 (connected to the integrated SPI flash) and GPIOs 34, 35, 36 and 39 (input only GPIOs);

Learn more about the ESP32 GPIOs: ESP32 GPIO Reference Guide

ESP32 Read Digital Inputs

First, set the GPIO you want to read as `INPUT`, using the `pinMode()` function as follows:

```
pinMode(GPIO, INPUT);
```

To read a digital input, like a button, you use the `digitalRead()` function, that accepts as argument, the GPIO (int number) you are referring to.

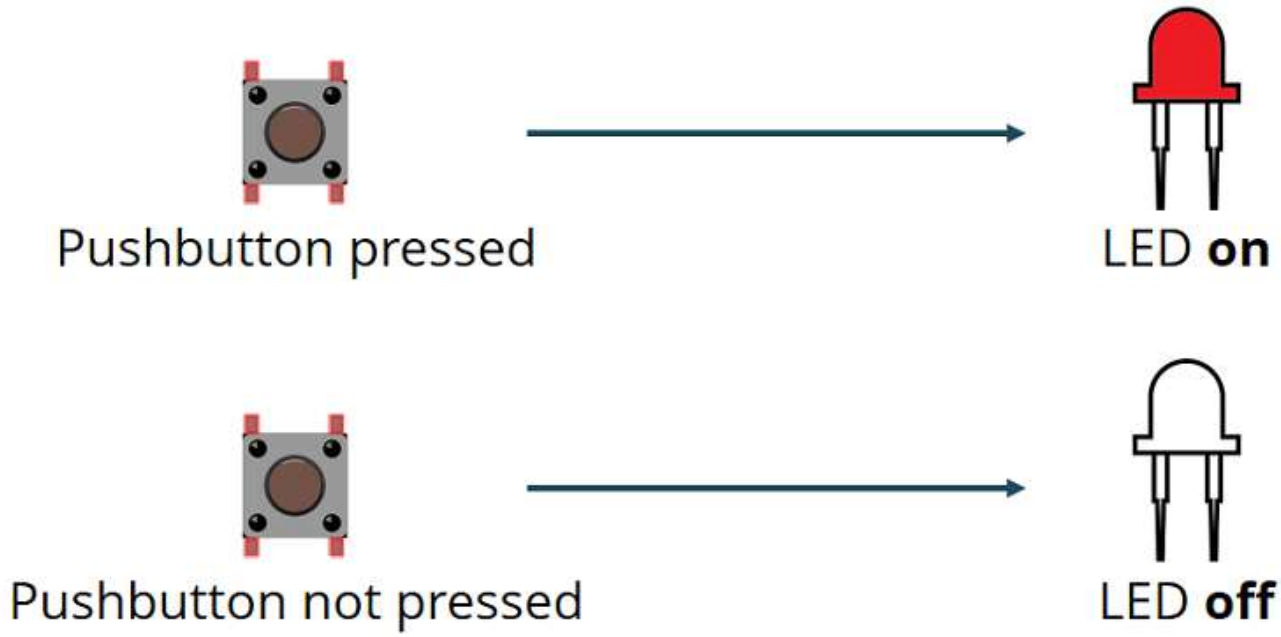
```
digitalRead(GPIO);
```

All ESP32 GPIOs can be used as inputs, except GPIOs 6 to 11 (connected to the integrated SPI flash).

Learn more about the ESP32 GPIOs: ESP32 GPIO Reference Guide

Project Example

To show you how to use digital inputs and digital outputs, we'll build a simple project example with a pushbutton and an LED. We'll read the state of the pushbutton and light up the LED accordingly as illustrated in the following figure.



Schematic Diagram

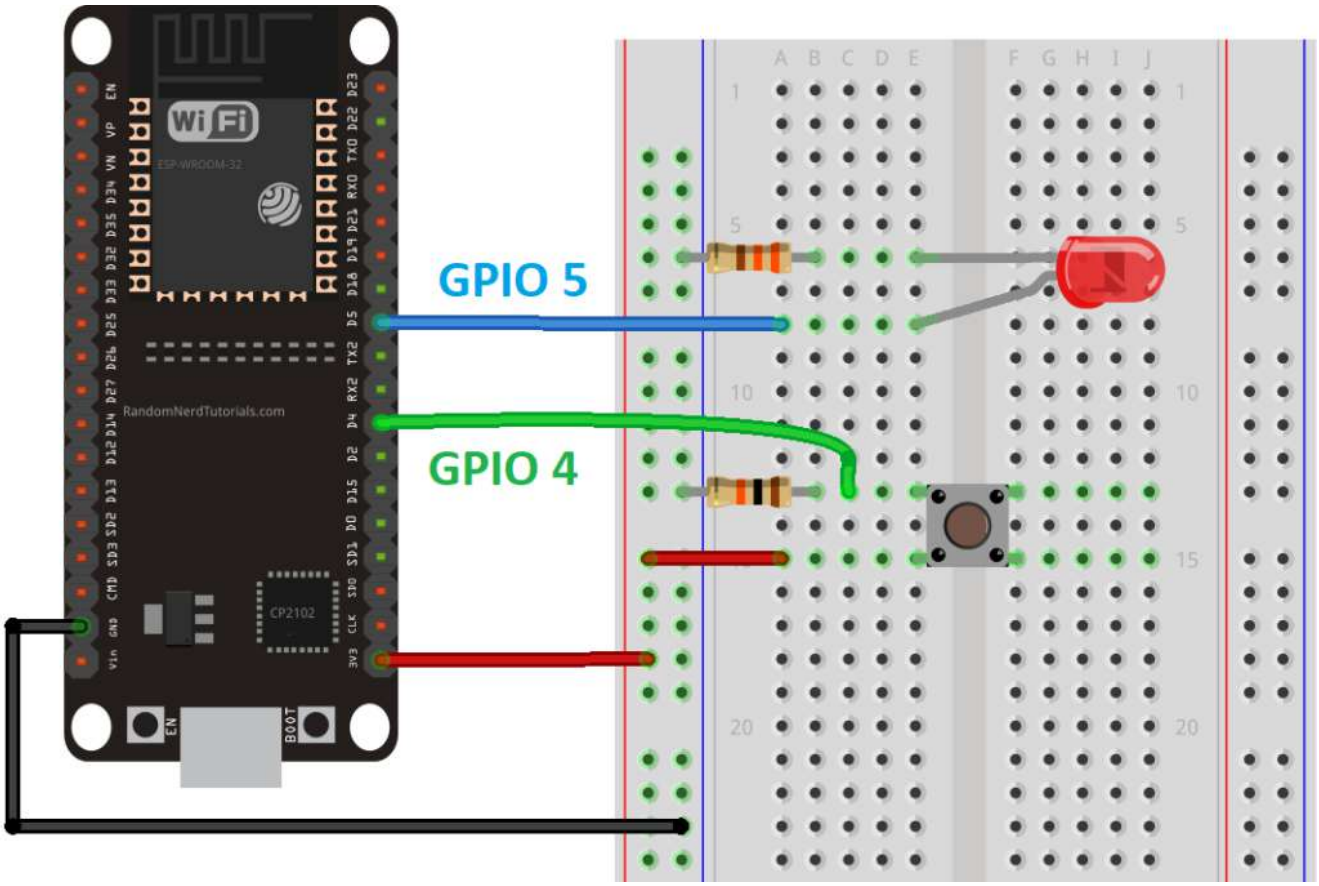
Before proceeding, you need to assemble a circuit with an LED and a pushbutton. We'll connect the LED to GPIO 5 and the pushbutton to GPIO 4 .

Parts Required

Here's a list of the parts to you need to build the circuit:

- ESP32 (read Best ESP32 Dev Boards)
- 5 mm LED
- 330 Ohm resistor
- Pushbutton
- 10k Ohm resistor
- Breadboard
- Jumper wires

You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!



Code

Copy the following code to your Arduino IDE.

```
// Complete Instructions: https://RandomNerdTutorials.com/esp32

// set pin numbers
const int buttonPin = 4;  // the number of the pushbutton pin
const int ledPin =  5;    // the number of the LED pin

// variable for storing the pushbutton status
int buttonState = 0;

void setup() {
  Serial.begin(115200);
  // initialize the pushbutton pin as an input
  pinMode(buttonPin, INPUT);
  // initialize the LED pin as an output
  pinMode(ledPin, OUTPUT);
}
```

```
void loop() {
  // read the state of the pushbutton value
  buttonState = digitalRead(buttonPin);
  Serial.println(buttonState);
  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH
  if (buttonState == HIGH) {
    // turn LED on
    digitalWrite(ledPin, HIGH);
  } else {
```

View raw code

How the Code Works

In the following two lines, you create variables to assign pins:

```
const int buttonPin = 4;
const int ledPin = 5;
```

The button is connected to GPIO 4 and the LED is connected to GPIO 5 . When using the Arduino IDE with the ESP32, 4 corresponds to GPIO 4 and 5 corresponds to GPIO 5 .

Next, you create a variable to hold the button state. By default, it's 0 (not pressed).

```
int buttonState = 0;
```

In the `setup()` , you initialize the button as an `INPUT` , and the LED as an `OUTPUT` . For that, you use the `pinMode()` function that accepts the pin you are referring to, and the mode: `INPUT` or `OUTPUT` .

```
pinMode(buttonPin, INPUT);
pinMode(ledPin, OUTPUT);
```

In the `loop()` is where you read the button state and set the LED accordingly.

In the next line, you read the button state and save it in the `buttonState` variable. As we've seen previously, you use the `digitalRead()` function.

```
buttonState = digitalRead(buttonPin);
```

The following if statement, checks whether the button state is `HIGH`. If it is, it turns the LED on using the `digitalWrite()` function that accepts as argument the `ledPin`, and the state `HIGH`.

```
if (buttonState == HIGH) {
  digitalWrite(ledPin, HIGH);
}
```

If the button state is not `HIGH`, you set the LED off. Just set `LOW` as a second argument to in the `digitalWrite()` function.

```
else {
  digitalWrite(ledPin, LOW);
}
```

Uploading the Code

Before clicking the upload button, go to **Tools** > **Board**, and select the board you're using. In my case, it's the DOIT ESP32 DEVKIT V1 board.

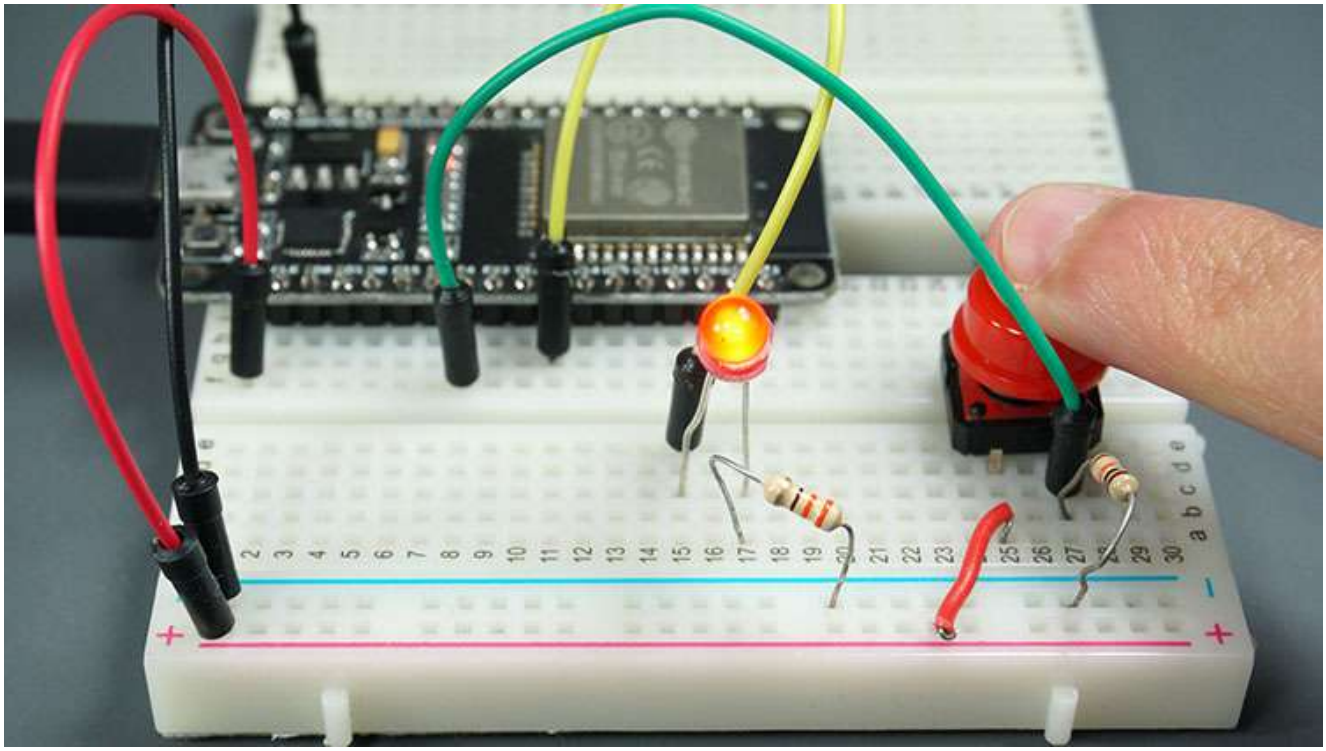
Go to **Tools** > **Port** and select the COM port the ESP32 is connected to. Then, press the upload button and wait for the "**Done uploading**" message.

If you see a lot of dots (…__…__) on the debugging window and the "Failed to connect to ESP32: Timed out waiting for packet header" message, that means

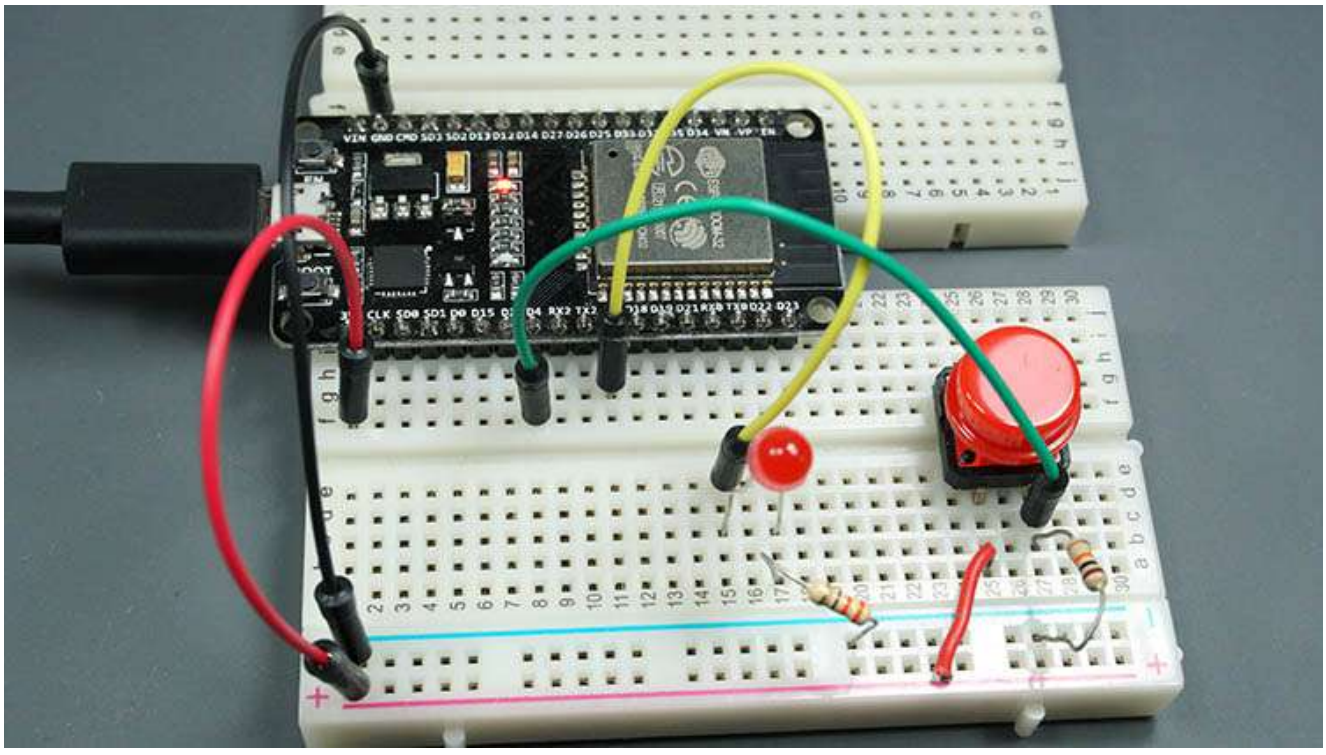
you need to press the ESP32 on-board BOOT button after the dots start appearing.

Demonstration

After uploading the code, test your circuit. Your LED should light up when you press the pushbutton:

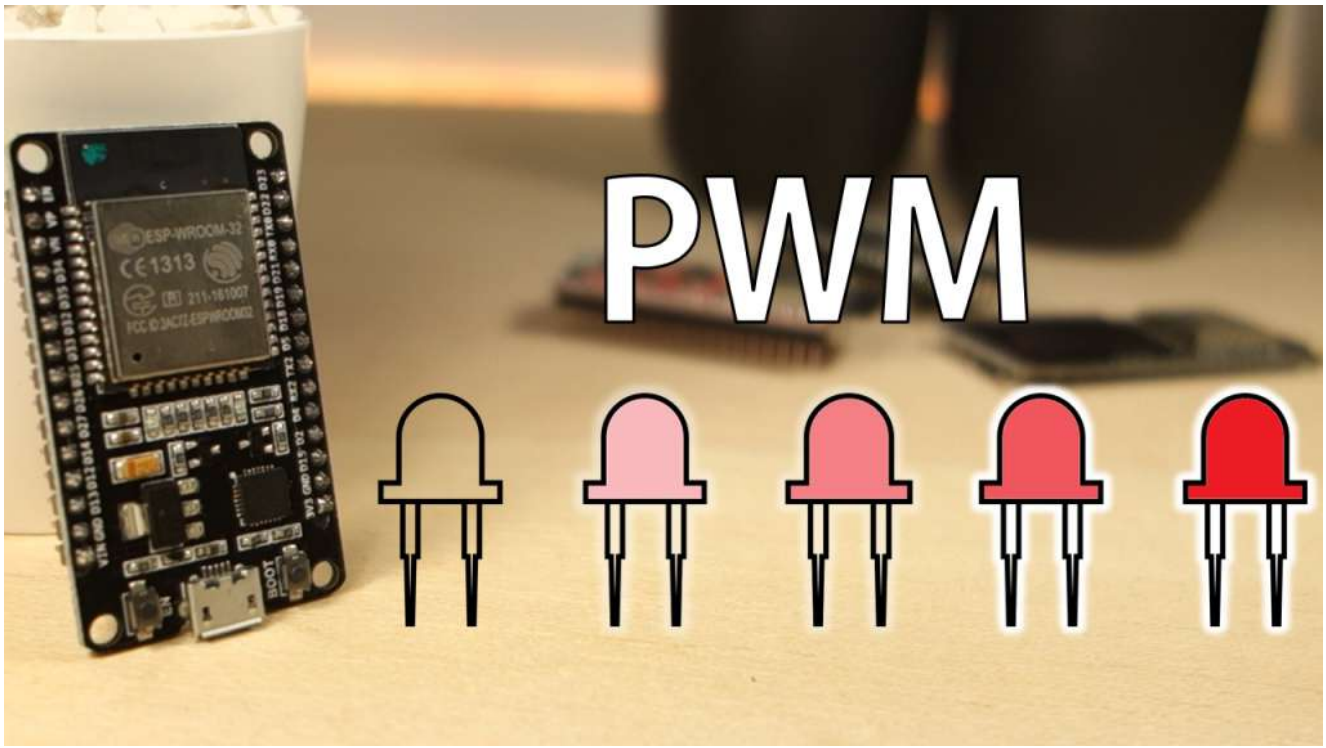


And turn off when you release it:



ESP32 PWM with Arduino IDE (Analog Output)

In this tutorial we'll show you how to generate PWM signals with the ESP32 using Arduino IDE. As an example we'll build a simple circuit that dims an LED using the LED PWM controller of the ESP32. We'll also show you how you can get the same PWM signal on different GPIOs at the same time.



Before proceeding with this tutorial you should have the ESP32 add-on installed in your Arduino IDE. Follow one of the following tutorials to install the ESP32 on the Arduino IDE, if you haven't already.

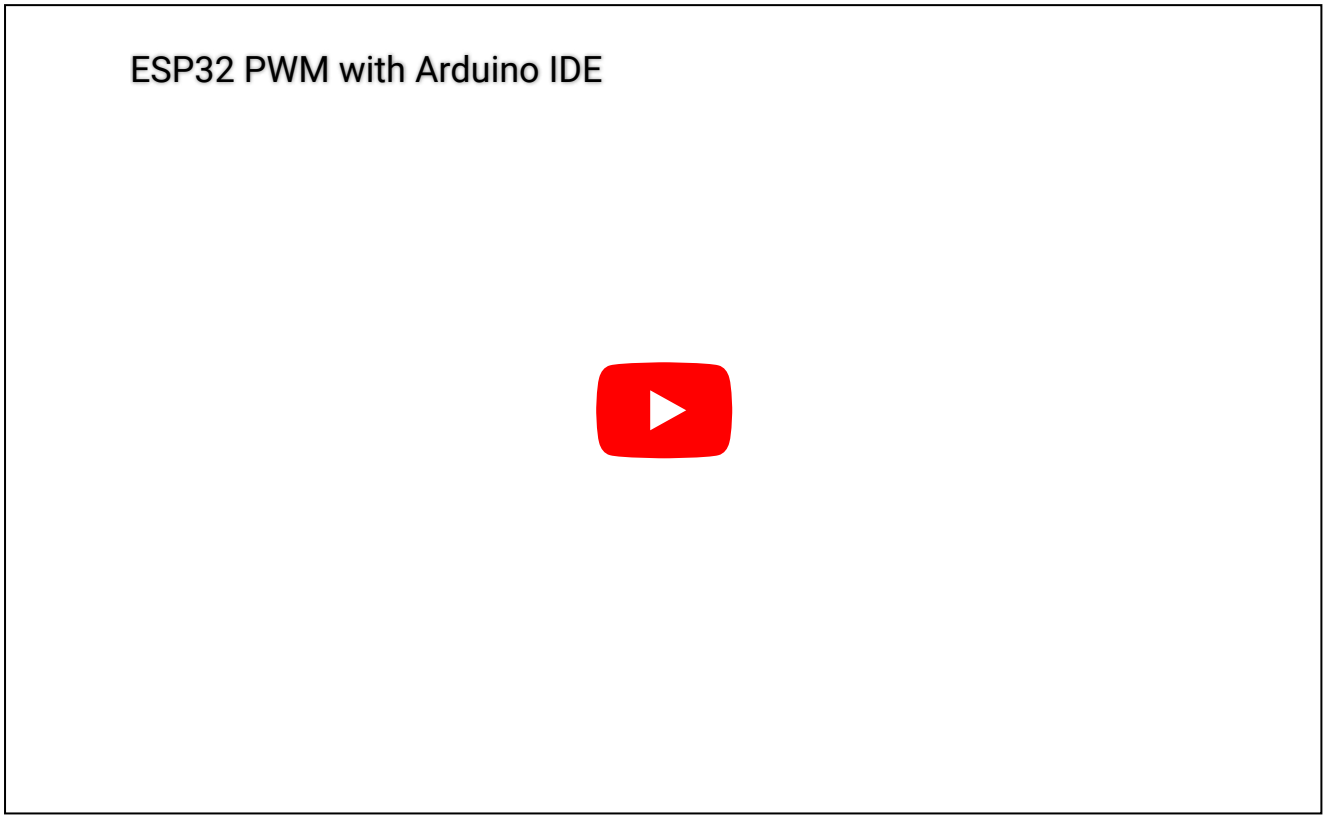
- Installing the ESP32 Board in Arduino IDE (Windows instructions)
- Installing the ESP32 Board in Arduino IDE (Mac and Linux instructions)

We also recommend taking a look at the following resources:

- Getting Started with ESP32 Dev Module
- ESP32 Pinout Reference: Which GPIO pins should you use?

Watch the Video Tutorial

This tutorial is available in video format (watch below) and in written format (continue reading).



Parts Required

To follow this tutorial you need these parts:

- ESP32 DOIT DEVKIT V1 Board – read best ESP32 development boards
- 3x 5mm LED
- 3x 330 Ohm resistor
- Breadboard
- Jumper wires

You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!

The ESP32 has a LED PWM controller with 16 independent channels that can be configured to generate PWM signals with different properties.

Here's the steps you'll have to follow to dim an LED with PWM using the Arduino IDE:

1. First, you need to choose a PWM channel. There are 16 channels from 0 to 15.

2. Then, you need to set the PWM signal frequency. For an LED, a frequency of 5000 Hz is fine to use.

3. You also need to set the signal's duty cycle resolution: you have resolutions from 1 to 16 bits. We'll use 8-bit resolution, which means you can control the LED brightness using a value from 0 to 255.

4. Next, you need to specify to which GPIO or GPIOs the signal will appear upon. For that you'll use the following function:

```
ledcAttachPin(GPIO, channel)
```

This function accepts two arguments. The first is the GPIO that will output the signal, and the second is the channel that will generate the signal.

5. Finally, to control the LED brightness using PWM, you use the following function:

```
ledcWrite(channel, dutycycle)
```

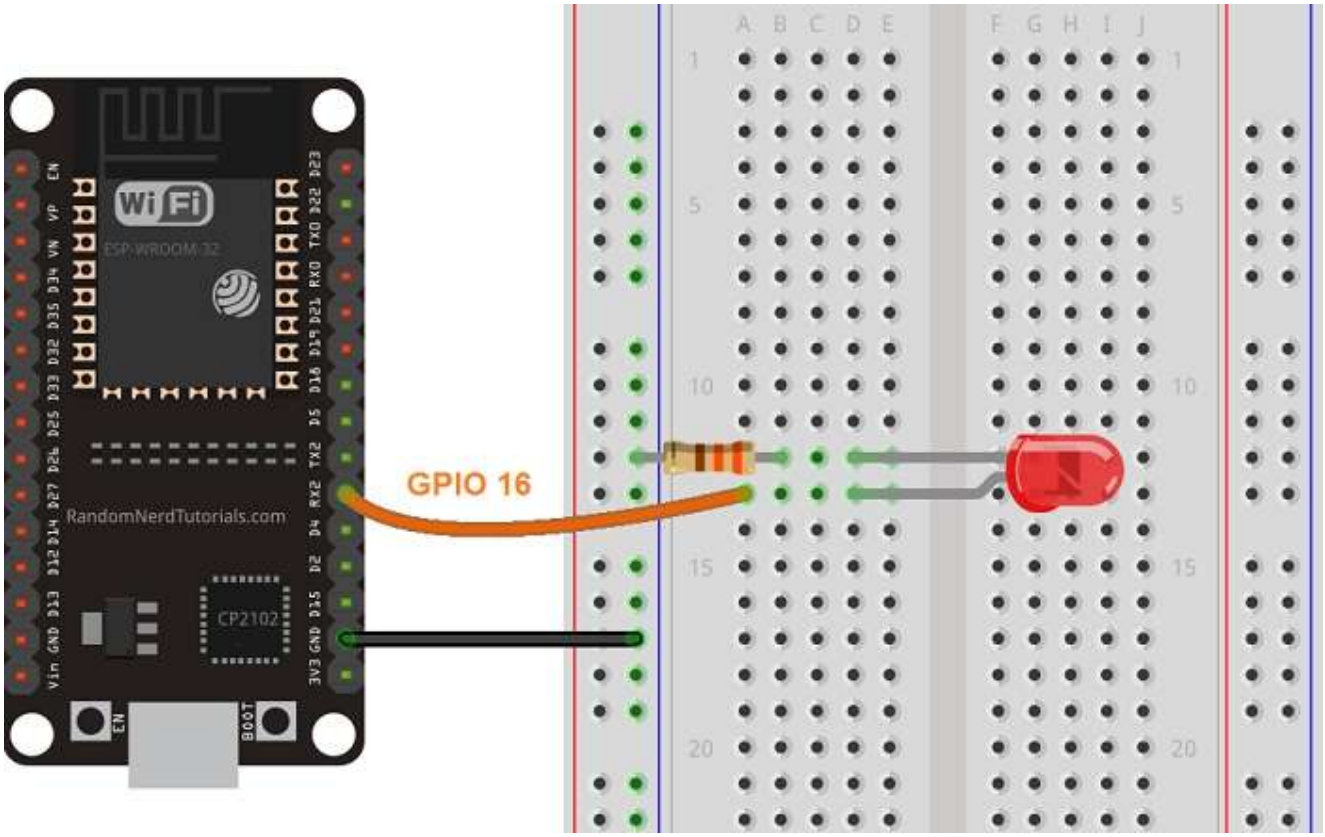
This function accepts as arguments the channel that is generating the PWM signal, and the duty cycle.

Dimming an LED

Let's see a simple example to see how to use the ESP32 LED PWM controller using the Arduino IDE.

Schematic

Wire an LED to your ESP32 as in the following schematic diagram. The LED should be connected to GPIO 16 .



(This schematic uses the ESP32 DEVKIT V1 module version with 30 GPIOs – if you're using another model, please check the pinout for the board you're using.)

> **Note:** you can use any pin you want, as long as it can act as an output. All pins that can act as outputs can be used as PWM pins. For more information about the ESP32 GPIOs, read: ESP32 Pinout Reference: Which GPIO pins should you use?

Code

Open your Arduino IDE and copy the following code.

```
// the number of the LED pin
const int ledPin = 16;  // 16 corresponds to GPIO16
```

```
const int ledChannel = 0;
const int resolution = 8;

void setup(){
  // configure LED PWM functionalitites
  ledcSetup(ledChannel, freq, resolution);

  // attach the channel to the GPIO to be controlled
  ledcAttachPin(ledPin, ledChannel);
}

void loop(){
  // increase the LED brightness
  for(int dutyCycle = 0; dutyCycle <= 255; dutyCycle++){
    // changing the LED brightness with PWM
    ledcWrite(ledChannel, dutyCycle);
    delay(15);
  }

  // decrease the LED brightness
  for(int dutyCycle = 255; dutyCycle >= 0; dutyCycle--){
    // changing the LED brightness with PWM
```

View raw code

You start by defining the pin the LED is attached to. In this case the LED is attached to GPIO 16 .

```
const int ledPin = 16;  // 16 corresponds to GPIO16
```

Then, you set the PWM signal properties. You define a frequency of 5000 Hz, choose channel 0 to generate the signal, and set a resolution of 8 bits. You can choose other properties, different than these, to generate different PWM signals.

```
const int freq = 5000;
```

```
const int resolution = 8;
```

In the `setup()`, you need to configure LED PWM with the properties you've defined earlier by using the `ledcSetup()` function that accepts as arguments, the ledChannel, the frequency, and the resolution, as follows:

```
ledcSetup(ledChannel, freq, resolution);
```

Next, you need to choose the GPIO you'll get the signal from. For that use the `ledcAttachPin()` function that accepts as arguments the GPIO where you want to get the signal, and the channel that is generating the signal. In this example, we'll get the signal in the `ledPin` GPIO, that corresponds to GPIO 16. The channel that generates the signal is the `ledChannel`, that corresponds to channel 0.

```
ledcAttachPin(ledPin, ledChannel);
```

In the loop, you'll vary the duty cycle between 0 and 255 to increase the LED brightness.

```
for(int dutyCycle = 0; dutyCycle <= 255; dutyCycle++){
    // changing the LED brightness with PWM
    ledcWrite(ledChannel, dutyCycle);
    delay(15);
}
```

And then, between 255 and 0 to decrease the brightness.

```
for(int dutyCycle = 255; dutyCycle >= 0; dutyCycle--){
    // changing the LED brightness with PWM
    ledcWrite(ledChannel, dutyCycle);
    delay(15);
```

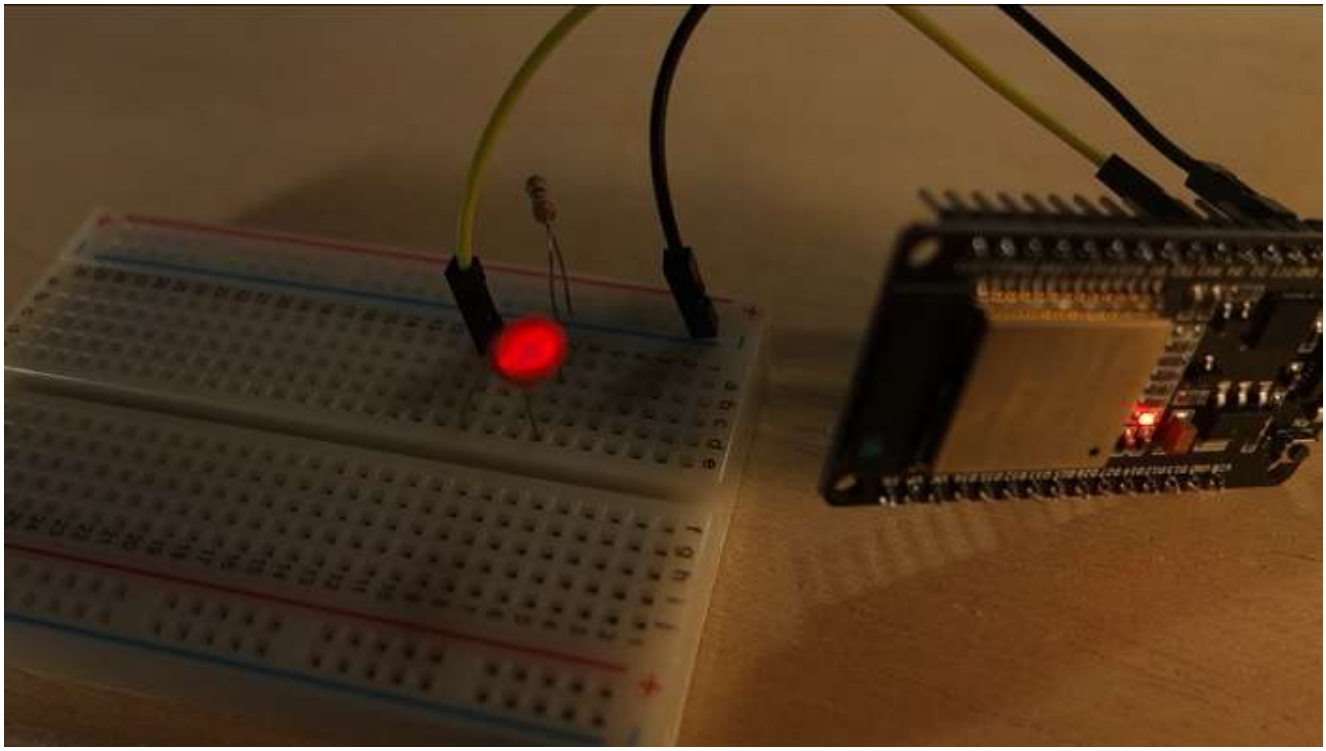
To set the brightness of the LED, you just need to use the `ledcWrite()` function that accepts as arguments the channel that is generating the signal, and the duty cycle.

```
ledcWrite(ledChannel, dutyCycle);
```

As we're using 8-bit resolution, the duty cycle will be controlled using a value from 0 to 255. Note that in the `ledcWrite()` function we use the channel that is generating the signal, and not the GPIO.

Testing the Example

Upload the code to your ESP32. Make sure you have the right board and COM port selected. Look at your circuit. You should have a dimmer LED that increases and decreases brightness.

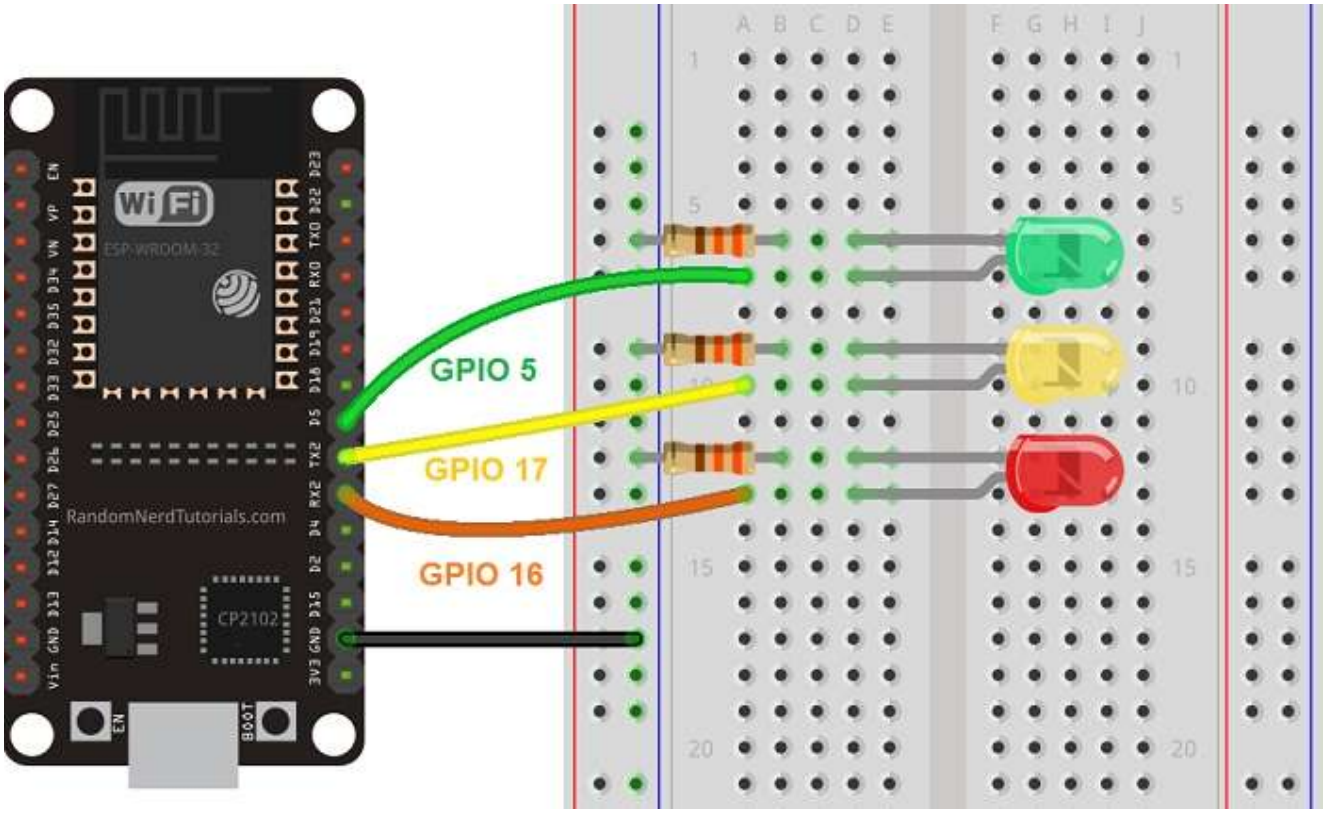


Getting the Same Signal on Different GPIOs

You can get the same signal from the same channel in different GPIOs. To achieve that, you just need to attach those GPIOs to the same channel on the `setup()`.

Schematic

Add two more LEDs to your circuit by following the next schematic diagram:



(This schematic uses the ESP32 DEVKIT V1 module version with 30 GPIOs – if you're using another model, please check the pinout for the board you're using.)

Code

Copy the following code to your Arduino IDE.

```
// the number of the LED pin
const int ledPin = 16;  // 16 corresponds to GPIO16
const int ledPin2 = 17; // 17 corresponds to GPIO17
const int ledPin3 = 5;  // 5 corresponds to GPIO5

// setting PWM properties
const int freq = 5000;
const int ledChannel = 0;
const int resolution = 8;
```

```
  // configure LED PWM functionalitites
  ledcSetup(ledChannel, freq, resolution);

  // attach the channel to the GPIO to be controlled
  ledcAttachPin(ledPin, ledChannel);
  ledcAttachPin(ledPin2, ledChannel);
  ledcAttachPin(ledPin3, ledChannel);
}

void loop(){
  // increase the LED brightness
  for(int dutyCycle = 0; dutyCycle <= 255; dutyCycle++){
    // changing the LED brightness with PWM
    ledcWrite(ledChannel, dutyCycle);
    delay(15);
  }
```

View raw code

This is the same code as the previous one but with some modifications. We've defined two more variables for two new LEDs, that refer to GPIO 17 and GPIO 5 .

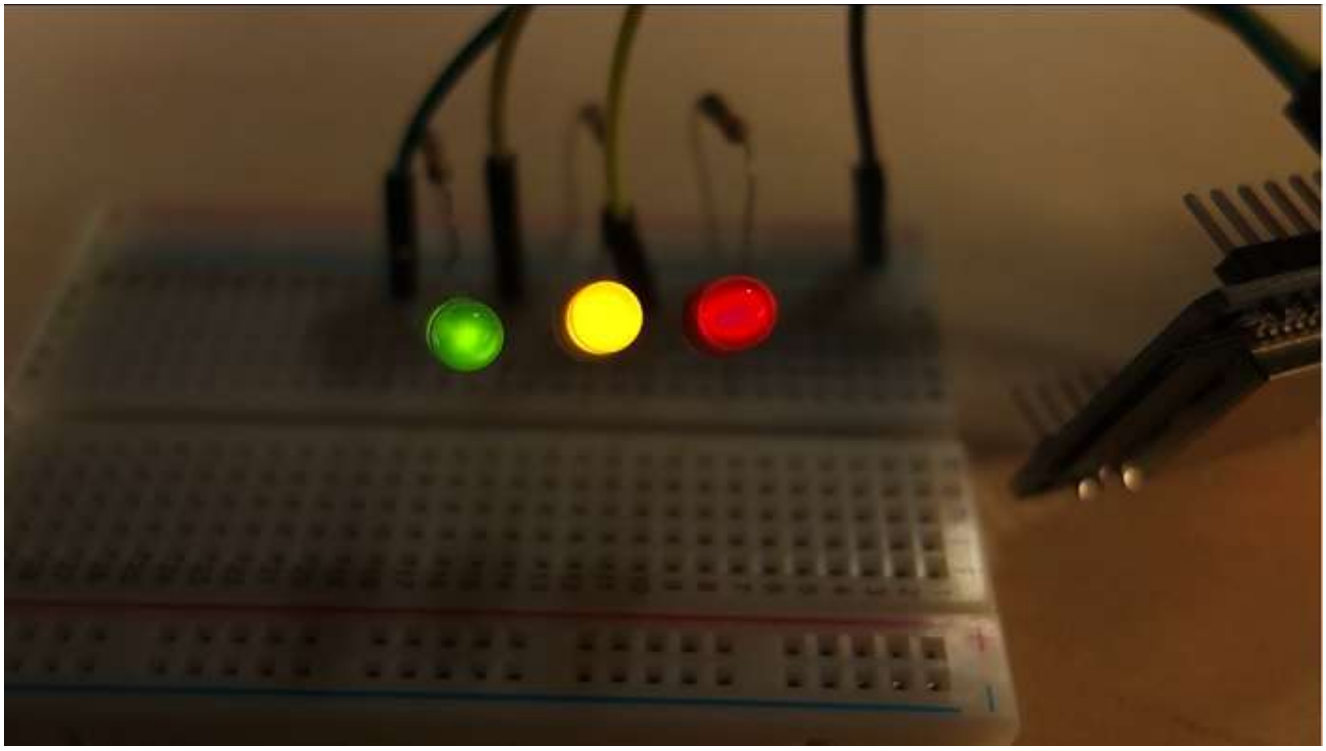
```
const int ledPin2 = 17; // 17 corresponds to GPIO17
const int ledPin3 = 5;  // 5 corresponds to GPIO5
```

Then, in the `setup()` , we've added the following lines to assign both GPIOs to channel 0. This means that we'll get the same signal, that is being generated on channel 0, on both GPIOs.

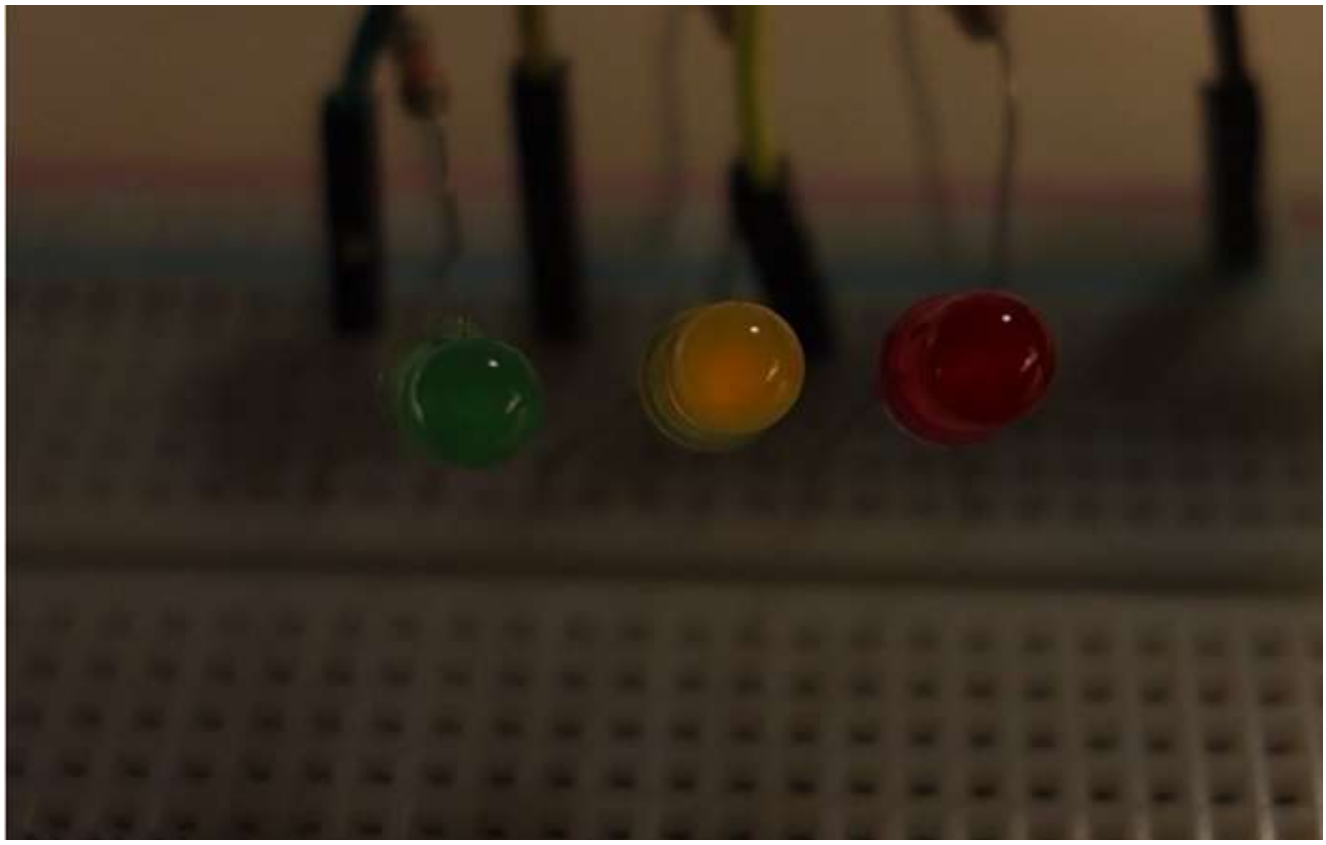
```
ledcAttachPin(ledPin2, ledChannel);
ledcAttachPin(ledPin3, ledChannel);
```

Testing the Project

Upload the new sketch to your ESP32. Make sure you have the right board and



All GPIOs are outputting the same PWM signal. So, all three LEDs increase and decrease the brightness simultaneously, resulting in a synchronized effect.

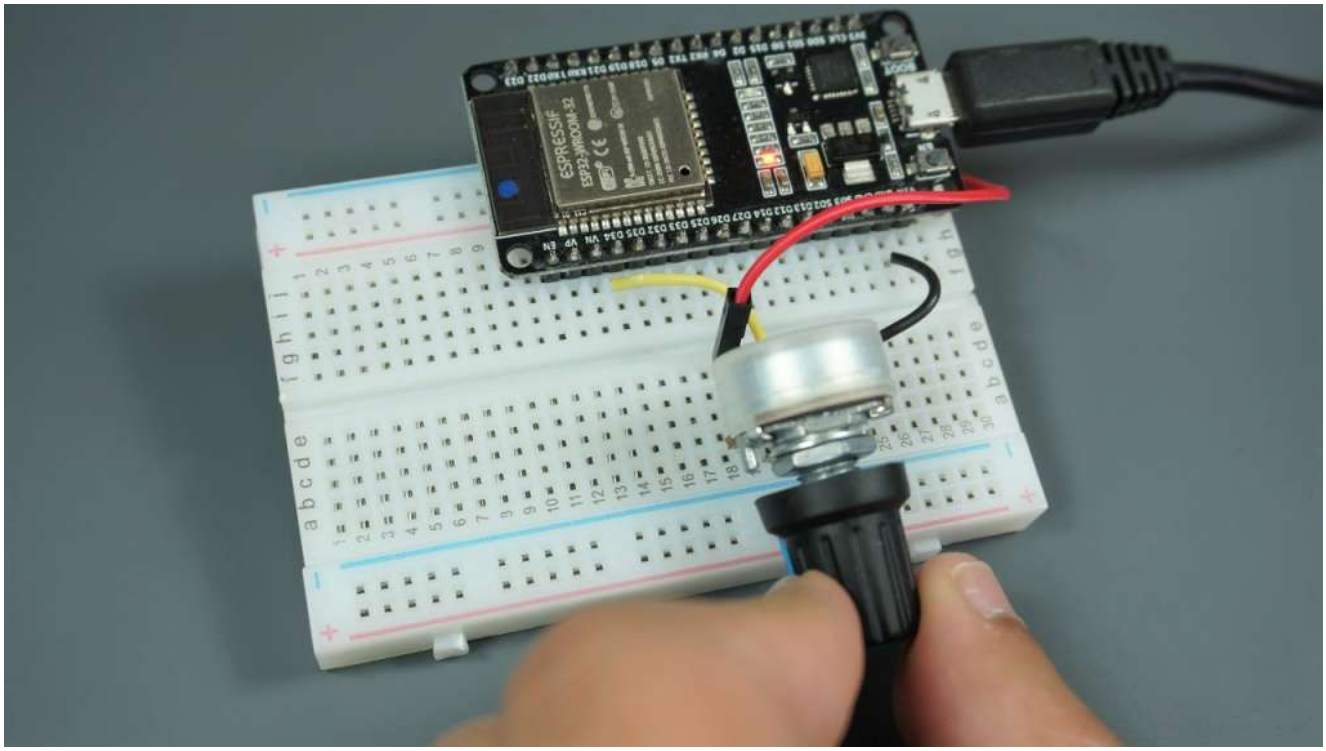


Wrapping Up

In summary, in this post you've learned how to use the LED PWM controller of the

ESP32 ADC – Read Analog Values with Arduino IDE

This article shows how to read analog inputs with the ESP32 using Arduino IDE. Analog reading is useful to read values from variable resistors like potentiometers, or analog sensors.



> Reading analog inputs with the ESP32 is as easy as using the `analogRead(GPIO)` function, that accepts as argument, the GPIO you want to read.

We also have other tutorials on how to use analog pins with ESP board:

- ESP8266 ADC – Read Analog Values with Arduino IDE, MicroPython and Lua
- ESP32 Analog Readings with MicroPython

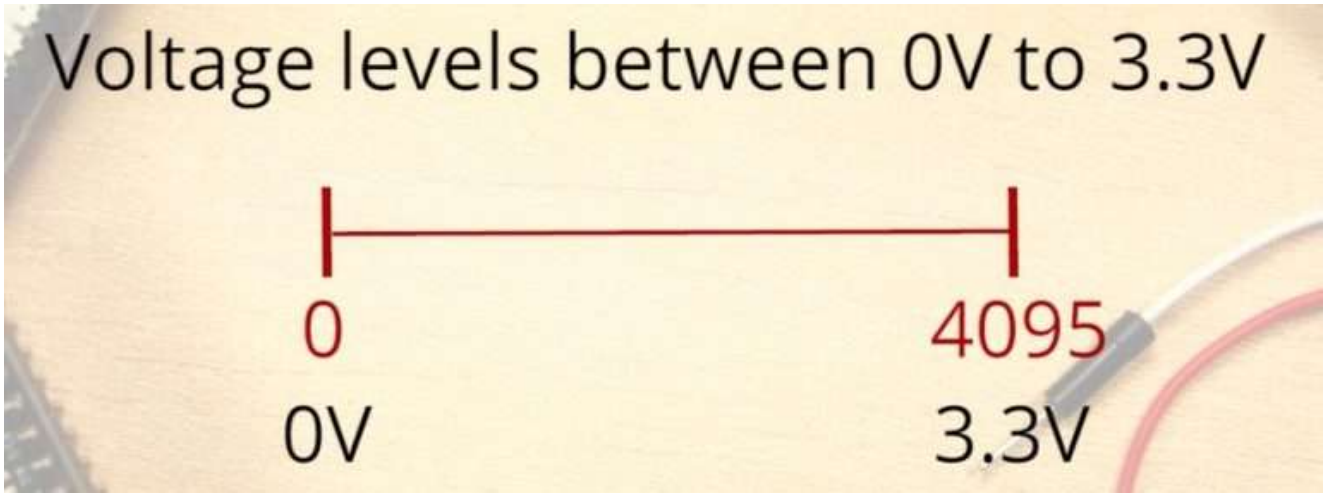
Watch the Video

ESP32 ADC – Read Analog Values with Arduino IDE

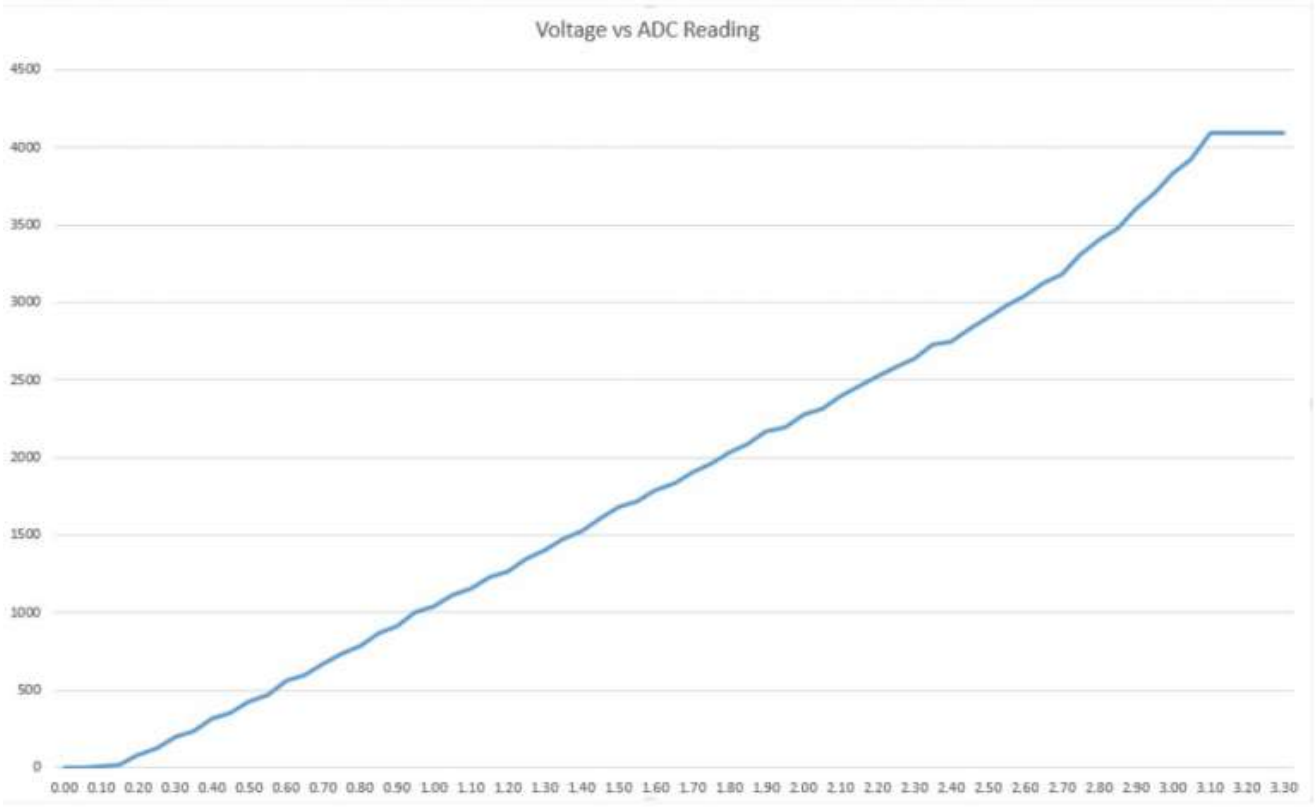
Analog Inputs (ADC)

Reading an analog value with the ESP32 means you can measure varying voltage levels between 0 V and 3.3 V.

The voltage measured is then assigned to a value between 0 and 4095, in which 0 V corresponds to 0, and 3.3 V corresponds to 4095. Any voltage between 0 V and 3.3 V will be given the corresponding value in between.



However, that doesn't happen. What you'll get is a behavior as shown in the following chart:



View source

This behavior means that your ESP32 is not able to distinguish 3.3 V from 3.2 V. You'll get the same value for both voltages: 4095.

The same happens for very low voltage values: for 0 V and 0.1 V you'll get the same value: 0. You need to keep this in mind when using the ESP32 ADC pins.

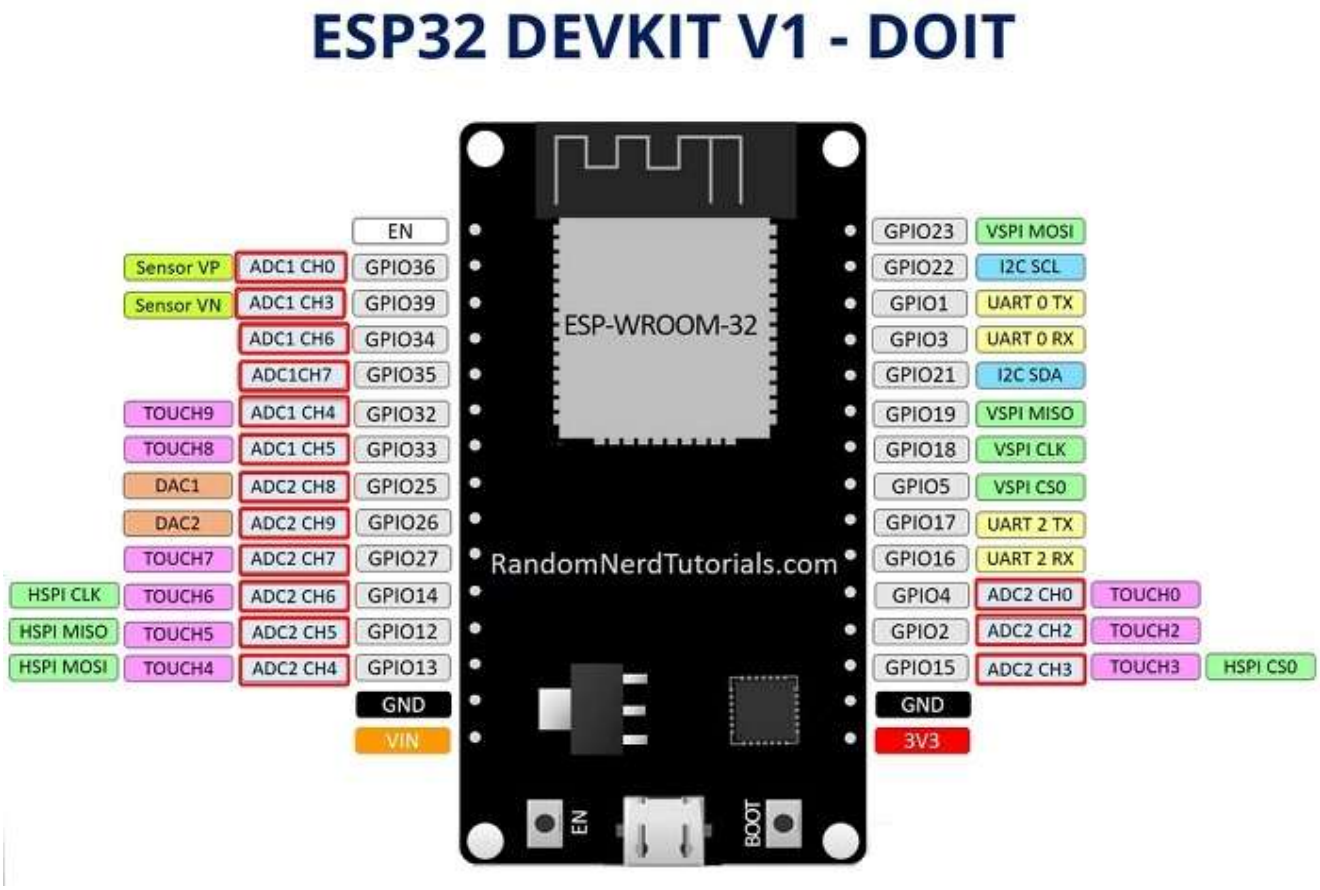
There's a discussion on GitHub about this subject.

analogRead() Function

Reading an analog input with the ESP32 using the Arduino IDE is as simple as using the `analogRead()` function. It accepts as argument, the GPIO you want to read:

```
analogRead(GPIO);
```

Grab your ESP32 board pinout and locate the ADC pins. These are highlighted with a red border in the figure below.



Learn more about the ESP32 GPIOs: ESP32 Pinout Reference.

These analog input pins have 12-bit resolution. This means that when you read an analog input, its range may vary from 0 to 4095.

> **Note:** ADC2 pins cannot be used when Wi-Fi is used. So, if you're using Wi-Fi and you're having trouble getting the value from an ADC2 GPIO, you may consider using an ADC1 GPIO instead, that should solve your problem.

Other Useful Functions

There are other more advanced functions to use with the ADC pins that can be useful in other projects.

Default is 12-bit resolution.

- `analogSetWidth(width)` : set the sample bits and resolution. It can be a value between 9 (0 – 511) and 12 bits (0 – 4095). Default is 12-bit resolution.
- `analogSetCycles(cycles)` : set the number of cycles per sample. Default is 8. Range: 1 to 255.
- `analogSetSamples(samples)` : set the number of samples in the range. Default is 1 sample. It has an effect of increasing sensitivity.
- `analogSetClockDiv(attenuation)` : set the divider for the ADC clock. Default is 1. Range: 1 to 255.
- `analogSetAttenuation(attenuation)` : sets the input attenuation for all ADC pins. Default is `ADC_11db` . Accepted values:
  - `ADC_0db` : sets no attenuation. ADC can measure up to approximately 800 mV (1V input = ADC reading of 1088).
  - `ADC_2_5db` : The input voltage of ADC will be attenuated, extending the range of measurement to up to approx. 1100 mV. (1V input = ADC reading of 3722).
  - `ADC_6db` : The input voltage of ADC will be attenuated, extending the range of measurement to up to approx. 1350 mV. (1V input = ADC reading of 3033).
  - `ADC_11db` : The input voltage of ADC will be attenuated, extending the range of measurement to up to approx. 2600 mV. (1V input = ADC reading of 1575).
- `analogSetPinAttenuation(pin, attenuation)` : sets the input attenuation for the specified pin. The default is `ADC_11db` . Attenuation values are the same from previous function.
- `adcAttachPin(pin)` : Attach a pin to ADC (also clears any other analog mode that could be on). Returns TRUE or FALSE result.
- `adcStart(pin )`, `adcBusy(pin)` and `resultadcEnd(pin)` : starts an ADC convertion on attached pin's bus. Check if conversion on the pin's ADC bus is currently running (returns TRUE or FALSE). Get the result of the conversion: returns 16-bit integer.

There is a very good video explaining these functions that you can watch here.

To see how everything ties together, we'll make a simple example to read an analog value from a potentiometer.

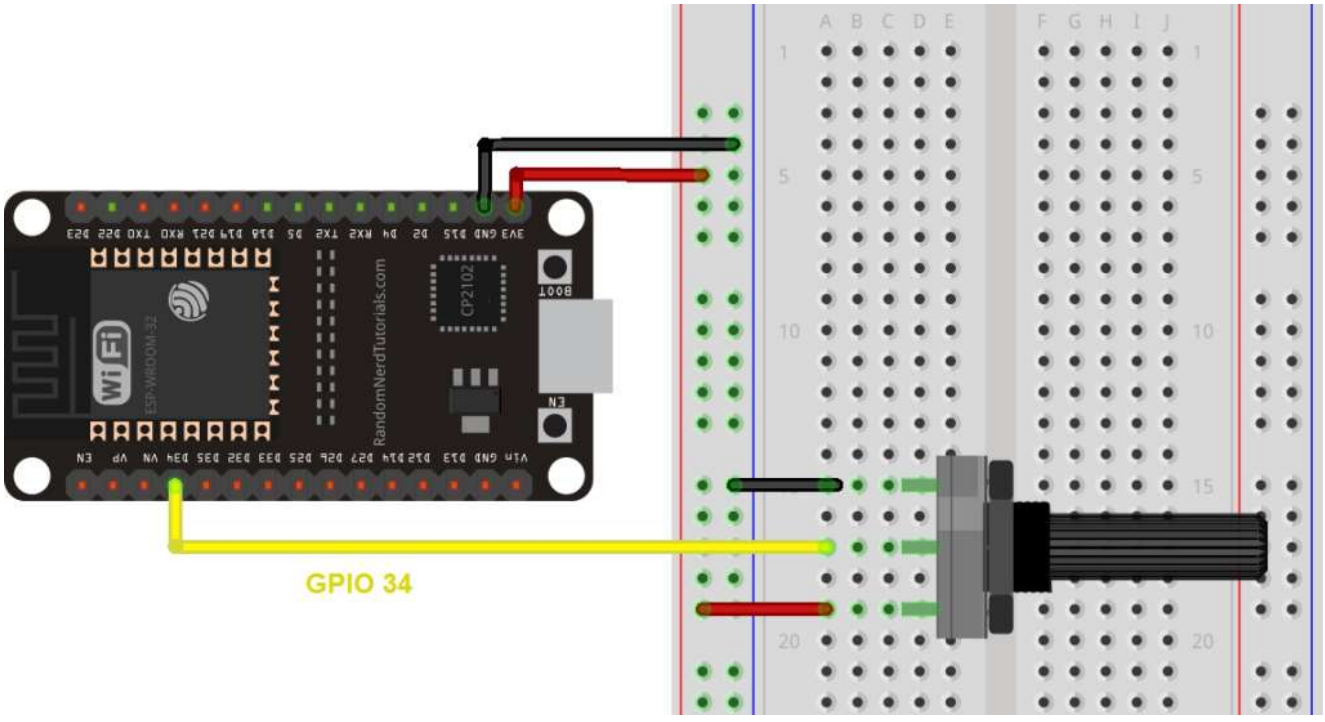
For this example, you need the following parts:

- ESP32 DOIT DEVKIT V1 Board (read Best ESP32 development boards)
- Potentiometer
- Breadboard
- Jumper wires

You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!

Schematic

Wire a potentiometer to your ESP32. The potentiometer middle pin should be connected to GPIO 34. You can use the following schematic diagram as a reference.



add-on installed before proceeding:

- **Windows** instructions – ESP32 Board in Arduino IDE
- **Mac and Linux** instructions – ESP32 Board in Arduino IDE

Open your Arduino IDE and copy the following code.

```
// Potentiometer is connected to GPIO 34 (Analog ADC1_CH6)
const int potPin = 34;

// variable for storing the potentiometer value
int potValue = 0;

void setup() {
  Serial.begin(115200);
  delay(1000);
}

void loop() {
  // Reading potentiometer value
  potValue = analogRead(potPin);
  Serial.println(potValue);
  delay(500);
}
```

View raw code

This code simply reads the values from the potentiometer and prints those values in the Serial Monitor.

In the code, you start by defining the GPIO the potentiometer is connected to. In this example, GPIO 34 .

```
const int potPin = 34;
```

```
Serial.begin(115200);
```

In the `loop()`, use the `analogRead()` function to read the analog input from the `potPin`.

```
potValue = analogRead(potPin);
```

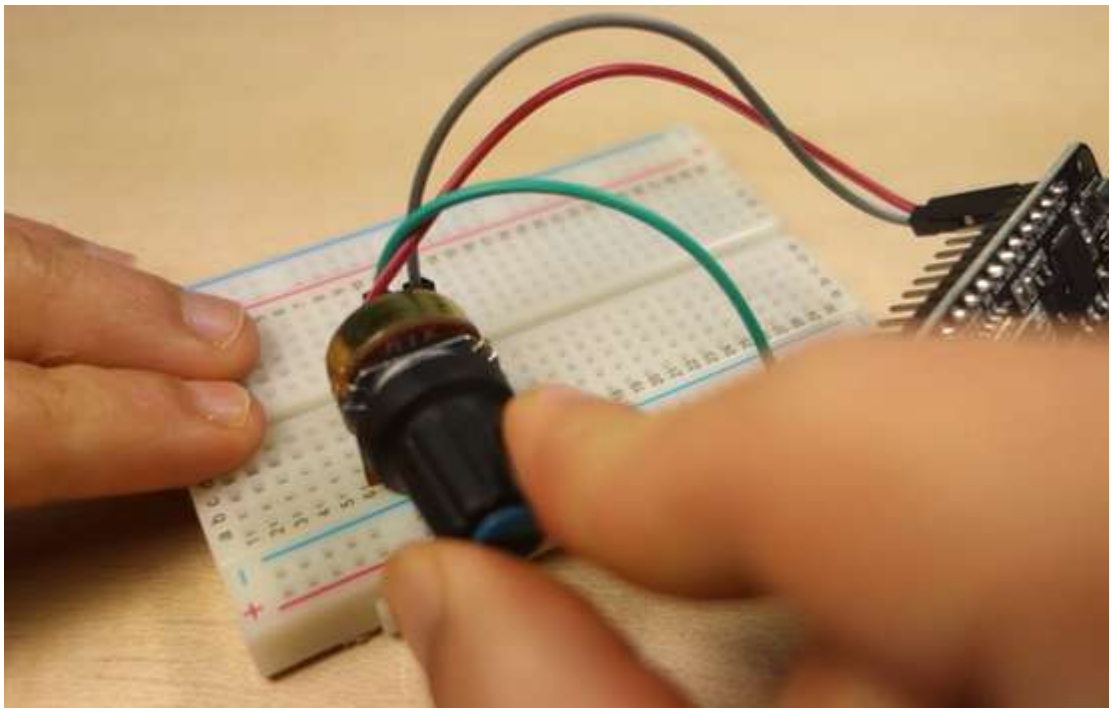
Finally, print the values read from the potentiometer in the serial monitor.

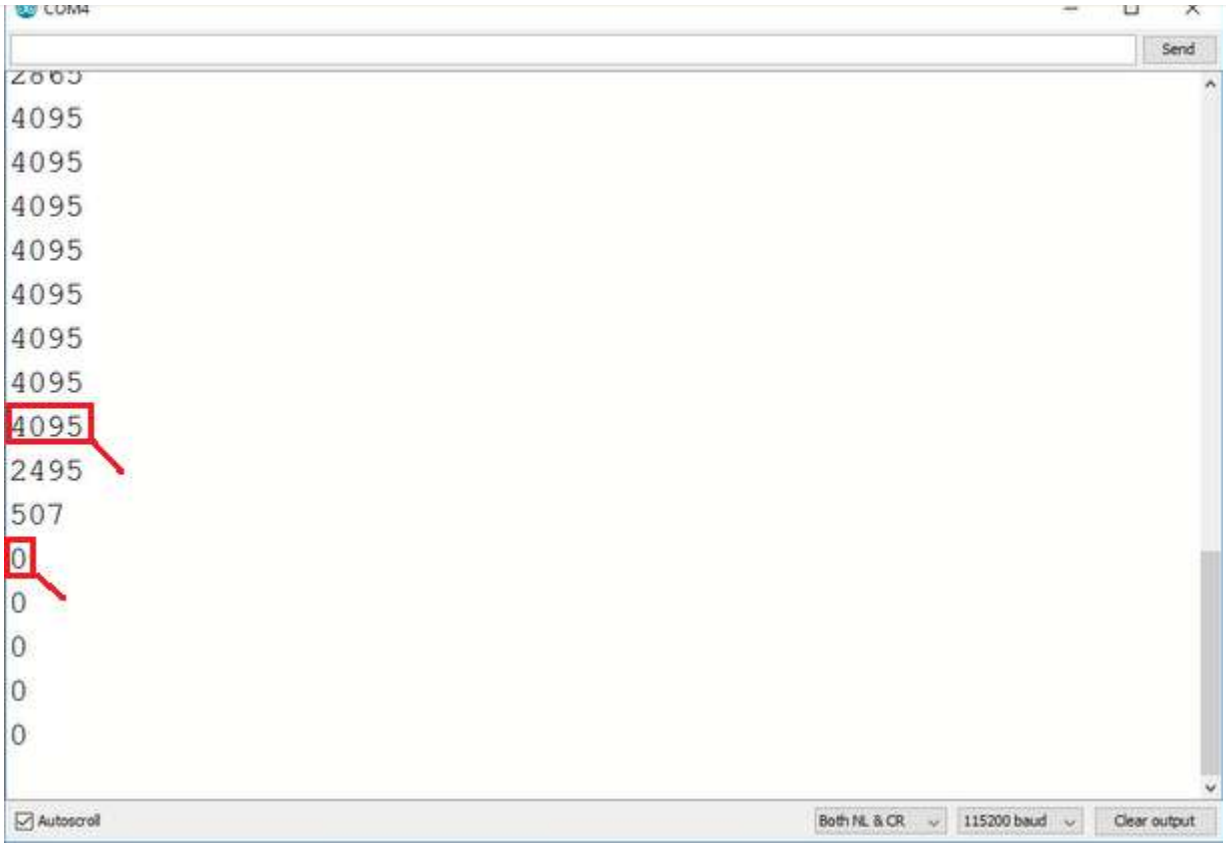
```
Serial.println(potValue);
```

Upload the code provided to your ESP32. Make sure you have the right board and COM port selected in the Tools menu.

Testing the Example

After uploading the code and pressing the ESP32 reset button, open the Serial Monitor at a baud rate of 115200. Rotate the potentiometer and see the values changing.





Wrapping Up

In this article you've learned how to read analog inputs using the ESP32 with the Arduino IDE. In summary:

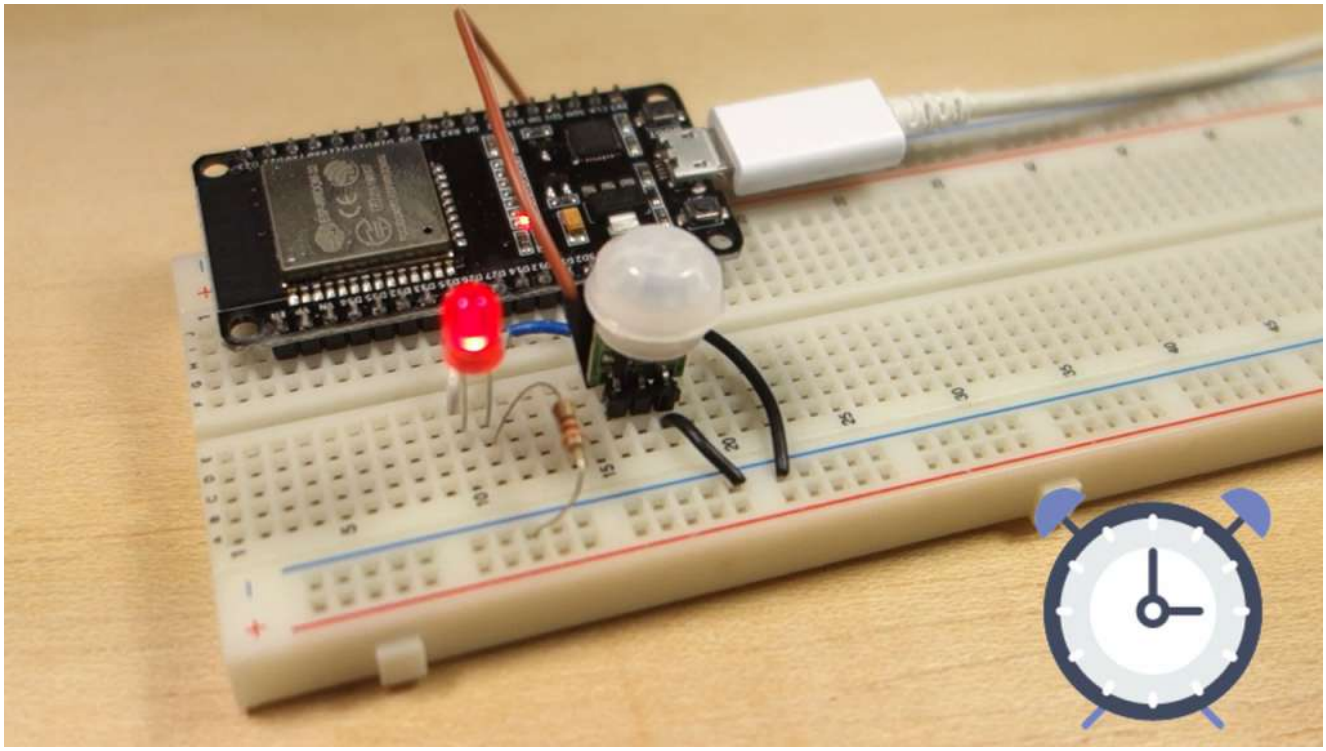
- The ESP32 DEVKIT V1 DOIT board (version with 30 pins) has 15 ADC pins you can use to read analog inputs.
- These pins have a resolution of 12 bits, which means you can get values from 0 to 4095.
- To read a value in the Arduino IDE, you simply use the `analogRead()` function.
- The ESP32 ADC pins don't have a linear behavior. You'll probably won't be able to distinguish between 0 and 0.1V, or between 3.2 and 3.3V. You need to keep that in mind when using the ADC pins.

We hope you've find this short guide useful. If you want to learn more about the ESP32, enroll in our course: Learn ESP32 with Arduino IDE.

Other ESP32 guides that you may also like:

ESP32 with PIR Motion Sensor using Interrupts and Timers

This tutorial shows how to detect motion with the ESP32 using a PIR motion sensor. In this example, when motion is detected (an interrupt is triggered), the ESP32 starts a timer and turns an LED on for a predefined number of seconds. When the timer finishes counting down, the LED is automatically turned off.



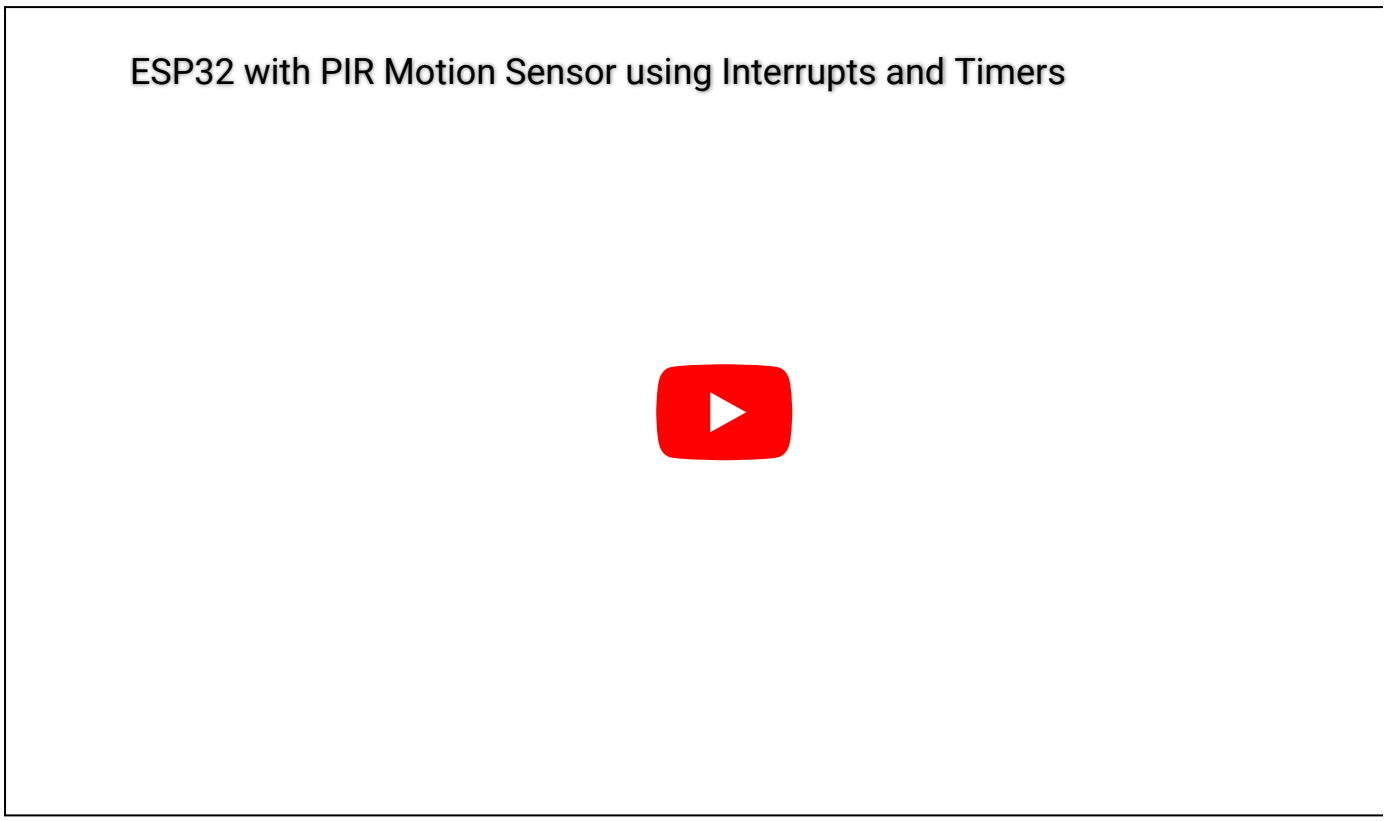
With this example we'll also explore two important concepts: interrupts and timers.

Before proceeding with this tutorial you should have the ESP32 add-on installed in your Arduino IDE. Follow one of the following tutorials to install the ESP32 on the Arduino IDE, if you haven't already.

- Installing the ESP32 Board in Arduino IDE (Windows instructions)
- Installing the ESP32 Board in Arduino IDE (Mac and Linux instructions)

Watch the Video Tutorial and Project Demo

This tutorial is available in video format (watch below) and in written format (continue reading).



Parts Required

To follow this tutorial you need the following parts

- ESP32 DOIT DEVKIT V1 Board – read ESP32 Development Boards Review and Comparison
- Mini PIR motion sensor (AM312) or PIR motion sensor (HC-SR501)
- 5mm LED
- 330 Ohm resistor
- Jumper wires
- Breadboard

You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!

Introducing Interrupts

To trigger an event with a PIR motion sensor, you use interrupts. Interrupts are useful for making things happen automatically in microcontroller programs, and can help solve timing problems.

With interrupts you don't need to constantly check the current value of a pin. With interrupts, when a change is detected, an event is triggered (a function is called).

To set an interrupt in the Arduino IDE, you use the **`attachInterrupt()`** function, that accepts as arguments: the GPIO pin, the name of the function to be executed, and mode:

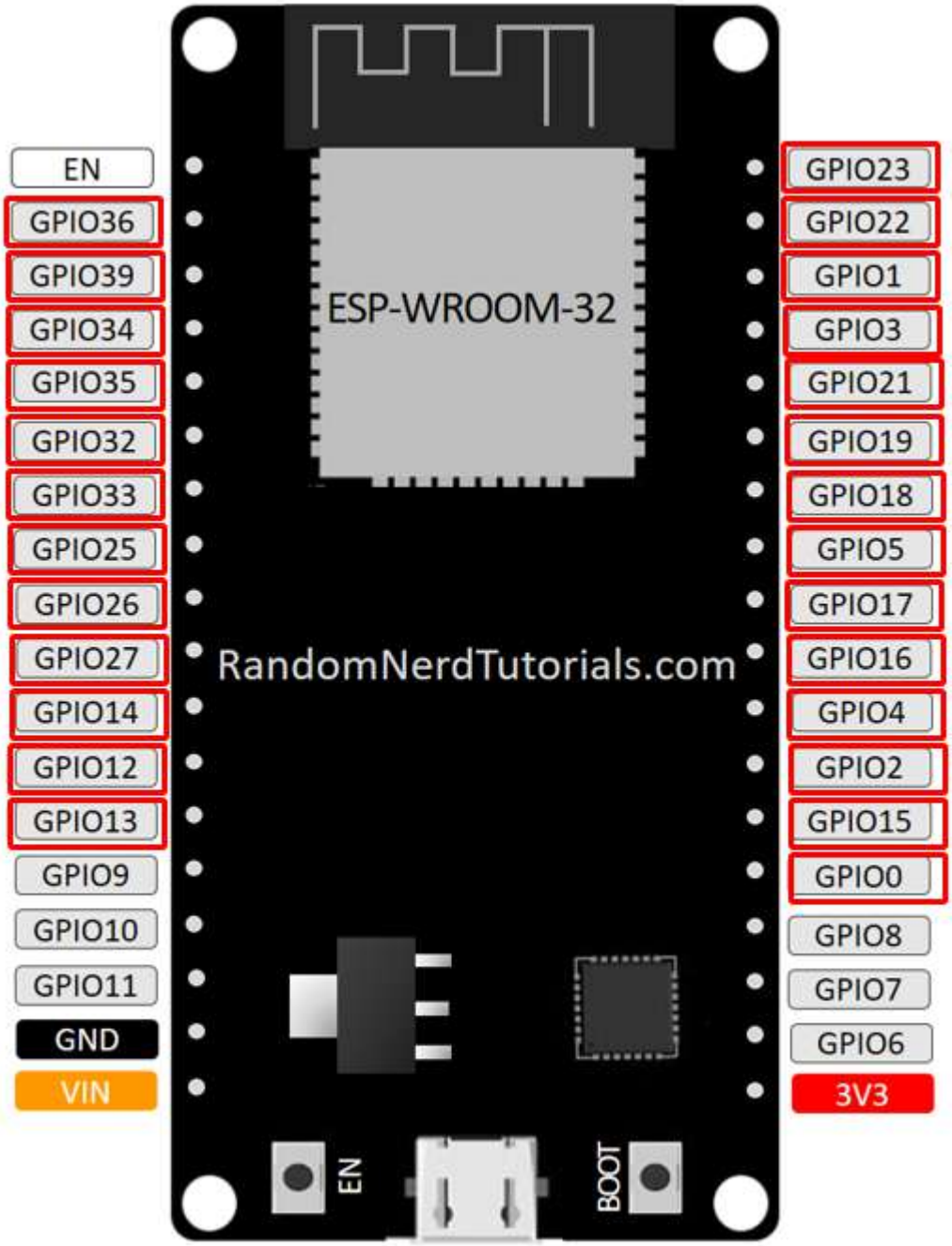
```
attachInterrupt(digitalPinToInterrupt(GPIO), function, mode);
```

GPIO Interrupt

The first argument is a GPIO number. Normally, you should use `digitalPinToInterrupt(GPIO)` to set the actual GPIO as an interrupt pin. For example, if you want to use GPIO 27 as an interrupt, use:

```
digitalPinToInterrupt(27)
```

With an ESP32 board, all the pins highlighted with a red rectangle in the following figure can be configured as interrupt pins. In this example we'll use GPIO 27 as an interrupt connected to the PIR Motion sensor.



Function to be triggered

The second argument of the `attachInterrupt()` function is the name of the function that will be called every time the interrupt is triggered.

Mode

The third argument is the mode. There are 5 different modes:

- `LOW: to trigger the interrupt whenever the pin is LOW;`
- `HIGH` : to trigger the interrupt whenever the pin is HIGH;
- `CHANGE` : to trigger the interrupt whenever the pin changes value – for example from HIGH to LOW or LOW to HIGH;

- `FALLING` : for when the pin goes from HIGH to LOW;
- `RISING` : to trigger when the pin goes from LOW to HIGH.

For this example will be using the RISING mode, because when the PIR motion sensor detects motion, the GPIO it is connected to goes from LOW to HIGH.

Introducing Timers

In this example we'll also introduce timers. We want the LED to stay on for a predetermined number of seconds after motion is detected. Instead of using a `delay()` function that blocks your code and doesn't allow you to do anything else for a determined number of seconds, we should use a timer.

The delay() function

You should be familiar with the `delay()` function as it is widely used. This function is pretty straightforward to use. It accepts a single int number as an argument. This number represents the time in milliseconds the program has to wait until moving on to the next line of code.

```
delay(time in milliseconds)
```

When you do `delay(1000)` your program stops on that line for 1 second.

`delay()` is a blocking function. Blocking functions prevent a program from doing anything else until that particular task is completed. If you need multiple tasks to occur at the same time, you cannot use `delay()` .

For most projects you should avoid using delays and use timers instead.

The millis() function

Using a function called `millis()` you can return the number of milliseconds that have passed since the program first started.

```
millis()
```

Why is that function useful? Because by using some math, you can easily verify how much time has passed without blocking your code.

Blinking an LED with millis()

The following snippet of code shows how you can use the `millis()` function to create a blink LED project. It turns an LED on for 1000 milliseconds, and then turns it off.

```
/*********
  Rui Santos
  Complete project details at https://randomnerdtutorials.com
*********/

// constants won't change. Used here to set a pin number :
const int ledPin =  26;      // the number of the LED pin

// Variables will change :
int ledState = LOW;             // ledState used to set the LED

// Generally, you should use "unsigned long" for variables that
// The value will quickly become too large for an int to store
unsigned long previousMillis = 0;        // will store last tim

// constants won't change :
const long interval = 1000;           // interval at which to b

void setup() {
  // set the digital pin as output:
  pinMode(ledPin, OUTPUT);
}

void loop() {
```

```
  // here is where you'd put code that needs to be running all

  // check to see if it's time to blink the LED; that is  if th
```

View raw code

How the code works

Let's take a closer look at this blink sketch that works without a `delay()` function (it uses the `millis()` function instead).

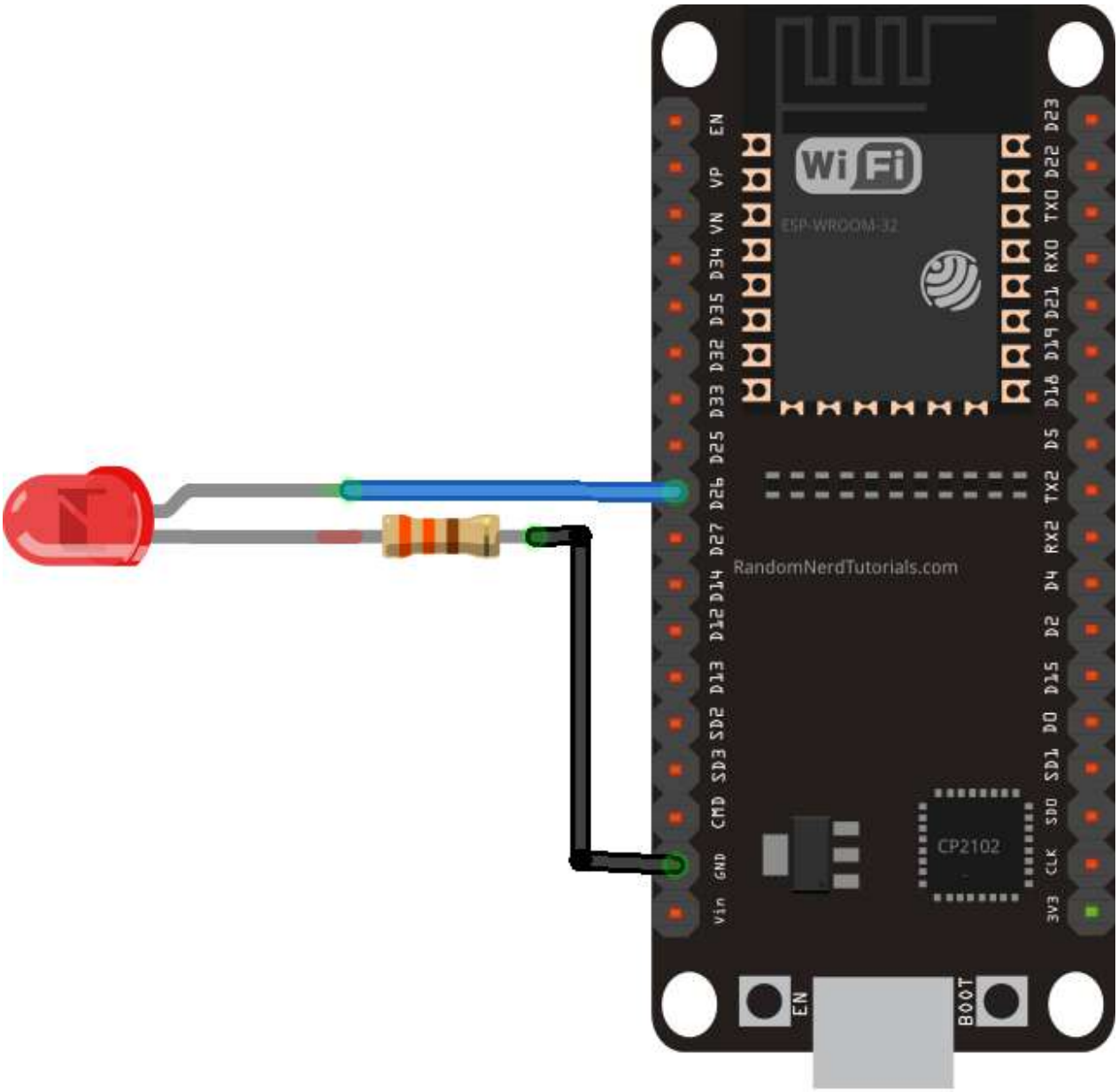
Basically, this code subtracts the previous recorded time (`previousMillis`) from the current time (`currentMillis`). If the remainder is greater than the interval (in this case, 1000 milliseconds), the program updates the `previousMillis` variable to the current time, and either turns the LED on or off.

```
if (currentMillis - previousMillis >= interval) {
  // save the last time you blinked the LED
  previousMillis = currentMillis;
  (...)
```

Because this snippet is non-blocking, any code that's located outside of that first `if` statement should work normally.

You should now be able to understand that you can add other tasks to your `loop()` function and your code will still be blinking the LED every one second.

You can upload this code to your ESP32 and assemble the following schematic diagram to test it and modify the number of milliseconds to see how it works.



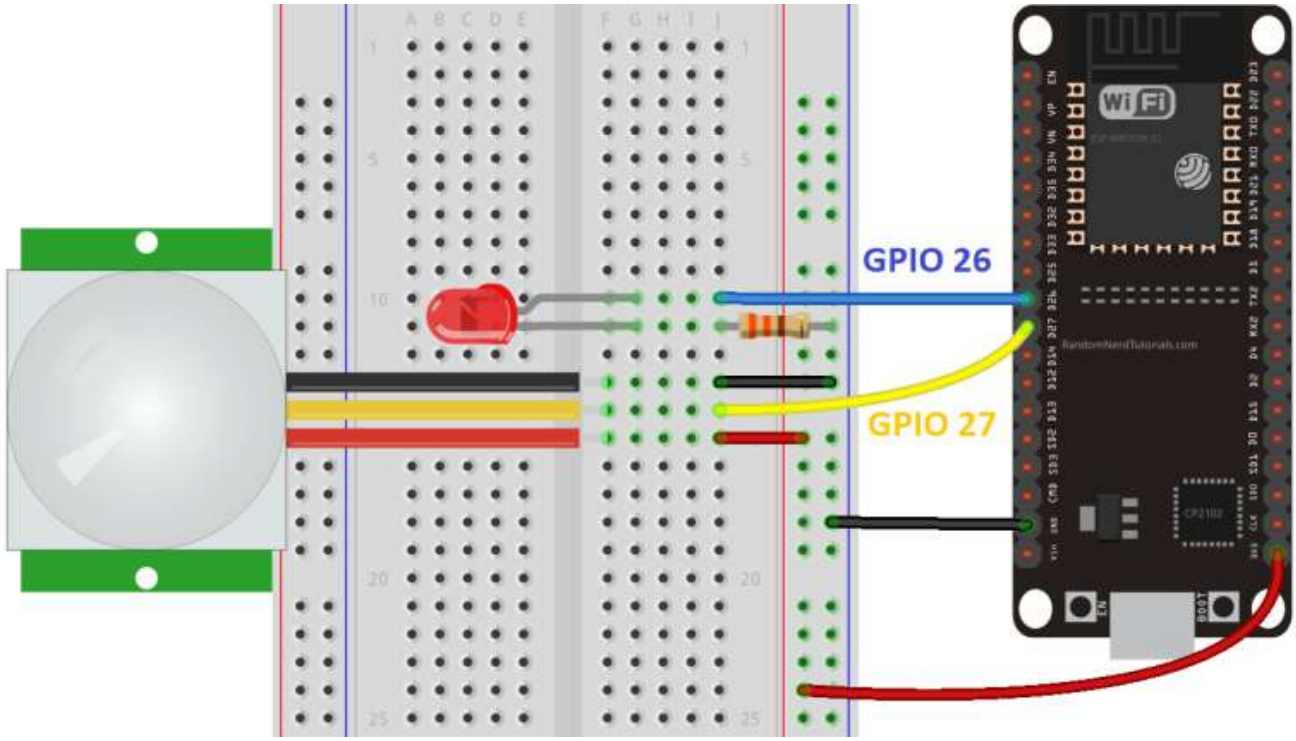
Note: If you've experienced any issues uploading code to your ESP32, take a look at the ESP32 Troubleshooting Guide.

ESP32 with PIR Motion Sensor

After understanding these concepts: interrupts and timers, let's continue with the project.

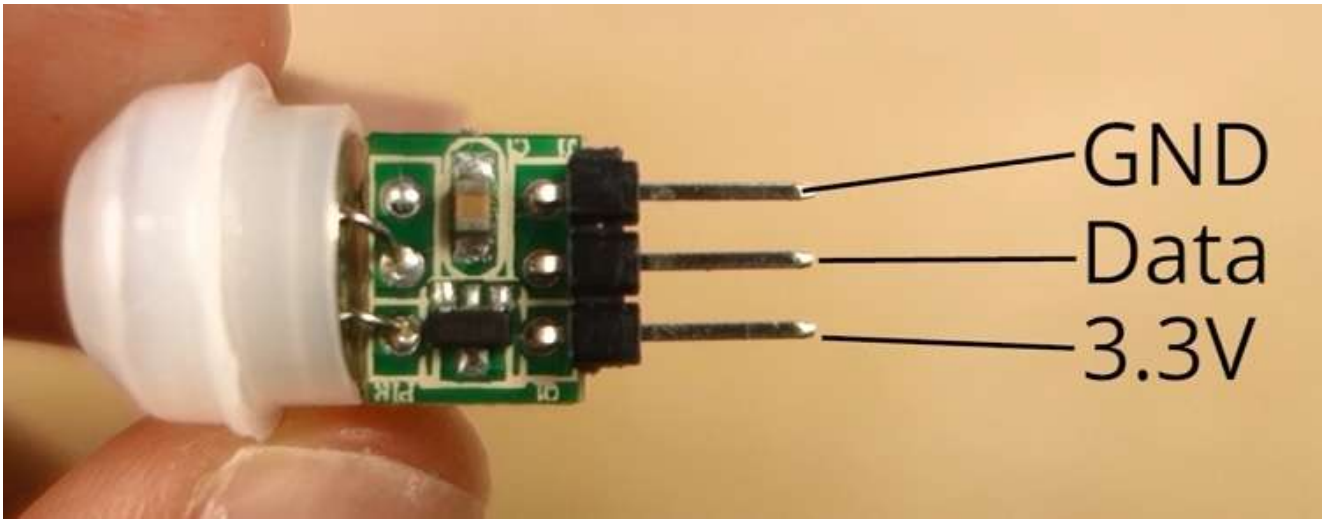
Schematic

The circuit we'll build is easy to assemble, we'll be using an LED with a resistor. The LED is connected to GPIO 26 . We'll be using the Mini AM312 PIR Motion Sensor that operates at 3.3V. It will be connected to GPIO 27 . Simply follow the next schematic diagram.



Important: the Mini AM312 PIR Motion Sensor used in this project operates at 3.3V. However, if you're using another PIR motion sensor like the HC-SR501, it operates at 5V. You can either modify it to operate at 3.3V or simply power it using the Vin pin.

The following figure shows the AM312 PIR motion sensor pinout.



Uploading the Code

After wiring the circuit as shown in the schematic diagram, copy the code provided to your Arduino IDE.

You can upload the code as it is, or you can modify the number of seconds the LED is lit after detecting motion. Simply change the `timeSeconds` variable with

the number of seconds you want.

```
/*********
  Rui Santos
  Complete project details at https://randomnerdtutorials.com
*********/

#define timeSeconds 10

// Set GPIOs for LED and PIR Motion Sensor
const int led = 26;
const int motionSensor = 27;

// Timer: Auxiliary variables
unsigned long now = millis();
unsigned long lastTrigger = 0;
boolean startTimer = false;
boolean motion = false;

// Checks if motion was detected, sets LED HIGH and starts a ti
void IRAM_ATTR detectsMovement() {
  digitalWrite(led, HIGH);
  startTimer = true;
  lastTrigger = millis();
}

void setup() {
  // Serial port for debugging purposes
  Serial.begin(115200);
```

View raw code

Note: if you've experienced any issues uploading code to your ESP32, take a look at the ESP32 Troubleshooting Guide.

How the Code Works

Let's take a look at the code. Start by assigning two GPIO pins to the `led` and `motionSensor` variables.

```
// Set GPIOs for LED and PIR Motion Sensor
const int led = 26;
const int motionSensor = 27;
```

Then, create variables that will allow you set a timer to turn the LED off after motion is detected.

```
// Timer: Auxiliar variables
long now = millis();
long lastTrigger = 0;
boolean startTimer = false;
```

The `now` variable holds the current time. The `lastTrigger` variable holds the time when the PIR sensor detects motion. The `startTimer` is a boolean variable that starts the timer when motion is detected.

setup()

In the `setup()`, start by initializing the Serial port at 115200 baud rate.

```
Serial.begin(115200);
```

Set the PIR Motion sensor as an INPUT PULLUP.

```
pinMode(motionSensor, INPUT_PULLUP);
```

To set the PIR sensor pin as an interrupt, use the `attachInterrupt()` function as described earlier.

```
attachInterrupt(digitalPinToInterrupt(motionSensor), detectsMovem
```

The pin that will detect motion is GPIO 27 and it will call the function `detectsMovement()` on RISING mode.

The LED is an OUTPUT whose state starts at LOW.

```
pinMode(led, OUTPUT);
digitalWrite(led, LOW);
```

loop()

The `loop()` function is constantly running over and over again. In every loop, the `now` variable is updated with the current time.

```
now = millis();
```

Nothing else is done in the `loop()`.

But, when motion is detected, the `detectsMovement()` function is called because we've set an interrupt previously on the `setup()`.

The `detectsMovement()` function prints a message in the Serial Monitor, turns the LED on, sets the `startTimer` boolean variable to `true` and updates the `lastTrigger` variable with the current time.

```
void IRAM_ATTR detectsMovement() {
  Serial.println("MOTION DETECTED!!!");
  digitalWrite(led, HIGH);
  startTimer = true;
  lastTrigger = millis();
}
```

Note: `IRAM_ATTR` is used to run the interrupt code in RAM, otherwise code is stored in flash and it's slower.

After this step, the code goes back to the `loop()`.

This time, the `startTimer` variable is true. So, when the time defined in seconds has passed (since motion was detected), the following `if` statement will be true.

```
if(startTimer && (now - lastTrigger > (timeSeconds*1000))) {
  Serial.println("Motion stopped...");
  digitalWrite(led, LOW);
  startTimer = false;
}
```

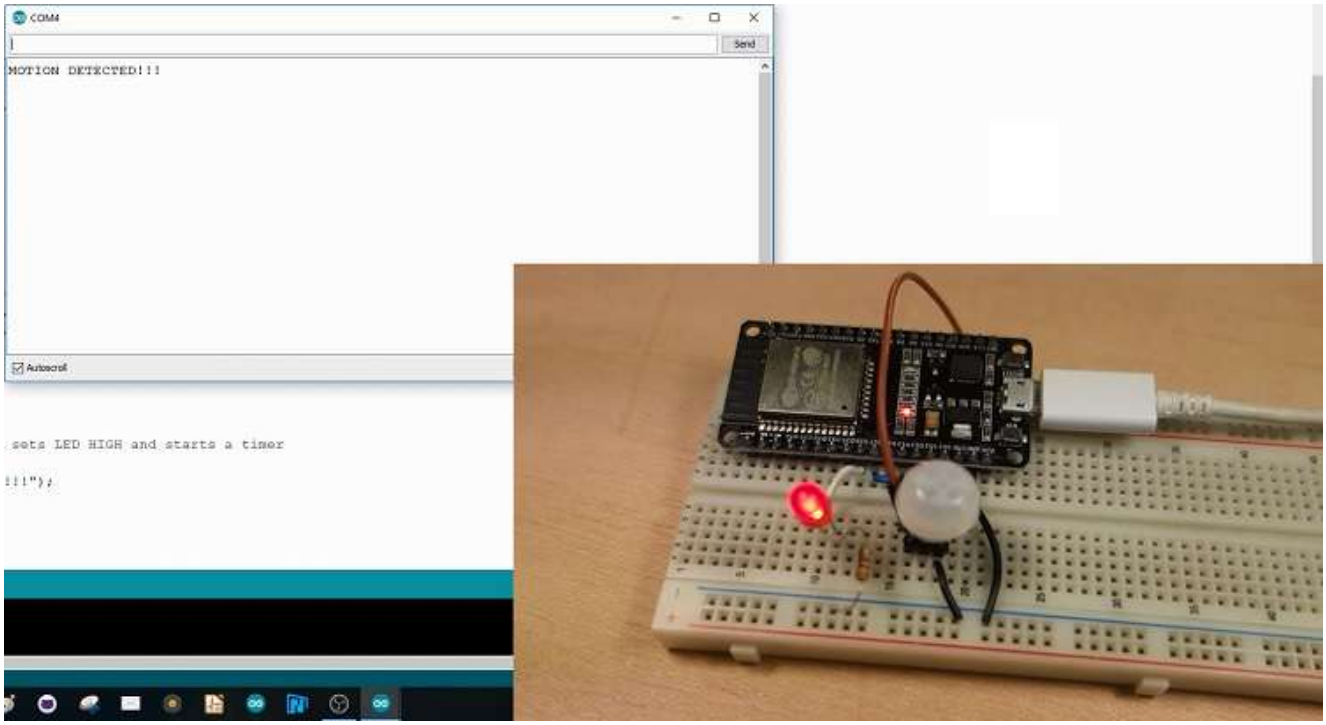
The “Motion stopped…” message will be printed in the Serial Monitor, the LED is turned off, and the startTimer variable is set to false.

Demonstration

Upload the code to your ESP32 board. Make sure you have the right board and COM port selected.

Open the Serial Monitor at a baud rate of 115200.

Move your hand in front of the PIR sensor. The LED should turn on, and a message is printed in the Serial Monitor saying “MOTION DETECTED!!!”. After 10 seconds the LED should turn off.



Wrapping Up

To wrap up, interrupts are used to detect a change in the GPIO state without the need to constantly read the current GPIO value. With interrupts, when a change is detected, a function is triggered. You've also learned how to set a simple timer that allows you to check if a predefined number of seconds have passed without having to block your code.

We have other tutorials related with ESP32 that you may also like:

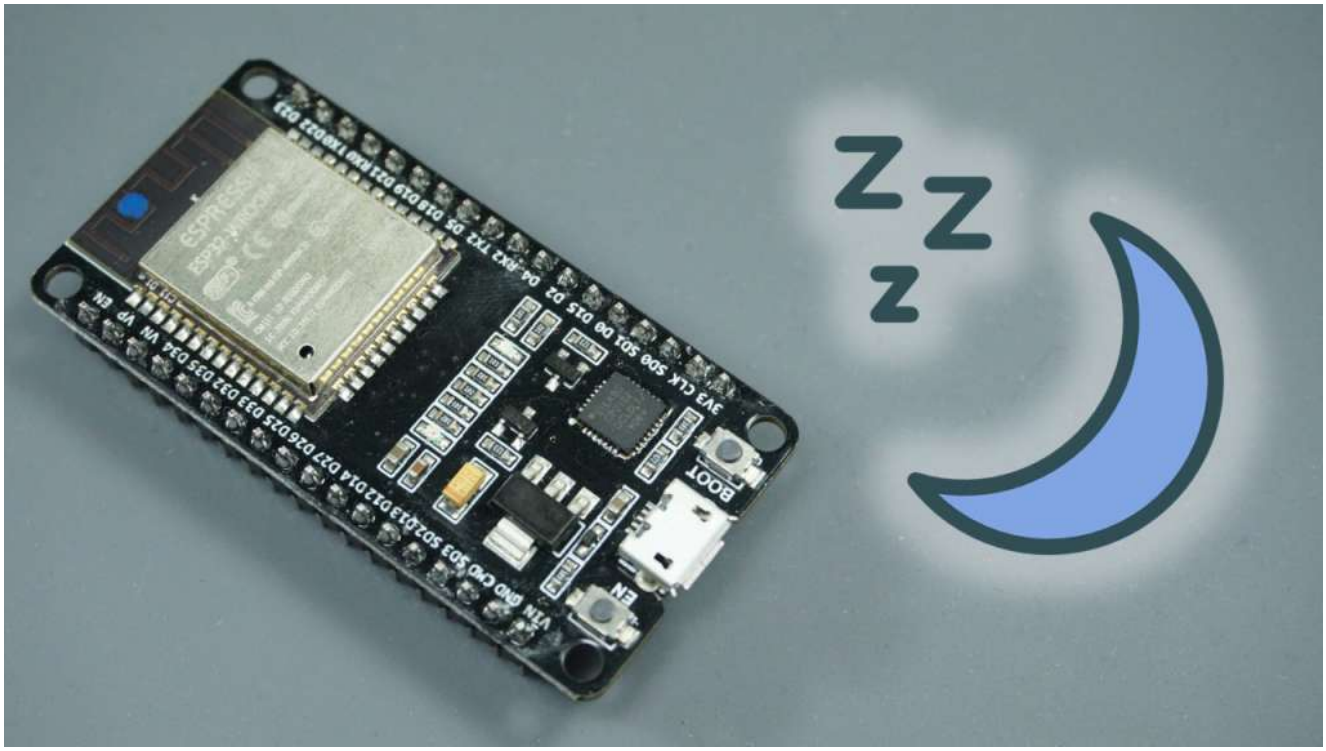
- ESP32 Web Server – Arduino IDE
- ESP32 Data Logging Temperature to MicroSD Card
- How to Use I2C LCD with ESP32 on Arduino IDE
- ESP32 vs ESP8266 – Pros and Cons

This is an excerpt from our course: Learn ESP32 with Arduino IDE. If you like ESP32 and you want to learn more, we recommend enrolling in Learn ESP32 with Arduino IDE course.

Thanks for reading.

ESP32 Deep Sleep with Arduino IDE and Wake Up Sources

This article is a complete guide for the ESP32 Deep Sleep mode with Arduino IDE. We'll show you how to put the ESP32 into deep sleep and take a look at different modes to wake it up: **timer wake up**, **touch wake up**, and **external wake up**. This guide provides practical examples with code, code explanation, and schematics.



Related Content: ESP8266 Deep Sleep with Arduino IDE

This article is divided into 4 different parts:

1. Introducing Deep Sleep Mode
2. Timer Wake Up
3. Touch Wake Up
4. External Wake Up

Introducing Deep Sleep Mode

The ESP32 can switch between different power modes:

- Active mode
- Modem Sleep mode
- Light Sleep mode
- Deep Sleep mode
- Hibernation mode

You can compare the five different modes on the following table from the ESP32 Espressif datasheet.

Power mode	Active	Modem-sleep	Light-sleep	Deep-sleep	Hibernation
Sleep pattern	Association sleep pattern			ULP sensor-monitored pattern	-
CPU	ON	ON	PAUSE	OFF	OFF
Wi-Fi/BT baseband and radio	ON	OFF	OFF	OFF	OFF
RTC memory and RTC peripherals	ON	ON	ON	ON	OFF
ULP co-processor	ON	ON	ON	ON/OFF	OFF

The ESP32 Espressif datasheet also provides a table comparing the power consumption of the different power modes.

Power mode	Description	Power consumption
Active (RF working)	Wi-Fi Tx packet 14 dBm ~ 19.5 dBm	Please refer to Table 10 for details.
	Wi-Fi / BT Tx packet 0 dBm	
	Wi-Fi / BT Rx and listening	
Modem-sleep	The CPU is powered on.	Max speed 240 MHz: 30 mA ~ 50 mA
		Normal speed 80 MHz: 20 mA ~ 25 mA
		Slow speed 2 MHz: 2 mA ~ 4 mA
Light-sleep	-	0.8 mA
Deep-sleep	The ULP co-processor is powered on.	150 μA
	ULP sensor-monitored pattern	100 μA @1% duty
	RTC timer + RTC memory	10 μA
Hibernation	RTC timer only	5 μA
Power off	CHIP_PU is set to low level, the chip is powered off	0.1 μA

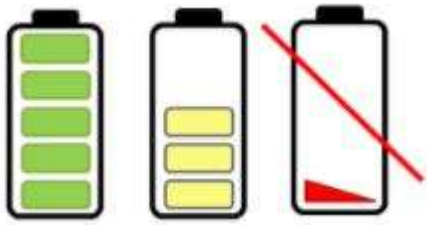
And here's also **Table 10** to compare the power consumption in active mode:

Table 10: RF Power-Consumption Specifications

Mode	Min	Typ	Max	Unit
Transmit 802.11b, DSSS 1 Mbps, POUT = +19.5 dBm	-	240	-	mA
Transmit 802.11b, OFDM 54 Mbps, POUT = +16 dBm	-	190	-	mA
Transmit 802.11g, OFDM MCS7, POUT = +14 dBm	-	180	-	mA
Receive 802.11b/g/n	-	95 ~ 100	-	mA
Transmit BT/BLE, POUT = 0 dBm	-	130	-	mA
Receive BT/BLE	-	95 ~ 100	-	mA

Why Deep Sleep Mode?

Having your ESP32 running on active mode with batteries it's not ideal, since the power from batteries will drain very quickly.



If you put your ESP32 in deep sleep mode, it will reduce the power consumption and your batteries will last longer.

Having your ESP32 in deep sleep mode means cutting with the activities that consume more power while operating, but leave just enough activity to wake up the processor when something interesting happens.

In deep sleep mode neither CPU or Wi-Fi activities take place, but the Ultra Low Power (ULP) co-processor can still be powered on.

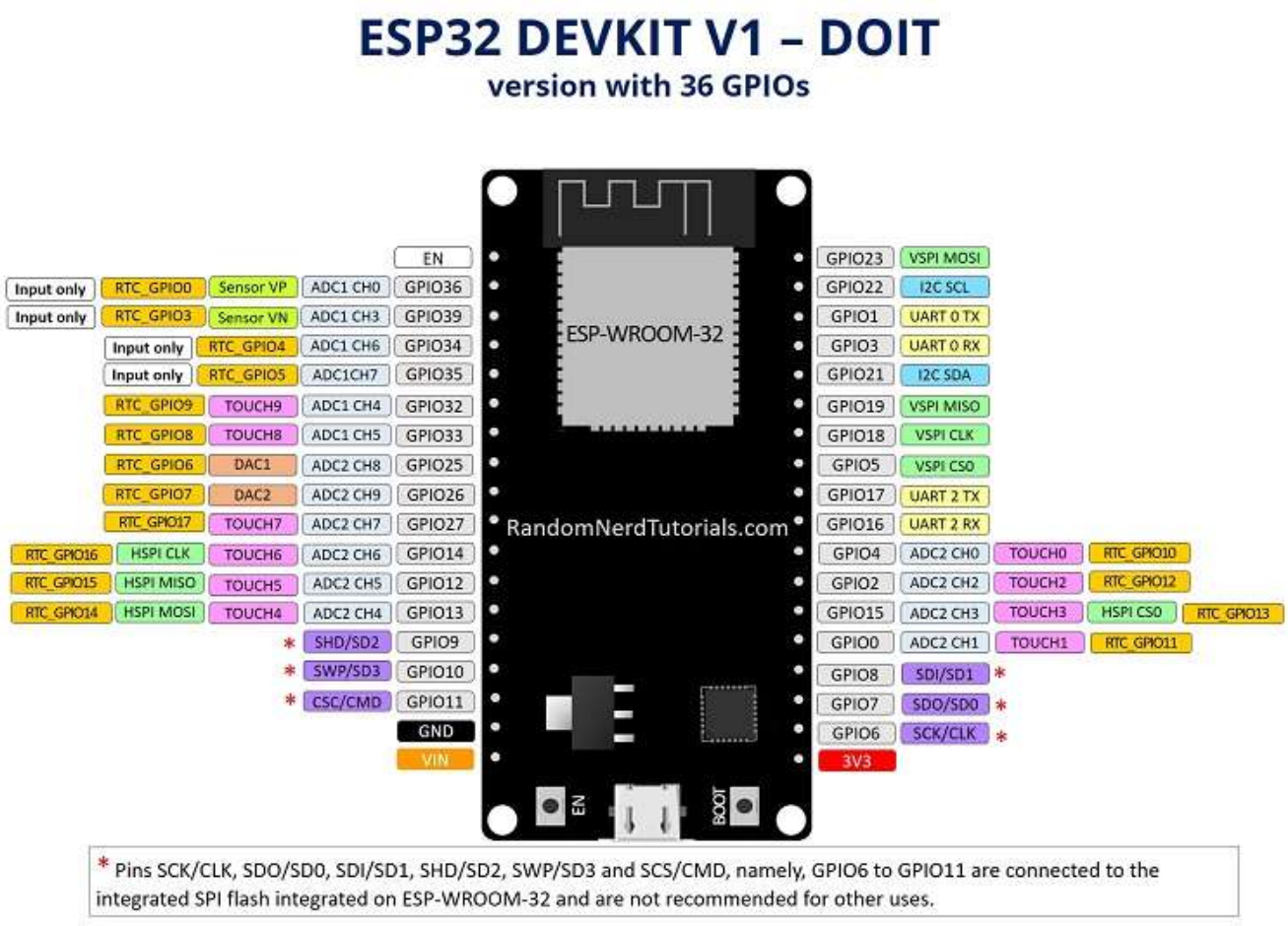
While the ESP32 is in deep sleep mode the RTC memory also remains powered on, so we can write a program for the ULP co-processor and store it in the RTC memory to access peripheral devices, internal timers, and internal sensors.

This mode of operation is useful if you need to wake up the main CPU by an external event, timer, or both, while maintaining minimal power consumption.

RTC_GPIO Pins

During deep sleep, some of the ESP32 pins can be used by the ULP co-processor, namely the RTC_GPIO pins, and the Touch Pins. The ESP32 datasheet provides a table identifying the RTC_GPIO pins. You can find that table here at page 7.

You can use that table as a reference, or take a look at the following pinout to locate the different RTC_GPIO pins. The RTC_GPIO pins are highlighted with an orange rectangular box.



You might also like reading: ESP32 Pinout Reference: Which GPIO pins should you use?

Wake Up Sources

After putting the ESP32 into deep sleep mode, there are several ways to wake it up:

1. You can use the **timer**, waking up your ESP32 using predefined periods of time;
2. You can use the **touch pins**;

3. You can use **two possibilities of external wake up**: you can use either one external wake up, or several different external wake ups;
4. You can use the **ULP co-processor** to wake up – this won't be covered in this guide.

Writing a Deep Sleep Sketch

To write a sketch to put your ESP32 into deep sleep mode, and then wake it up, you need to keep in mind that:

1. First, you need to configure the wake up sources. This means configure what will wake up the ESP32. You can use one or combine more than one wake up source.
2. You can decide what peripherals to shut down or keep on during deep sleep. However, by default, the ESP32 automatically powers down the peripherals that are not needed with the wake up source you define.
3. Finally, you use the `esp_deep_sleep_start()` function to put your ESP32 into deep sleep mode.

Timer Wake Up

The ESP32 can go into deep sleep mode, and then wake up at predefined periods of time. This feature is specially useful if you are running projects that require time stamping or daily tasks, while maintaining low power consumption.

The ESP32 RTC controller has a built-in timer you can use to wake up the ESP32 after a predefined amount of time.

Enable Timer Wake Up

Enabling the ESP32 to wake up after a predefined amount of time is very straightforward. In the Arduino IDE, you just have to specify the sleep time in microseconds in the following function:

```
esp_sleep_enable_timer_wakeup(time_in_us)
```

Code

Let's see how this works using an example from the library. Open your Arduino IDE, and go to **File** > **Examples** > **ESP32** > **Deep Sleep**, and open the **TimerWakeUp** sketch.

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

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

This code is under Public Domain License.

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

#define uS_TO_S_FACTOR 1000000  /* Conversion factor for micro
#define TIME_TO_SLEEP  5        /* Time ESP32 will go to sleep

RTC_DATA_ATTR int bootCount = 0;

/*
```

View raw code

Let's take a look at this code. The first comment describes what is powered off during deep sleep with timer wake up.

```
In this mode CPUs, most of the RAM,
and all the digital peripherals which are clocked
from APB_CLK are powered off. The only parts of
the chip which can still be powered on are:
RTC controller, RTC peripherals ,and RTC memories
```

When you use timer wake up, the parts that will be powered on are RTC controller, RTC peripherals, and RTC memories.

Define the Sleep Time

These first two lines of code define the period of time the ESP32 will be sleeping.

```
#define uS_TO_S_FACTOR 1000000 /* Conversion factor for micro sec
#define TIME_TO_SLEEP 5 /* Time ESP32 will go to sleep (in second
```

This example uses a conversion factor from microseconds to seconds, so that you can set the sleep time in the `TIME_TO_SLEEP` variable in seconds. In this case, the example will put the ESP32 into deep sleep mode for 5 seconds.

Save Data on RTC Memories

With the ESP32, you can save data on the RTC memories. The ESP32 has 8kB SRAM on the RTC part, called RTC fast memory. The data saved here is not erased during deep sleep. However, it is erased when you press the reset button (the button labeled EN on the ESP32 board).

To save data on the RTC memory, you just have to add `RTC_DATA_ATTR` before a variable definition. The example saves the `bootCount` variable on the RTC memory. This variable will count how many times the ESP32 has woken up from deep sleep.

```
RTC_DATA_ATTR int bootCount = 0;
```

Wake Up Reason

Then, the code defines the `print_wakeup_reason()` function, that prints the reason by which the ESP32 has been awaken from sleep.

```
void print_wakeup_reason(){
  esp_sleep_wakeup_cause_t wakeup_reason;

  wakeup_reason = esp_sleep_get_wakeup_cause();

  switch(wakeup_reason){
    case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wakeup caused by
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("Wakeup caused by
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wakeup caused b
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wakeup cause
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("Wakeup caused by
    default : Serial.printf("Wakeup was not caused by deep sleep:
  }
}
```

The setup()

In the `setup()` is where you should put your code. In deep sleep, the sketch never reaches the `loop()` statement. So, you need to write all the sketch in the `setup()`.

This example starts by initializing the serial communication at a baud rate of 115200.

```
Serial.begin(115200);
```

Then, the `bootCount` variable is increased by one in every reboot, and that number is printed in the serial monitor.

```
++bootCount;
```

```
  Serial.println("Boot number: " + String(bootCount));
```

Then, the code calls the `print_wakeup_reason()` function, but you can call any function you want to perform a desired task. For example, you may want to wake up your ESP32 once a day to read a value from a sensor.

Next, the code defines the wake up source by using the following function:

```
esp_sleep_enable_timer_wakeup(time_in_us)
```

This function accepts as argument the time to sleep in microseconds as we've seen previously.

In our case, we have the following:

```
esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
```

Then, after all the tasks are performed, the esp goes to sleep by calling the following function:

```
esp_deep_sleep_start()
```

The loop()

The `loop()` section is empty, because the ESP32 will go to sleep before reaching this part of the code. So, you need to write all your sketch in the `setup()`.

Upload the example sketch to your ESP32. Make sure you have the right board and COM port selected.

Testing the Timer Wake Up

Open the Serial Monitor at a baud rate of 115200.

Every 5 seconds, the ESP wakes up, prints a message on the serial monitor, and goes to deep sleep again.

Every time the ESP wakes up the `bootCount` variable increases. It also prints the wake up reason as shown in the figure below.

```
rst:0x5 (DEEPSLEEP_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:812
load:0x40078000,len:0
load:0x40078000,len:11392
entry 0x40078a9c
Boot number: 2
Wakeup caused by timer
Setup ESP32 to sleep for every 5 Seconds
Going to sleep now
ets Jun  8 2016 00:22:57

rst:0x5 (DEEPSLEEP_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:812
load:0x40078000,len:0
load:0x40078000,len:11392
entry 0x40078a9c
Boot number: 3
Wakeup caused by timer
Setup ESP32 to sleep for every 5 Seconds
Going to sleep now
```

However, notice that if you press the EN button on the ESP32 board, it resets the boot count to 1 again.

You can modify the provided example, and instead of printing a message you can make your ESP do any other task. The timer wake up is useful to perform periodic tasks with the ESP32, like daily tasks, without draining much power.

Touch Wake Up

You can wake up the ESP32 from deep sleep using the touch pins. This section shows how to do that using the Arduino IDE.

Enable Touch Wake Up

Enabling the ESP32 to wake up using a touchpin is simple. In the Arduino IDE, you need to use the following function:

```
esp_sleep_enable_touchpad_wakeup()
```

Code

Let's see how this works using an example from the library. Open your Arduino IDE, and go to **File** > **Examples** > **ESP32** > **Deep Sleep**, and open the **TouchWakeUp** sketch.

```
/*
  Deep Sleep with Touch Wake Up
  This code displays how to use deep sleep with a touch as a wa
  ESP32 can have multiple touch pads enabled as wakeup source
  ESP32-S2 and ESP32-S3 supports only 1 touch pad as wakeup sou
  This code is under Public Domain License. Author: Pranav Cher
*/

#if CONFIG_IDF_TARGET_ESP32
  #define THRESHOLD   40      /* Greater the value, more the se
#else //ESP32-S2 and ESP32-S3 + default for other chips (to be
  #define THRESHOLD   5000   /* Lower the value, more the sensi
#endif

RTC_DATA_ATTR int bootCount = 0;
```

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

  wakeup_reason = esp_sleep_get_wakeup_cause();

  switch(wakeup_reason)
```

View raw code

Setting the Threshold

The first thing you need to do is setting a threshold value for the touch pins. In this case we're setting the `Threshold` to 40. You may need to change the threshold value depending on your project.

```
#define Threshold 40
```

When you touch a touch-sensitive GPIO, the value read by the sensor decreases. So, you can set a threshold value that makes something happen when touch is detected.

The threshold value set here means that when the value read by the touch-sensitive GPIO is below 40, the ESP32 should wake up. You can adjust that value accordingly to the desired sensitivity.

Attach Interrupts

You need to attach interrupts to the touch sensitive pins. When touch is detected on a specified GPIO, a callback function is executed. For example, take a look at the following line:

```
//Setup interrupt on Touch Pad 3 (GPIO15)
touchAttachInterrupt(T3, callback, Threshold);
```

When the value read on T3 (GPIO 15) is lower than the value set on the `Threshold` variable, the ESP32 wakes up and the `callback` function is executed.

The `callback()` function will only be executed if the ESP32 is awake.

- If the ESP32 is asleep and you touch T3, the ESP will wake up – the `callback()` function won't be executed if you just press and release the touch pin;
- If the ESP32 is awake and you touch T3, the callback function will be executed. So, if you want to execute the `callback()` function when you wake up the ESP32, you need to hold the touch on that pin for a while, until the function is executed.

In this case the `callback()` function is empty.

```
void callback(){
   //placeholder callback function
}
```

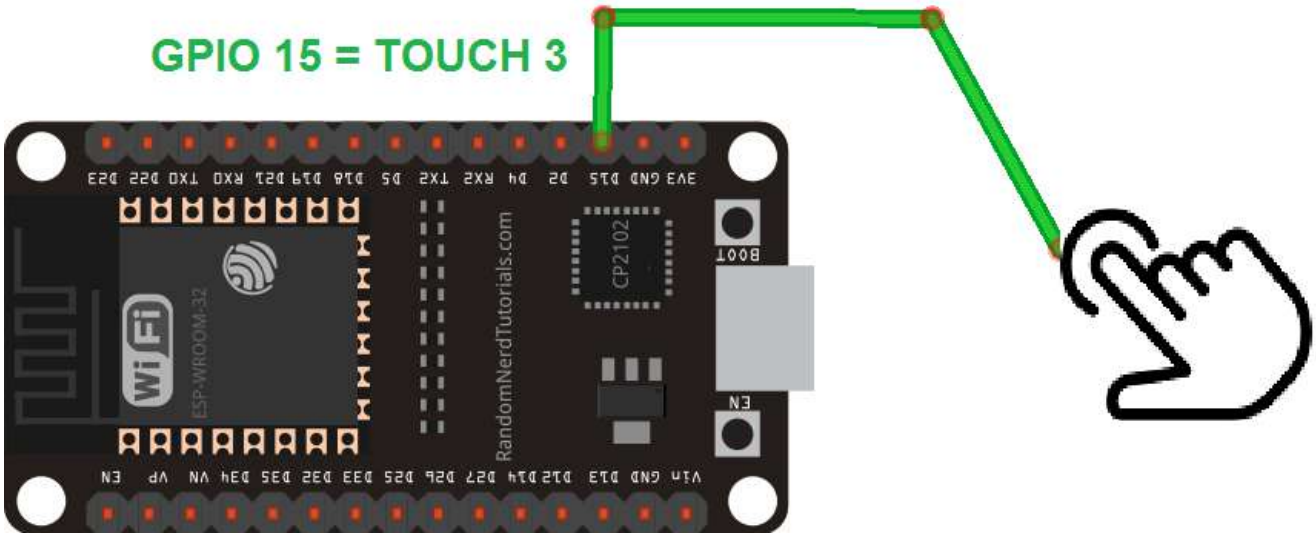
If you want to wake up the ESP32 using different touch pins, you just have to attach interrupts to those pins.

Next, you need to use the `esp_sleep_enable_touchpad_wakeup()` function to set the touch pins as a wake up source.

```
//Configure Touchpad as wakeup source
esp_sleep_enable_touchpad_wakeup()
```

Schematic

To test this example, wire a cable to GPIO 15 , as shown in the schematic below.



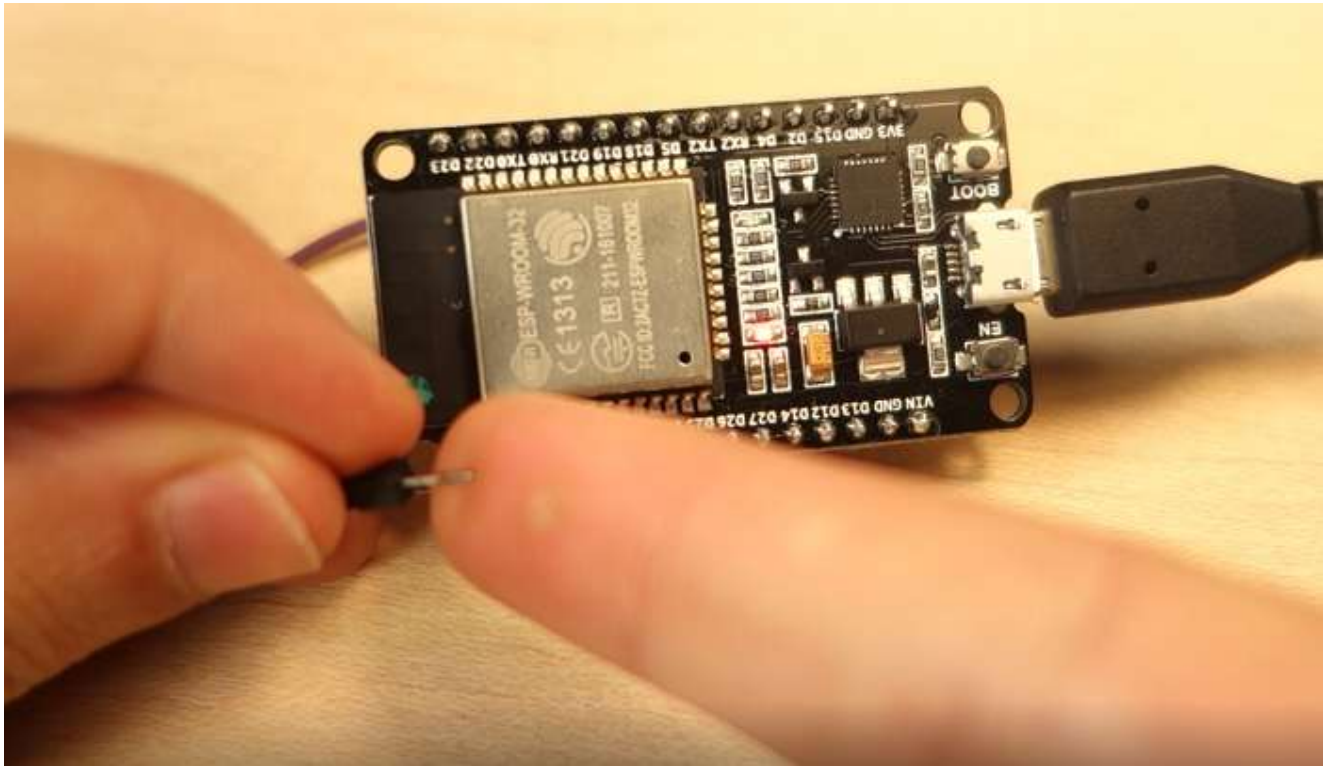
(This schematic uses the ESP32 DEVKIT V1 module version with 30 GPIOs – if you're using another model, please check the pinout for the board you're using.)

Testing the Example

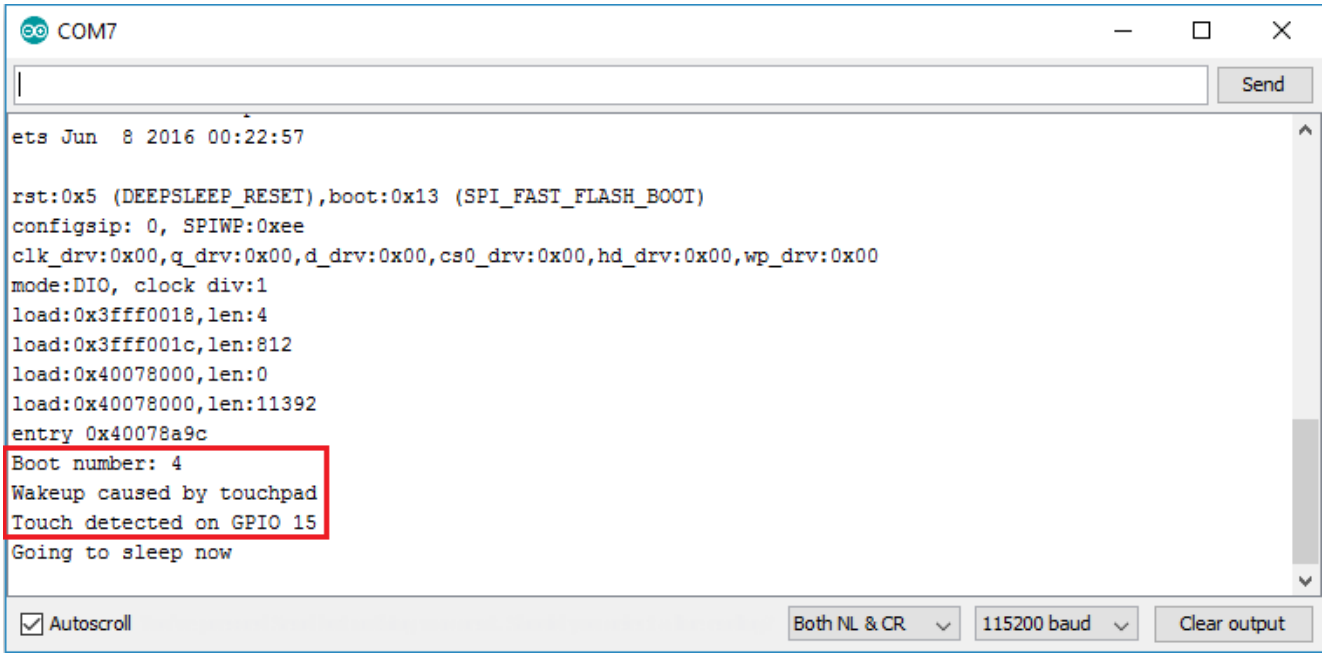
Upload the code to your ESP32, and open the Serial Monitor at a baud rate of 115200.

The ESP32 goes into deep sleep mode.

You can wake it up by touching the wire connected to Touch Pin 3.



When you touch the pin, the ESP32 displays on the Serial Monitor: the boot number, the wake up cause, and in which GPIO touch was detected.



External Wake Up

Besides the timer and the touch pins, we can also awake the ESP32 from deep sleep by toggling the value of a signal on a pin, like the press of a button. This is called an external wake up. You have two possibilities of external wake up: ext0, and ext1.

External Wake Up (ext0)

This wake up source allows you to use a pin to wake up the ESP32.

The ext0 wake up source option uses RTC GPIOs to wake up. So, RTC peripherals will be kept on during deep sleep if this wake up source is requested.

To use this wake up source, you use the following function:

```
esp_sleep_enable_ext0_wakeup(GPIO_NUM_X, level)
```

This function accepts as first argument the pin you want to use, in this format `GPIO_NUM_X` , in which **X** represents the GPIO number of that pin.

The second argument, `level` , can be either 1 or 0. This represents the state of the GPIO that will trigger wake up.

Note: with this wake up source, you can only use pins that are RTC GPIOs.

External Wake Up (ext1)

This wake up source allows you to use multiple RTC GPIOs. You can use two different logic functions:

- Wake up the ESP32 if any of the pins you've selected are high;
- Wake up the ESP32 if all the pins you've selected are low.

This wake up source is implemented by the RTC controller. So, RTC peripherals and RTC memories can be powered off in this mode.

To use this wake up source, you use the following function:

```
esp_sleep_enable_ext1_wakeup(bitmask, mode)
```

This function accepts two arguments:

- A bitmask of the GPIO numbers that will cause the wake up;
- Mode: the logic to wake up the ESP32. It can be:
  - `ESP_EXT1_WAKEUP_ALL_LOW` : wake up when all GPIOs go low;
  - `ESP_EXT1_WAKEUP_ANY_HIGH` : wake up if any of the GPIOs go high.

Note: with this wake up source, you can only use pins that are RTC GPIOs.

Code

Let's explore the example that comes with the ESP32 library. Go to **File** > **Examples** > **ESP32** > **Deep Sleep** > **ExternalWakeUp**:

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

This code is under Public Domain License.

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

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

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

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

  RTC_DATA_ATTR int bootCount = 0;
```

View raw code

This example awakes the ESP32 when you trigger GPIO 33 to high. The code example shows how to use both methods: ext0 and ext1. If you upload the code as it is, you'll use ext0. The function to use ext1 is commented. We'll show you how both methods work and how to use them.

This code is very similar with the previous ones in this article. In the `setup()`, you start by initializing the serial communication:

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

Then, you increment one to the `bootCount` variable, and print that variable in the Serial Monitor.

```
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));
```

Next, you print the wake up reason using the `print_wakeup_reason()` function defined earlier.

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

After this, you need to enable the wake up sources. We'll test each of the wake up sources, ext0 and ext1, separately.

ext0

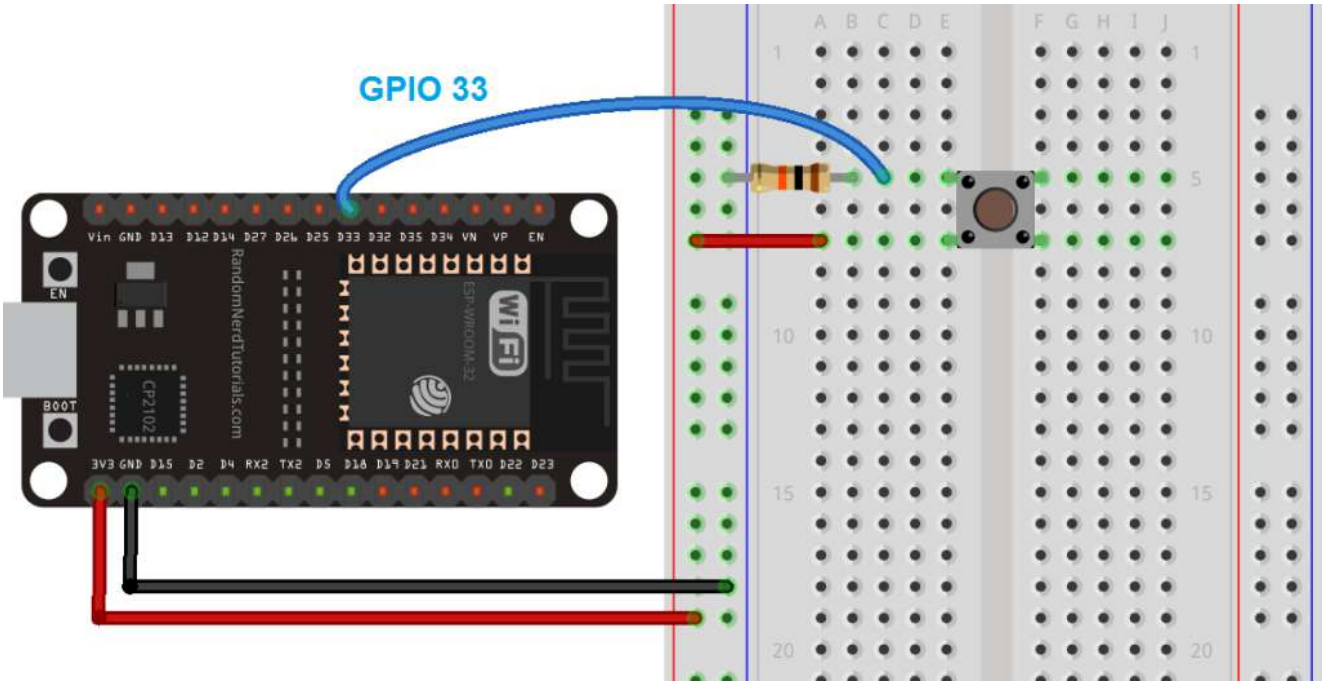
In this example, the ESP32 wakes up when the GPIO 33 is triggered to high:

```
esp_sleep_enable_ext0_wakeup(GPIO_NUM_33,1); //1 = High, 0 = Low
```

Instead of GPIO 33 , you can use any other RTC GPIO pin.

Schematic

To test this example, wire a pushbutton to your ESP32 by following the next schematic diagram. The button is connected to GPIO 33 using a pull down 10K Ohm resistor.



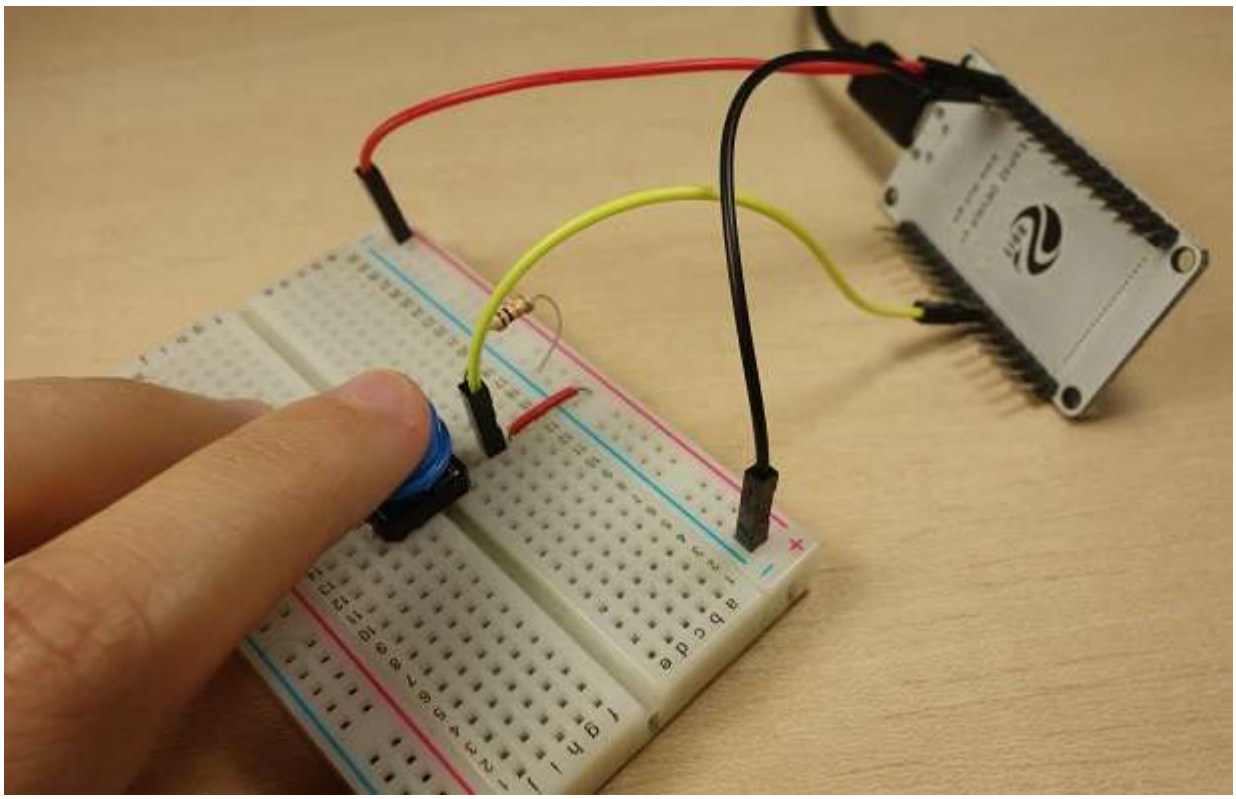
(This schematic uses the ESP32 DEVKIT V1 module version with 30 GPIOs – if you're using another model, please check the pinout for the board you're using.)

> **Note:** only RTC GPIOs can be used as a wake up source. Instead of GPIO 33, you could also use any RTC GPIO pins to connect your button.

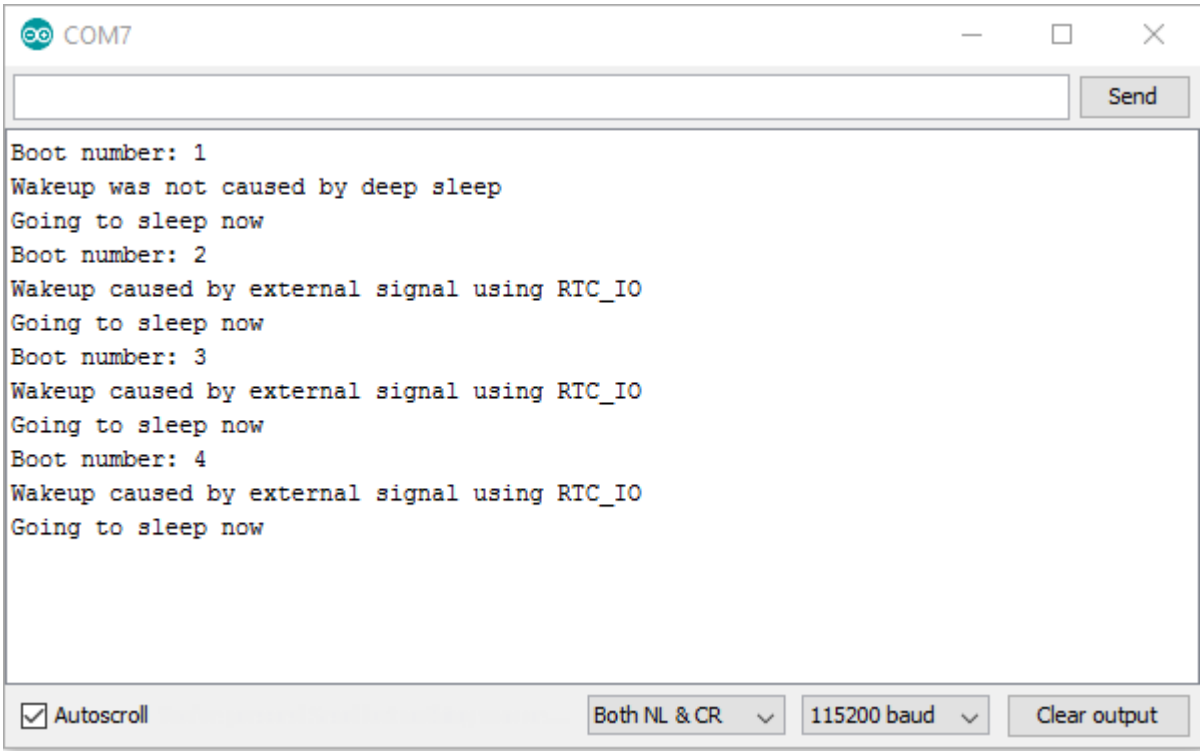
Testing the Example

Let’s test this example. Upload the example code to your ESP32. Make sure you have the right board and COM port selected. Open the Serial Monitor at a baud rate of 115200.

Press the pushbutton to wake up the ESP32.



Try this several times, and see the boot count increasing in each button press.



Using this method is useful to wake up your ESP32 using a pushbutton, for example, to make a certain task. However, with this method you can only use one GPIO as wake up source.

What if you want to have different buttons, all of them wake up the ESP, but do different tasks? For that you need to use the ext1 method.

ext1

The ext1 allows you to wake up the ESP using different buttons and perform different tasks depending on the button you pressed.

Instead of using the `esp_sleep_enable_ext0_wakeup()` function, you use the `esp_sleep_enable_ext1_wakeup()` function. In the code, that function is commented:

```
//If you were to use ext1, you would use it like
//esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP
```

Uncomment that function so that you have:

```
esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_A
```

The first argument of the function is a bitmask of the GPIOs you'll use as a wake up source, and the second argument defines the logic to wake up the ESP32.

In this example we're using the variable `BUTTON_PIN_BITMASK`, that was defined at the beginning of the code:

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

This is only defining one pin as a wake up source, GPIO 33. You need to modify the bitmask to configure more GPIOs as a wake up source.

GPIOs Bitmask

To get the GPIOs bitmask, follow the next steps:

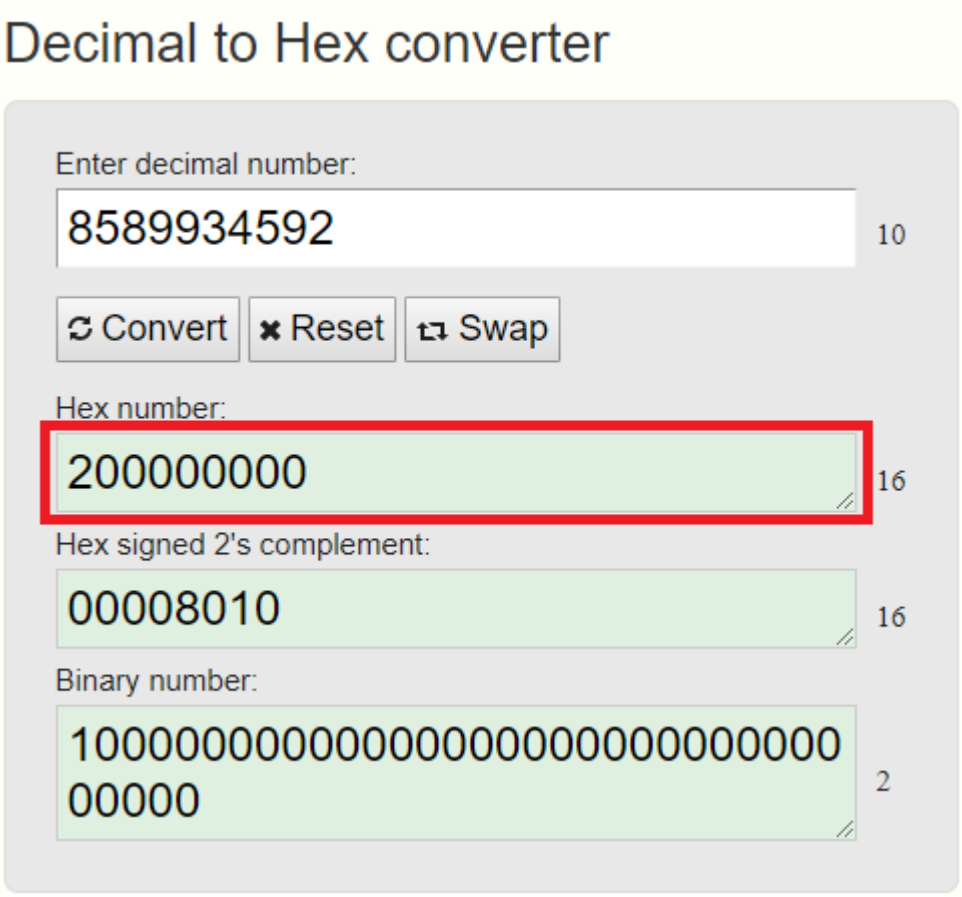
1. Calculate 2^(GPIO_NUMBER). Save the result in decimal;
2. Go to rapidtables.com/convert/number/decimal-to-hex.html and convert the decimal number to hex;
3. Replace the hex number you've obtained in the `BUTTON_PIN_BITMASK` variable.

Mask for a single GPIO

For you to understand how to get the GPIOs bitmask, let's go through an example. In the code from the library, the button is connected to GPIO 33. To get the mask for GPIO 33:

1. Calculate 2^33. You should get **8589934592**;

2. Convert that number (8589934592) to hexadecimal. You can go to this converter to do that:



3. Copy the Hex number to the `BUTTON_PIN_BITMASK` variable, and you should get:

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

Mask for several GPIOs

If you want to use GPIO 2 and GPIO 15 as a wake up source, you should do the following:

1. Calculate 2^2 + 2^15. You should get 32772
2. Convert that number to hex. You should get: 8004
3. Replace that number in the `BUTTON_PIN_BITMASK` as follows:

```
#define BUTTON_PIN_BITMASK 0x8004
```

Identifying the GPIO used as a wake up source

When you use several pins to wake up the ESP32, it is useful to know which pin caused the wake up. For that, you can use the following function:

```
esp_sleep_get_ext1_wakeup_status()
```

This function returns a number of base 2, with the GPIO number as an exponent: 2^(GPIO). So, to get the GPIO in decimal, you need to do the following calculation:

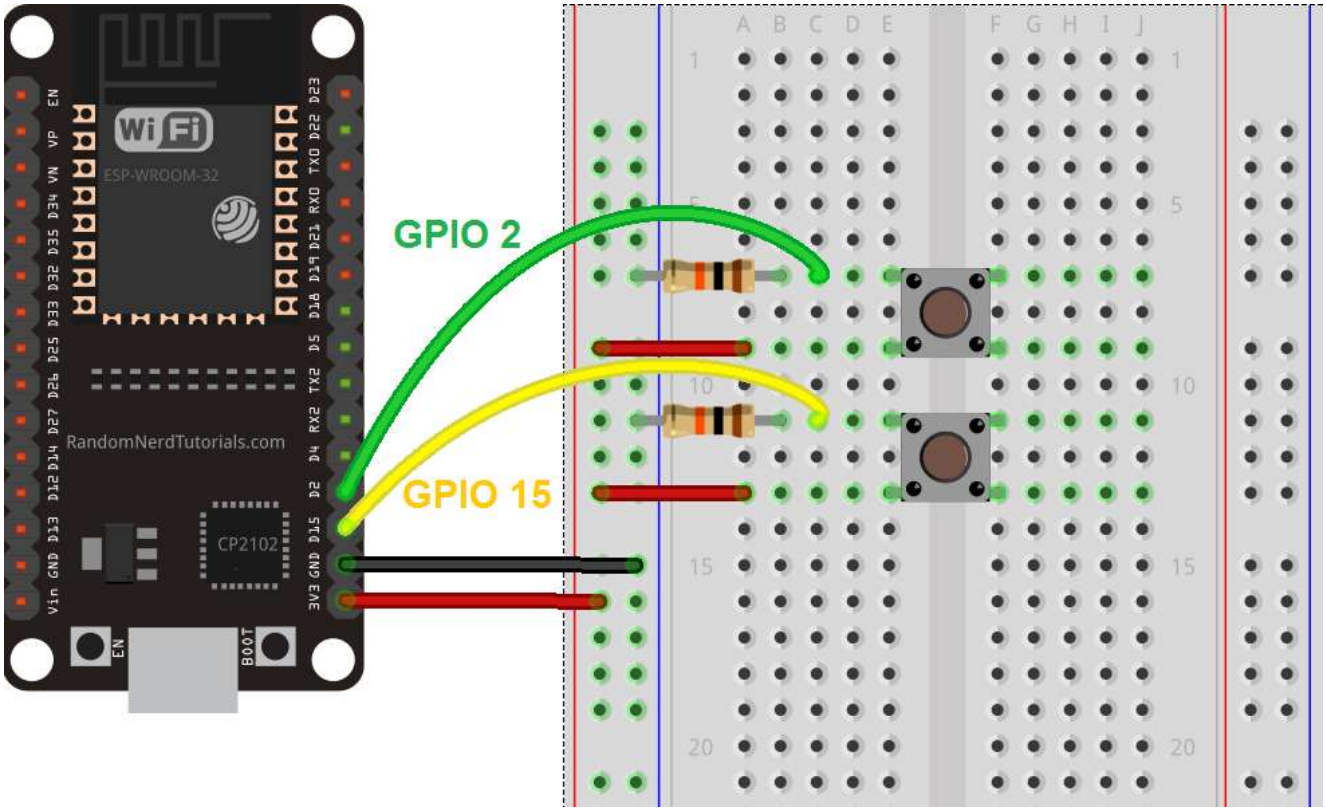
```
GPIO = log(RETURNED_VALUE)/log(2)
```

External Wake Up – Multiple GPIOs

Now, you should be able to wake up the ESP32 using different buttons, and identify which button caused the wake up. In this example we'll use GPIO 2 and GPIO 15 as a wake up source.

Schematic

Wire two buttons to your ESP32. In this example we're using GPIO 2 and GPIO 15 , but you can connect your buttons to any RTC GPIOs.



Code

You need to make some modifications to the example code we've used before:

- create a bitmask to use GPIO 15 and GPIO 2 . We've shown you how to do this before;
- enable ext1 as a wake up source;
- use the `esp_sleep_get_ext1_wakeup_status()` function to get the GPIO that triggered wake up.

The next sketch has all those changes implemented.

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

This code is under Public Domain License.

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

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

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

#define BUTTON_PIN_BITMASK 0x8004 // GPIOs 2 and 15

RTC_DATA_ATTR int bootCount = 0;
```

View raw code

You define the GPIOs mask at the beginning of the code:

```
#define BUTTON_PIN_BITMASK 0x8004 // GPIOs 2 and 15
```

You create a function to print the GPIO that caused the wake up:

```
void print_GPIO_wake_up(){
  int GPIO_reason = esp_sleep_get_ext1_wakeup_status();
  Serial.print("GPIO that triggered the wake up: GPIO ");
  Serial.println((log(GPIO_reason))/log(2), 0);
}
```

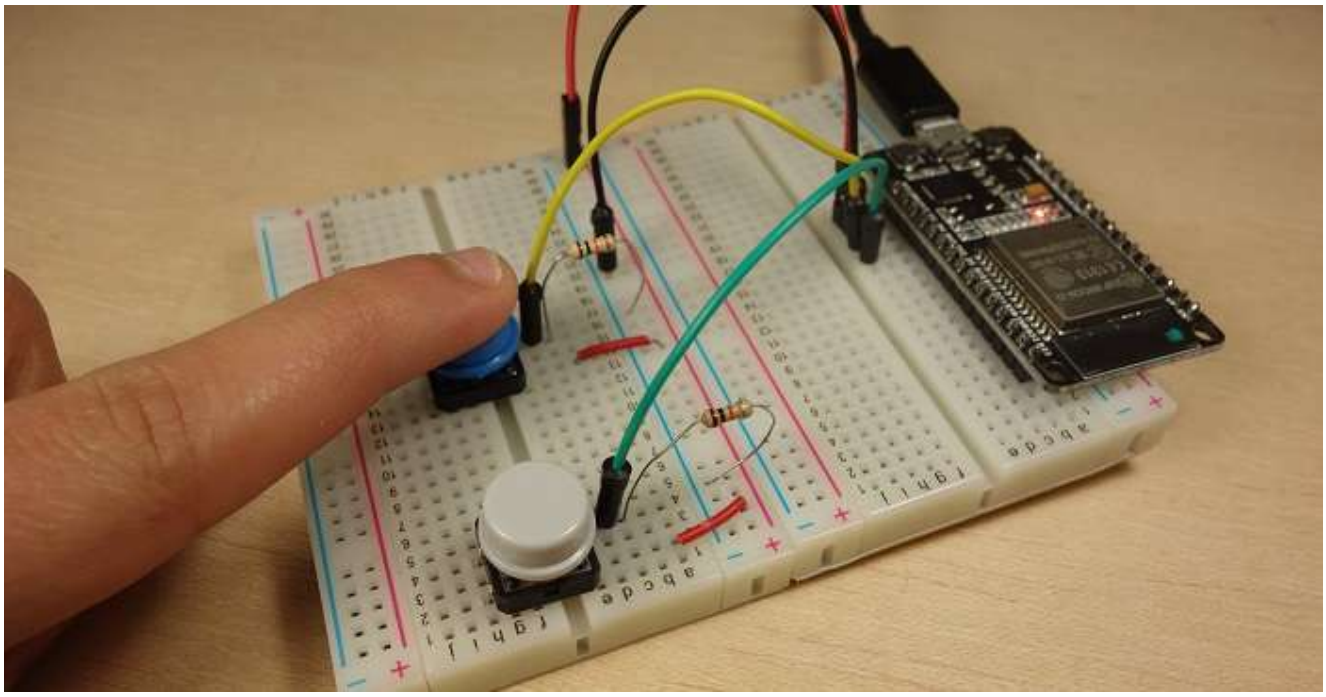
And finally, you enable ext1 as a wake up source:

```
esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_A
```

Testing the Sketch

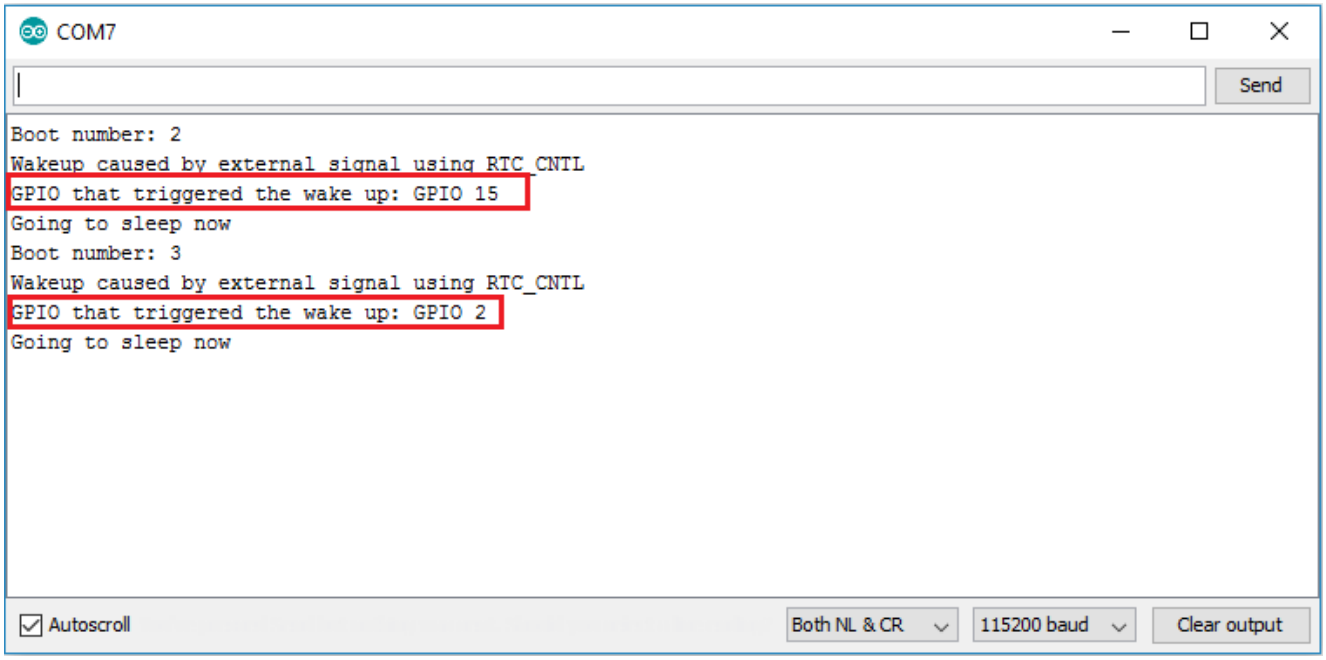
Having two buttons connected to GPIO 2 and GPIO 15, you can upload the code provided to your ESP32. Make sure you have the right board and COM port selected.

The ESP32 is in deep sleep mode now. You can wake it up by pressing the pushbuttons.



Open the Serial Monitor at a baud rate of 115200. Press the pushbuttons to wake up the ESP32.

You should get something similar on the serial monitor.



Wrapping Up

In this article we've shown you how to use deep sleep with the ESP32 and different ways to wake it up. You can wake up the ESP32 using a timer, the touch pins, or a change on a GPIO state.

Let's summarize what we've seen about each wake up source:

Timer Wake Up

- To enable the timer wake up, you use the `esp_sleep_enable_timer_wakeup(time_in_us)` function;
- Use the `esp_deep_sleep_start()` function to start deep sleep.

Touch Wake Up

- To use the touch pins as a wake up source, first, you need to attach interrupts to the touch pins using: `touchAttachInterrupt(Touchpin, callback, Threshold)`
- Then, you enable the touch pins as a wake up source using: `esp_sleep_enable_touchpad_wakeup()`
- Finally, you use the `esp_deep_sleep_start()` function to put the ESP32 in deep sleep mode.

External Wake Up

- You can only use RTC GPIOs as an external wake up;
- You can use two different methods: ext0 and ext1;
- ext0 allows you to wake up the ESP32 using one single GPIO pin;
- ext1 allows you to wake up the ESP32 using several GPIO pins.

We hope you've found this guide useful. If you want to learn more about ESP32, make sure you check all our ESP32 projects and our ESP32 course.

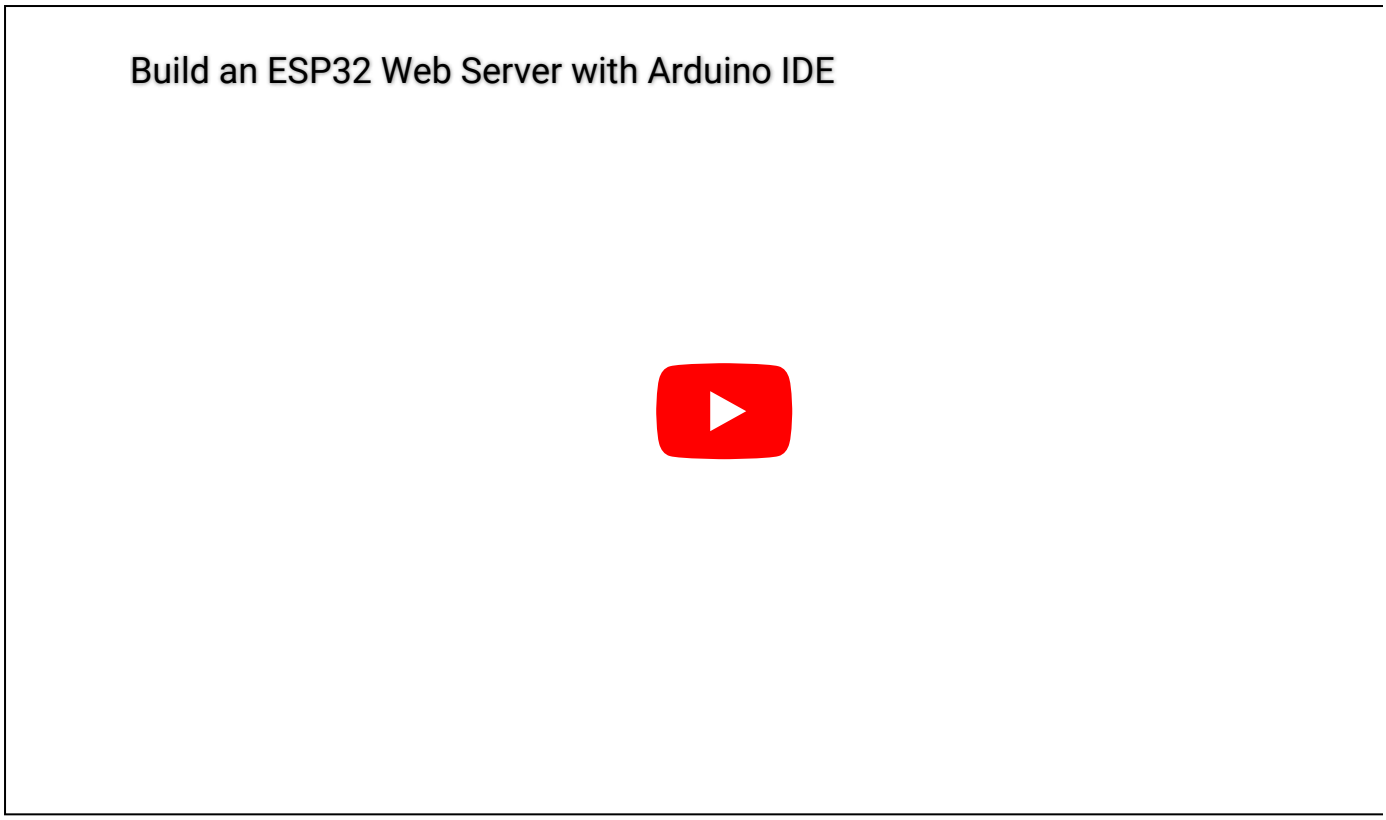
ESP32 Web Server – Arduino IDE

In this project you'll create a standalone web server with an ESP32 that controls outputs (two LEDs) using the Arduino IDE programming environment. The web server is mobile responsive and can be accessed with any device that as a browser on the local network. We'll show you how to create the web server and how the code works step-by-step.

If you want to learn more about the ESP32, read Getting Started Guide with ESP32.

Watch the Video Tutorial

This tutorial is available in video format (watch below) and in written format (continue reading this page).



Project Overview

Before going straight to the project, it is important to outline what our web server will do, so that it is easier to follow the steps later on.

- The web server you'll build controls two LEDs connected to the ESP32 GPIO 26 and GPIO 27 ;
- You can access the ESP32 web server by typing the ESP32 IP address on a browser in the local network;
- By clicking the buttons on your web server you can instantly change the state of each LED.

This is just a simple example to illustrate how to build a web server that controls outputs, the idea is to replace those LEDs with a relay, or any other electronic components you want.

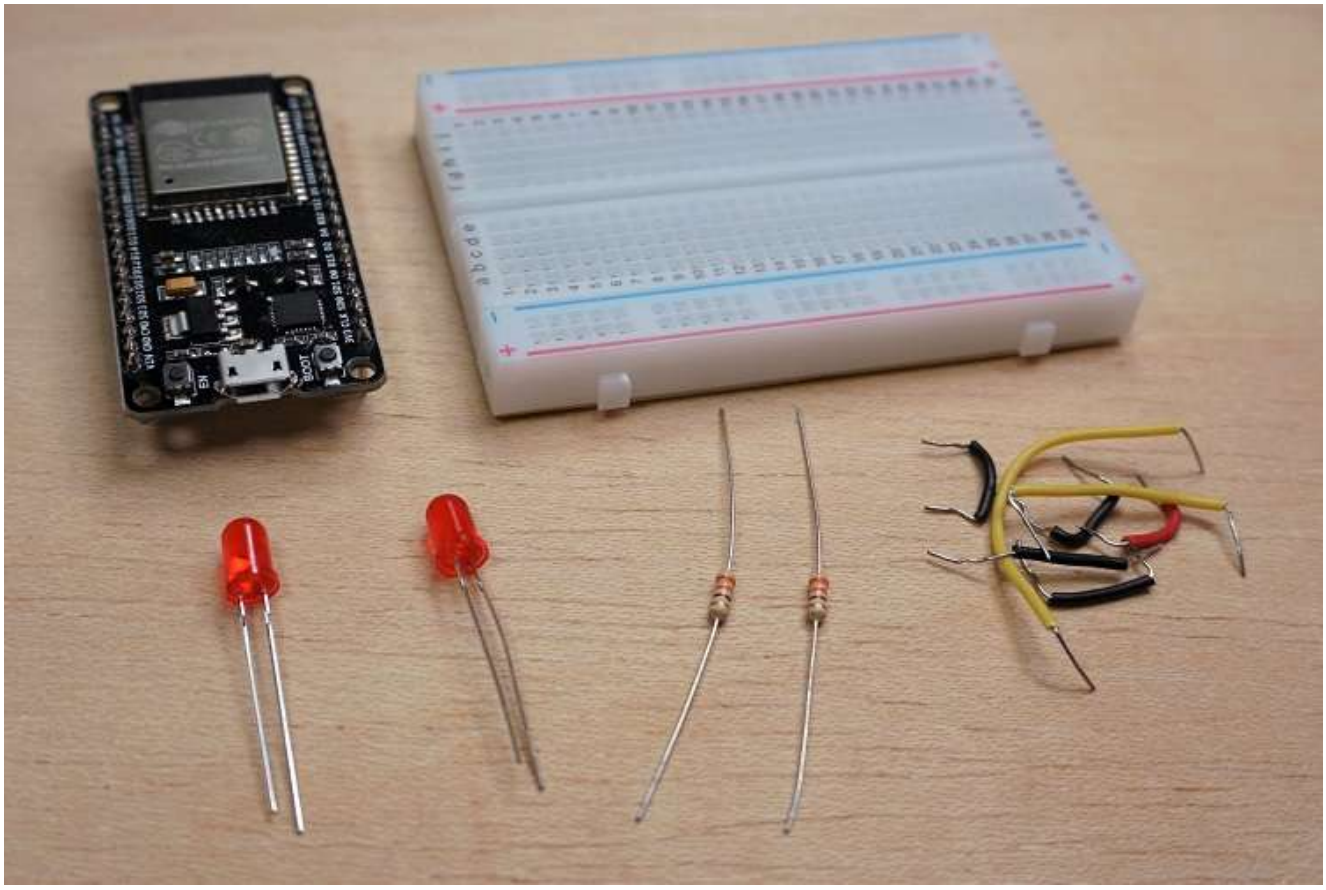
Installing the ESP32 board in Arduino IDE

There's an add-on for the Arduino IDE that allows you to program the ESP32 using the Arduino IDE and its programming language. Follow one of the following tutorials to prepare your Arduino IDE:

- **Windows instructions** – Installing the ESP32 Board in Arduino IDE
- **Mac and Linux instructions** – Installing the ESP32 Board in Arduino IDE

Parts Required

For this tutorial you'll need the following parts:



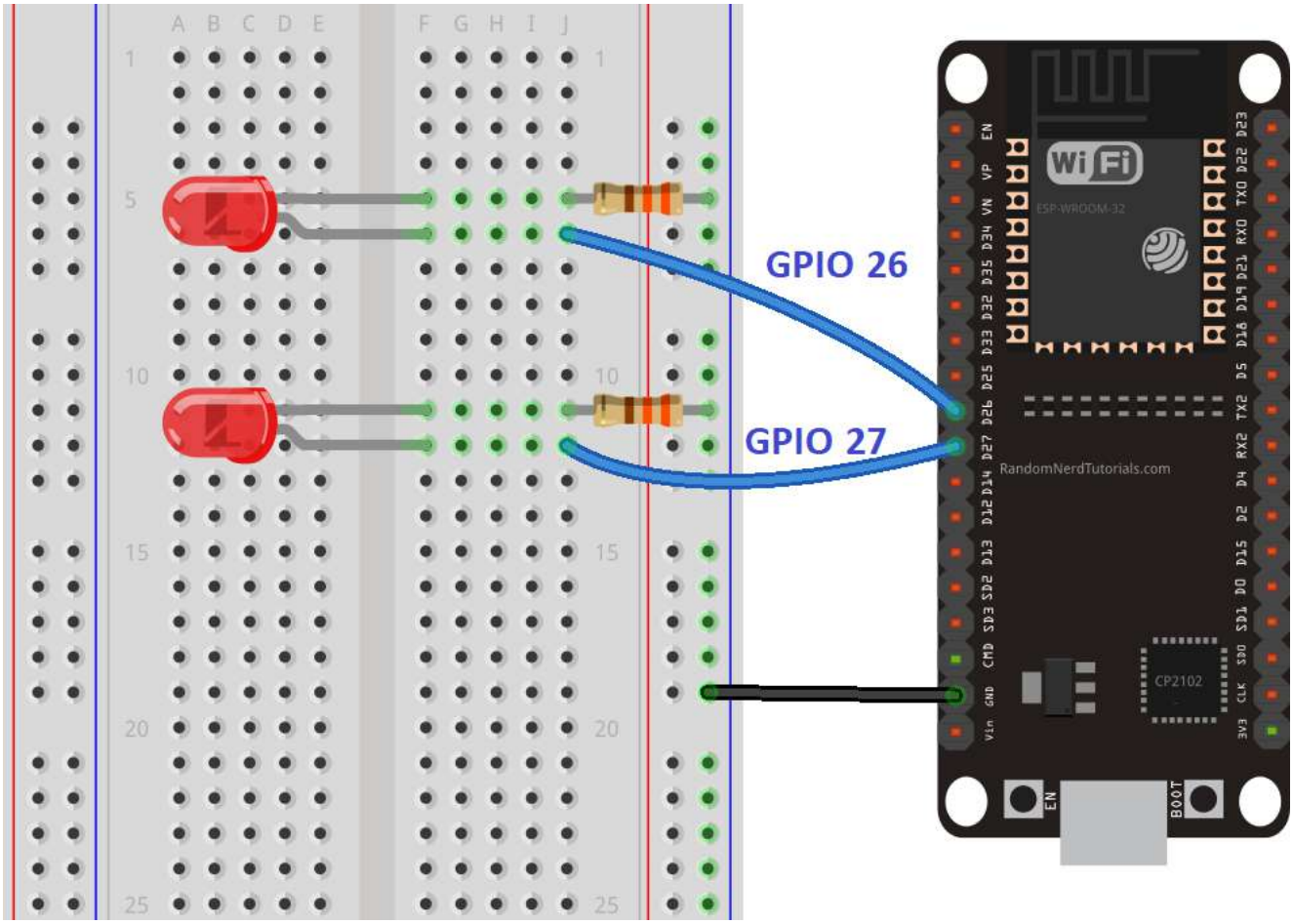
- ESP32 development board – read ESP32 Development Boards Review and Comparison
- 2x 5mm LED
- 2x 330 Ohm resistor
- Breadboard
- Jumper wires

You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!

Schematic

Start by building the circuit. Connect two LEDs to the ESP32 as shown in the following schematic diagram – one LED connected to GPIO 26 , and the other to GPIO 27.

Note: We're using the ESP32 DEVKIT DOIT board with 36 pins. Before assembling the circuit, make sure you check the pinout for the board you're using.



ESP32 Web Server Code

Here we provide the code that creates the ESP32 web server. Copy the following code to your Arduino IDE, but don't upload it yet. You need to make some changes to make it work for you.

```
/*********
  Rui Santos
  Complete project details at https://randomnerdtutorials.com
*********/

// Load Wi-Fi library
#include <WiFi.h>
```

```
// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

// Set web server port number to 80
WiFiServer server(80);

// Variable to store the HTTP request
String header;

// Auxiliar variables to store the current output state
String output26State = "off";
String output27State = "off";

// Assign output variables to GPIO pins
const int output26 = 26;
const int output27 = 27;

// Current time
```

View raw code

Setting Your Network Credentials

You need to modify the following lines with your network credentials: SSID and password. The code is well commented on where you should make the changes.

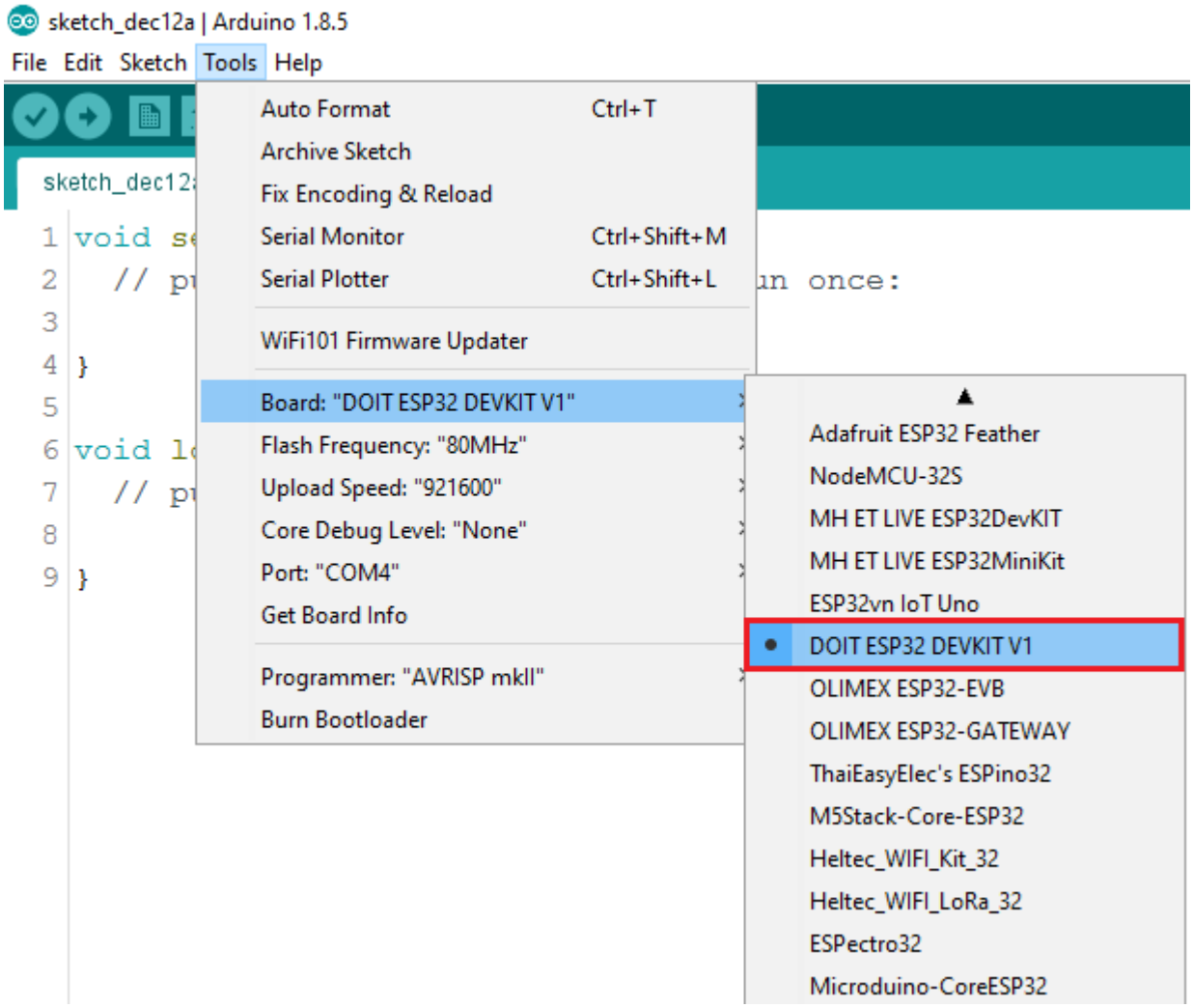
```
// Replace with your network credentials
const char* ssid     = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

Uploading the Code

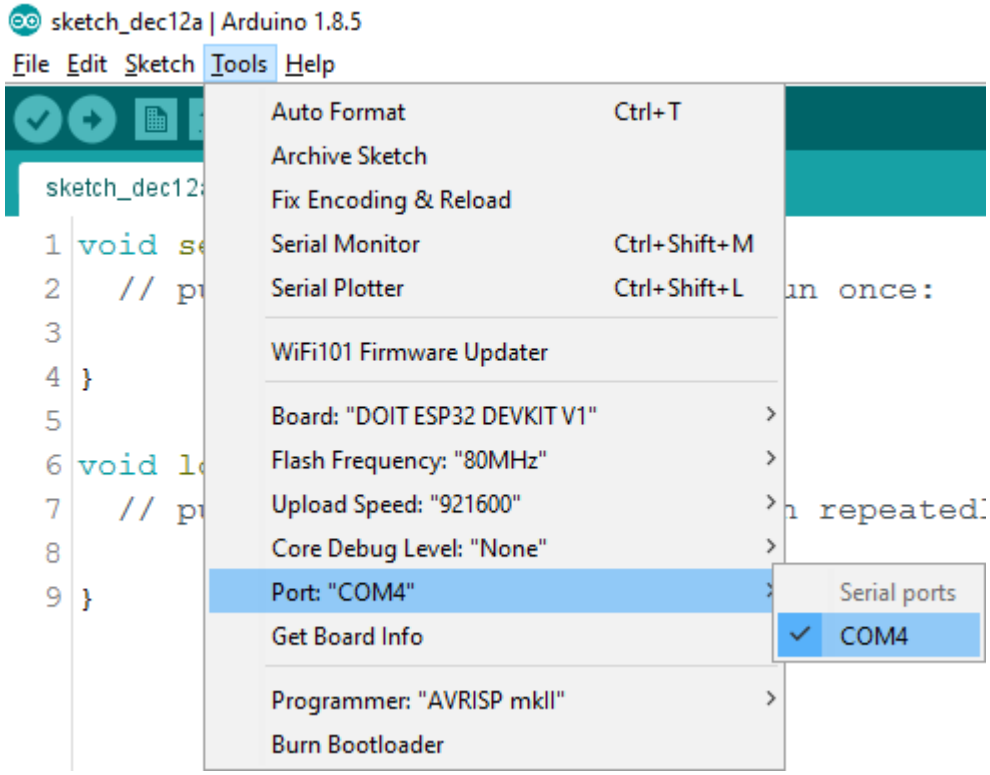
Now, you can upload the code and and the web server will work straight away. Follow the next steps to upload code to the ESP32:

1) Plug your ESP32 board in your computer;

2) In the Arduino IDE select your board in **Tools** > **Board** (in our case we're using the ESP32 DEVKIT DOIT board);

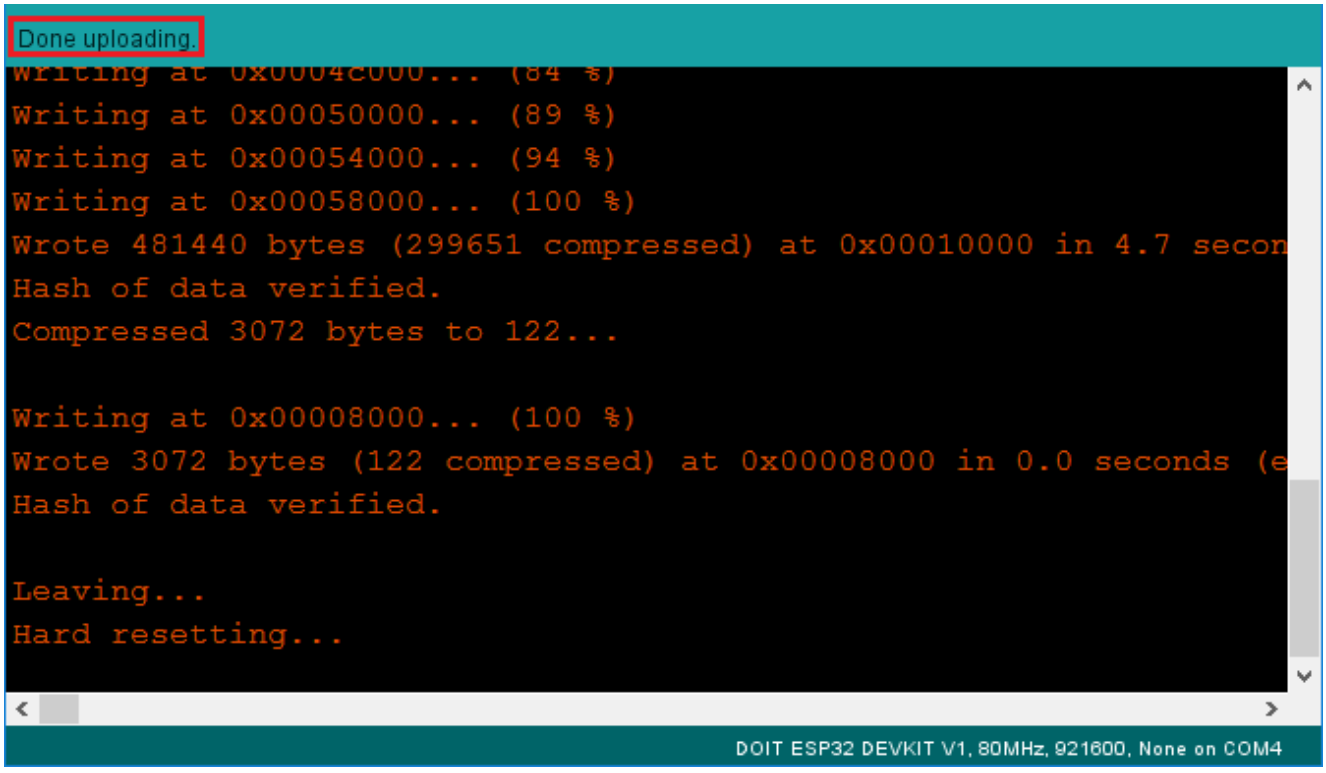


3) Select the COM port in **Tools** > **Port**.



4) Press the **Upload** button in the Arduino IDE and wait a few seconds while the code compiles and uploads to your board.

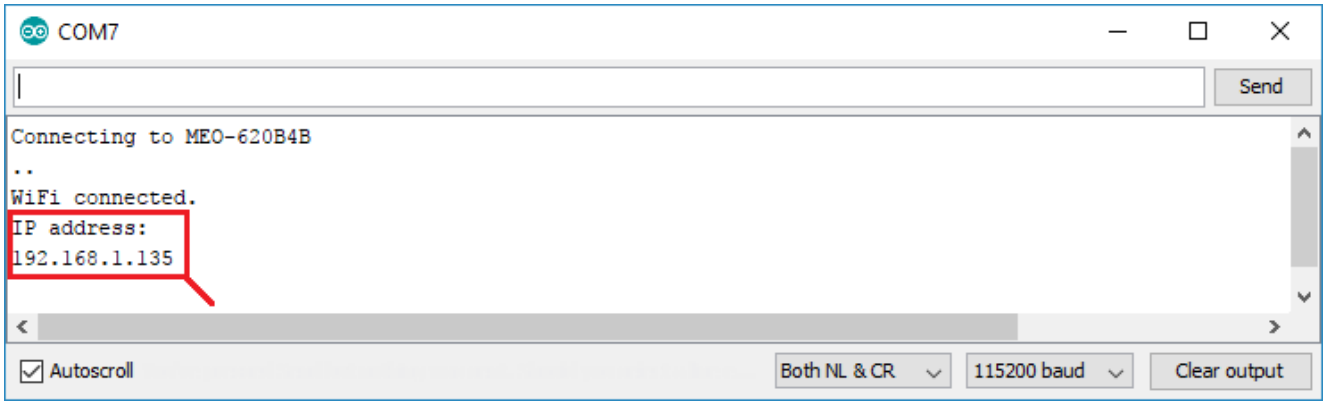
5) Wait for the "**Done uploading**" message.



Finding the ESP IP Address

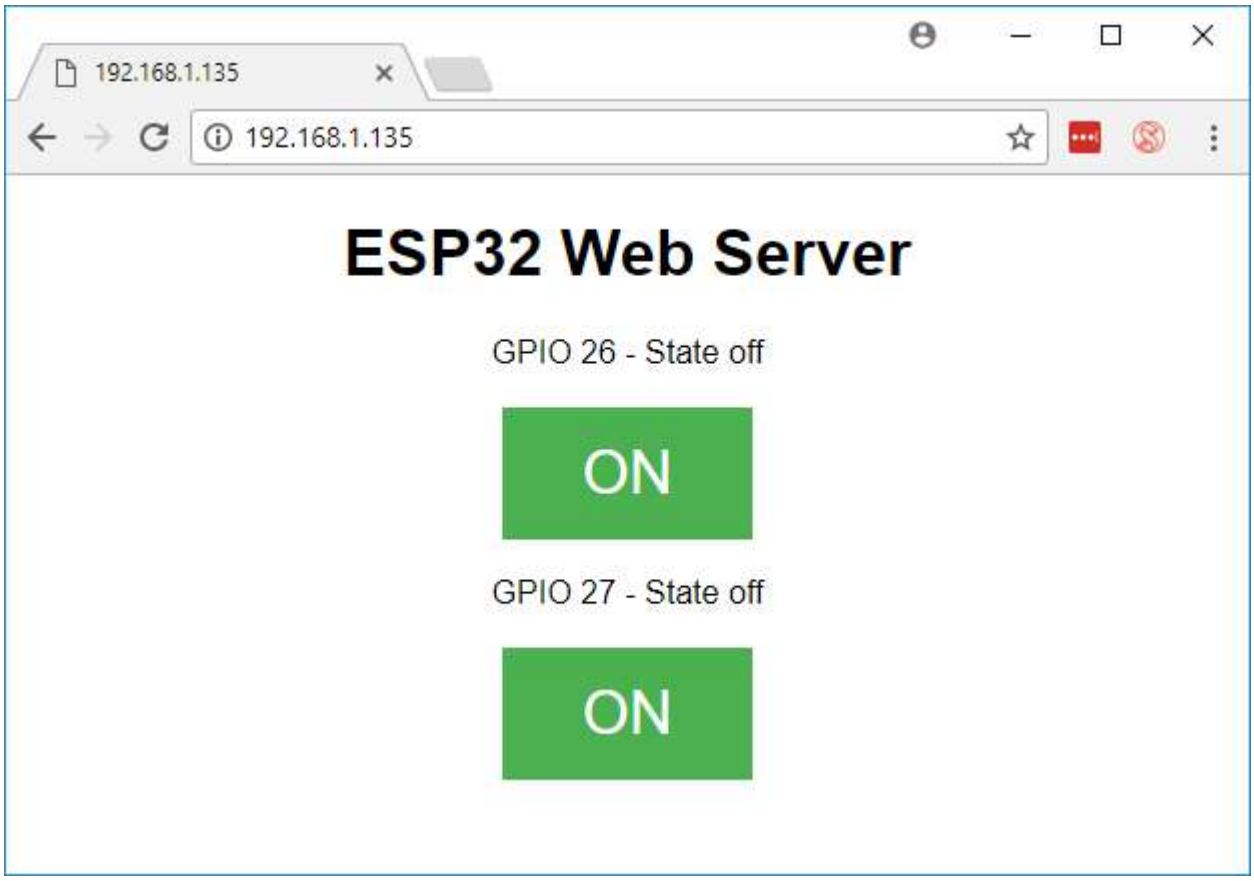
After uploading the code, open the Serial Monitor at a baud rate of 115200.

Press the ESP32 EN button (reset). The ESP32 connects to Wi-Fi, and outputs the ESP IP address on the Serial Monitor. Copy that IP address, because you need it to access the ESP32 web server.

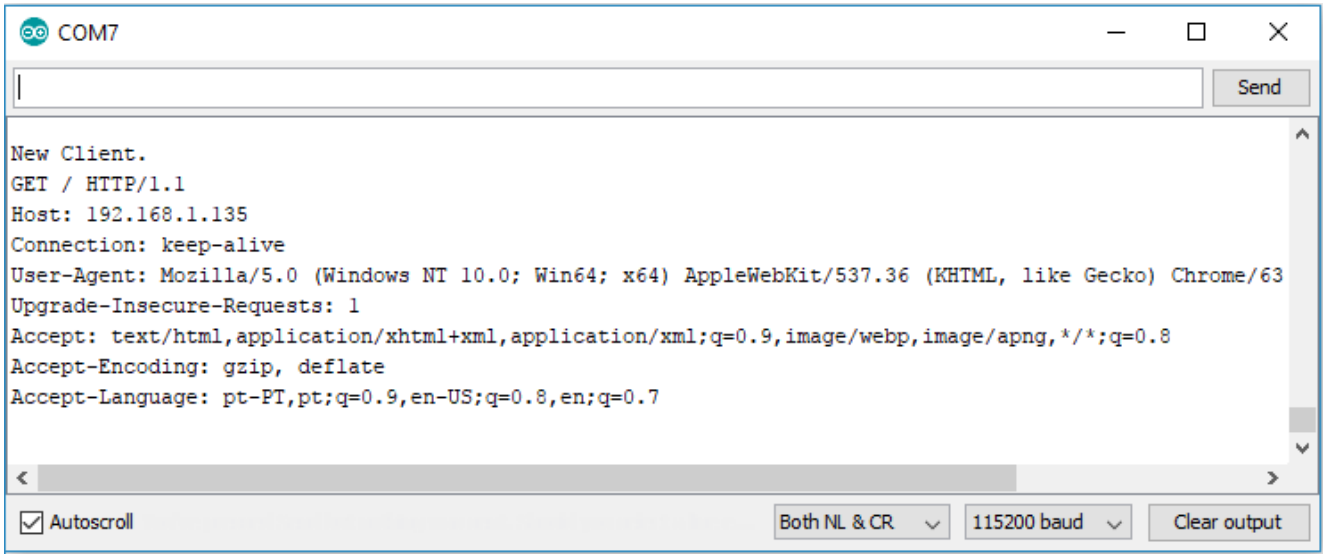


Accessing the Web Server

To access the web server, open your browser, paste the ESP32 IP address, and you'll see the following page. In our case it is **192.168.1.135**.



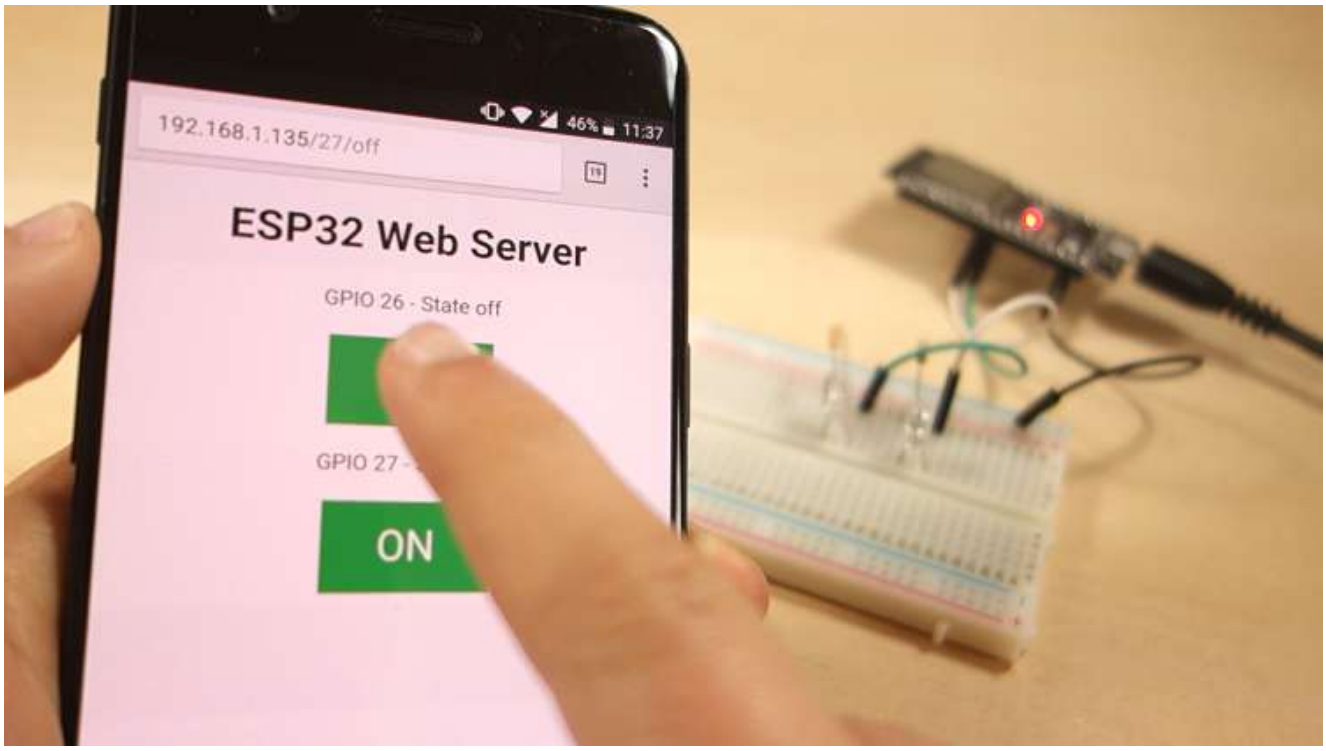
If you take a look at the Serial Monitor, you can see what's happening on the background. The ESP receives an HTTP request from a new client (in this case, your browser).



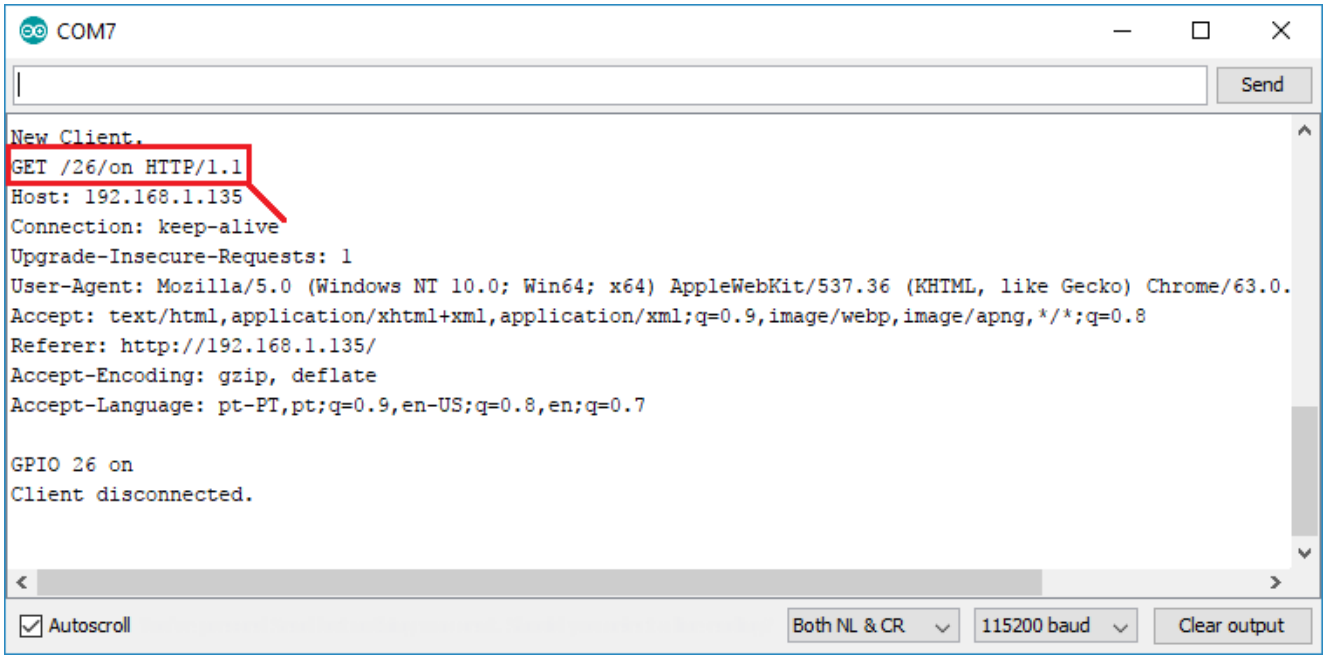
You can also see other information about the HTTP request.

Testing the Web Server

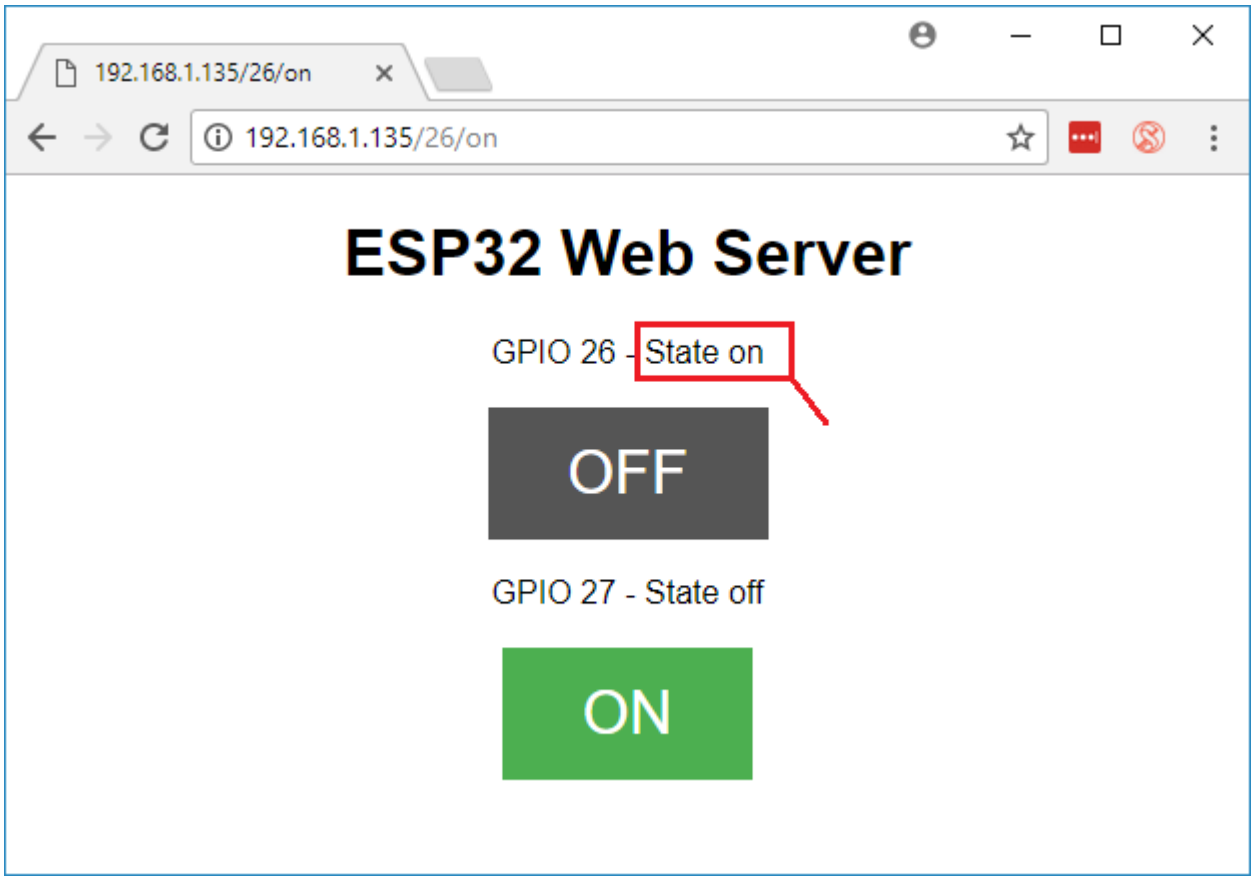
Now you can test if your web server is working properly. Click the buttons to control the LEDs.



At the same time, you can take a look at the Serial Monitor to see what's going on in the background. For example, when you click the button to turn GPIO 26 ON, ESP32 receives a request on the **/26/on** URL.



When the ESP32 receives that request, it turns the LED attached to GPIO 26 ON and updates its state on the web page.



The button for GPIO 27 works in a similar way. Test that it is working properly.

How the Code Works

In this section will take a closer look at the code to see how it works.

The first thing you need to do is to include the WiFi library.

```
#include <WiFi.h>
```

As mentioned previously, you need to insert your ssid and password in the following lines inside the double quotes.

```
const char* ssid = "";
const char* password = "";
```

Then, you set your web server to port 80.

```
WiFiServer server(80);
```

The following line creates a variable to store the header of the HTTP request:

```
String header;
```

Next, you create auxiliar variables to store the current state of your outputs. If you want to add more outputs and save its state, you need to create more variables.

```
String output26State = "off";
String output27State = "off";
```

You also need to assign a GPIO to each of your outputs. Here we are using GPIO 26 and GPIO 27 . You can use any other suitable GPIOs.

```
const int output26 = 26;
const int output27 = 27;
```

setup()

Now, let's go into the `setup()` . First, we start a serial communication at a baud rate of 115200 for debugging purposes.

```
Serial.begin(115200);
```

You also define your GPIOs as OUTPUTs and set them to LOW.

```
// Initialize the output variables as outputs
pinMode(output26, OUTPUT);
pinMode(output27, OUTPUT);

// Set outputs to LOW
digitalWrite(output26, LOW);
digitalWrite(output27, LOW);
```

The following lines begin the Wi-Fi connection with `WiFi.begin(ssid, password)` , wait for a successful connection and print the ESP IP address in the Serial Monitor.

```
// Connect to Wi-Fi network with SSID and password
Serial.print("Connecting to ");
Serial.println(ssid);
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
}
// Print local IP address and start web server
Serial.println("");
Serial.println("WiFi connected.");
Serial.println("IP address: ");
Serial.println(WiFi.localIP());
server.begin();
```

loop()

In the `loop()` we program what happens when a new client establishes a connection with the web server.

The ESP32 is always listening for incoming clients with the following line:

```
WiFiClient client = server.available(); // Listen for incoming cl
```

When a request is received from a client, we'll save the incoming data. The while loop that follows will be running as long as the client stays connected. We don't recommend changing the following part of the code unless you know exactly what you are doing.

```
if (client) { // If a new client connects,
  Serial.println("New Client."); // print a message out in the se
  String currentLine = ""; // make a String to hold incoming data
  while (client.connected()) { // loop while the client's connect
    if (client.available()) { // if there's bytes to read from th
      char c = client.read(); // read a byte, then
      Serial.write(c); // print it out the serial monitor
```

```
        header += c;
        if (c == '\n') { // if the byte is a newline character
        // if the current line is blank, you got two newline charac
        / that's the end of the client HTTP request, so send a resp
          if (currentLine.length() == 0) {
          // HTTP headers always start with a response code (e.g. H
          // and a content-type so the client knows what's coming,
            client.println("HTTP/1.1 200 OK");
            client.println("Content-type:text/html");
            client.println("Connection: close");
            client.println();
```

The next section of if and else statements checks which button was pressed in your web page, and controls the outputs accordingly. As we've seen previously, we make a request on different URLs depending on the button pressed.

```
// turns the GPIOs on and off
if (header.indexOf("GET /26/on") >= 0) {
  Serial.println("GPIO 26 on");
  output26State = "on";
  digitalWrite(output26, HIGH);
} else if (header.indexOf("GET /26/off") >= 0) {
  Serial.println("GPIO 26 off");
  output26State = "off";
  digitalWrite(output26, LOW);
} else if (header.indexOf("GET /27/on") >= 0) {
  Serial.println("GPIO 27 on");
  output27State = "on";
  digitalWrite(output27, HIGH);
} else if (header.indexOf("GET /27/off") >= 0) {
  Serial.println("GPIO 27 off");
  output27State = "off";
  digitalWrite(output27, LOW);
}
```

For example, if you've press the GPIO 26 ON button, the ESP32 receives a request on the **/26/ON URL** (we can see that that information on the HTTP header

on the Serial Monitor). So, we can check if the header contains the expression **GET /26/on**. If it contains, we change the `output26state` variable to ON, and the ESP32 turns the LED on.

This works similarly for the other buttons. So, if you want to add more outputs, you should modify this part of the code to include them.

Displaying the HTML web page

The next thing you need to do, is creating the web page. The ESP32 will be sending a response to your browser with some HTML code to build the web page.

The web page is sent to the client using this expressing `client.println()` . You should enter what you want to send to the client as an argument.

The first thing we should send is always the following line, that indicates that we are sending HTML.

```
<!DOCTYPE HTML><html>
```

Then, the following line makes the web page responsive in any web browser.

```
client.println("<head><meta name=\"viewport\" content=\"width=dev
```

And the following is used to prevent requests on the favicon. – You don't need to worry about this line.

```
client.println("<link rel=\"icon\" href=\"data:,\">");
```

Styling the Web Page

Next, we have some CSS text to style the buttons and the web page appearance. We choose the Helvetica font, define the content to be displayed as a block and aligned at the center.

```
client.println("<style>html { font-family: Helvetica; display: in
```

We style our buttons with the #4CAF50 color, without border, text in white color, and with this padding: 16px 40px. We also set the text-decoration to none, define the font size, the margin, and the cursor to a pointer.

```
client.println(".button { background-color: #4CAF50; border: none
client.println("text-decoration: none; font-size: 30px; margin: 2
```

We also define the style for a second button, with all the properties of the button we've defined earlier, but with a different color. This will be the style for the off button.

```
client.println(".button2 {background-color: #555555;}</style></he
```

Setting the Web Page First Heading

In the next line you can set the first heading of your web page. Here we have "**ESP32 Web Server**", but you can change this text to whatever you like.

```
// Web Page Heading
client.println("<h1>ESP32 Web Server</h1>");
```

Displaying the Buttons and Corresponding State

Then, you write a paragraph to display the GPIO 26 current state. As you can see we use the `output26State` variable, so that the state updates instantly when this variable changes.

```
client.println("<p>GPIO 26 - State " + output26State + "</p>");
```

Then, we display the on or the off button, depending on the current state of the GPIO. If the current state of the GPIO is off, we show the ON button, if not, we display the OFF button.

```
if (output26State=="off") {
  client.println("<p><a href=\"/26/on\"><button class=\"button\">
} else {
  client.println("<p><a href=\"/26/off\"><button class=\"button b
}
```

We use the same procedure for GPIO 27 .

Closing the Connection

Finally, when the response ends, we clear the `header` variable, and stop the connection with the client with `client.stop()` .

```
// Clear the header variable
header = "";
// Close the connection
client.stop();
```

Wrapping Up

In this tutorial we've shown you how to build a web server with the ESP32. We've shown you a simple example that controls two LEDs, but the idea is to replace those LEDs with a relay, or any other output you want to control. For more projects with ESP32, check the following tutorials:

- Build an All-in-One ESP32 Weather Station Shield
- ESP32 Servo Motor Web Server
- Getting Started with ESP32 Bluetooth Low Energy (BLE)
- More ESP32 tutorials

This is an excerpt from our course: Learn ESP32 with Arduino IDE. If you like ESP32 and you want to learn more, we recommend enrolling in Learn ESP32 with Arduino IDE course.

ESP32 with LoRa using Arduino IDE – Getting Started

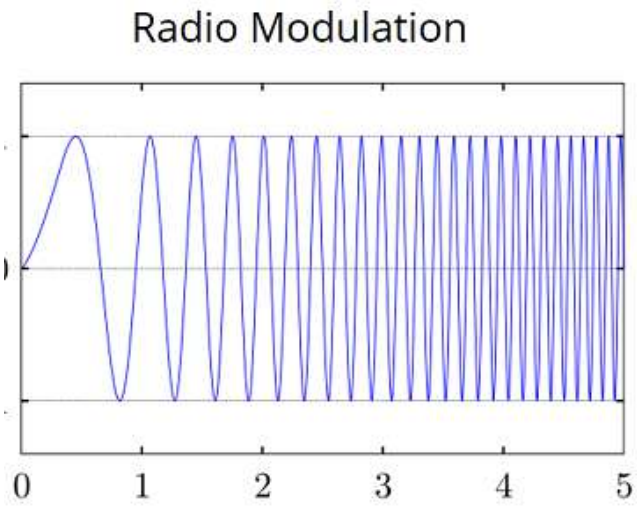
In this tutorial we'll explore the basic principles of LoRa, and how it can be used with the ESP32 for IoT projects using the Arduino IDE. To get you started, we'll also show you how to create a simple LoRa Sender and LoRa Receiver with the RFM95 transceiver module.

Introducing LoRa

For a quick introduction to LoRa, you can watch the video below, or you can scroll down for a written explanation.

What is LoRa?

LoRa is a wireless data communication technology that uses a radio modulation technique that can be generated by Semtech LoRa transceiver chips.



This modulation technique allows long range communication of small amounts of data (which means a low bandwidth), high immunity to interference, while minimizing power consumption. So, it allows long distance communication with low power requirements.

LoRa Frequencies

LoRa uses unlicensed frequencies that are available worldwide. These are the most widely used frequencies:

- 868 MHz for Europe
- 915 MHz for North America
- 433 MHz band for Asia

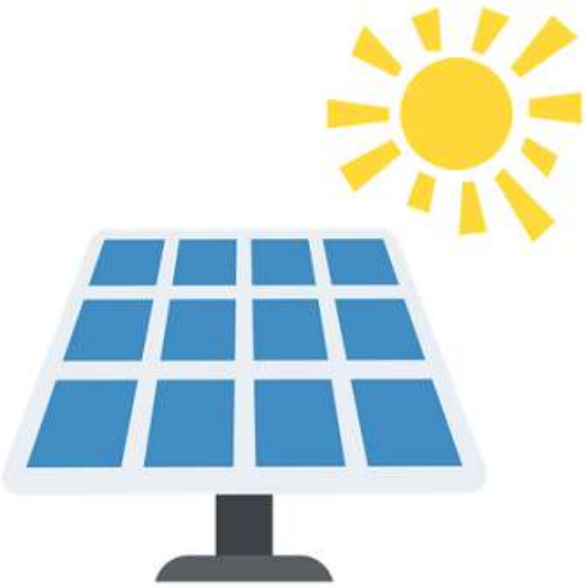
Because these bands are unlicensed, anyone can freely use them without paying or having to get a license. Check the frequencies used in your country.

LoRa Applications

LoRa long range and low power features, makes it perfect for battery-operated sensors and low-power applications in:

- Internet of Things (IoT)
- Smart home
- Machine-to-machine communication
- And much more…

So, LoRa is a good choice for sensor nodes running on a coil cell or solar powered, that transmit small amounts of data.

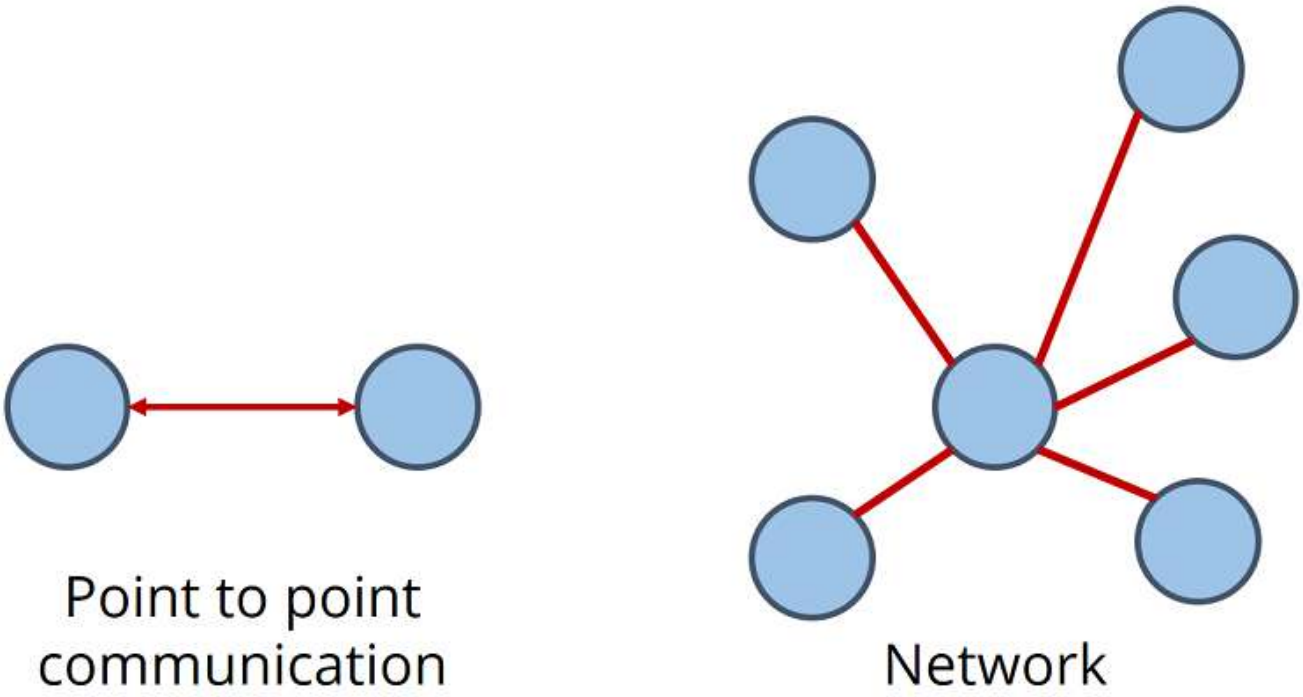


Keep in mind that LoRa is not suitable for projects that:

- Require high data-rate transmission;
- Need very frequent transmissions;
- Or are in highly populated networks.

LoRa Topologies

You can use LoRa in:

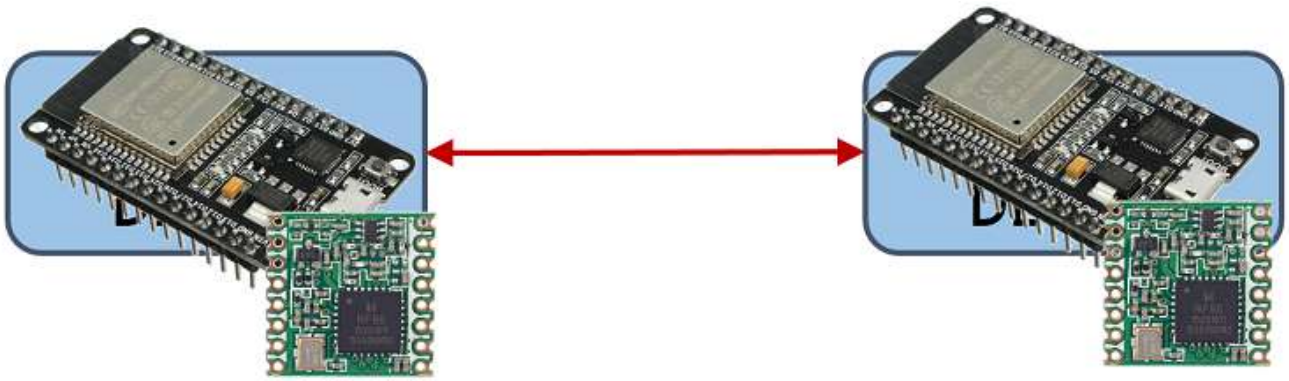


- Point to point communication
- Or build a LoRa network (using LoRaWAN for example)

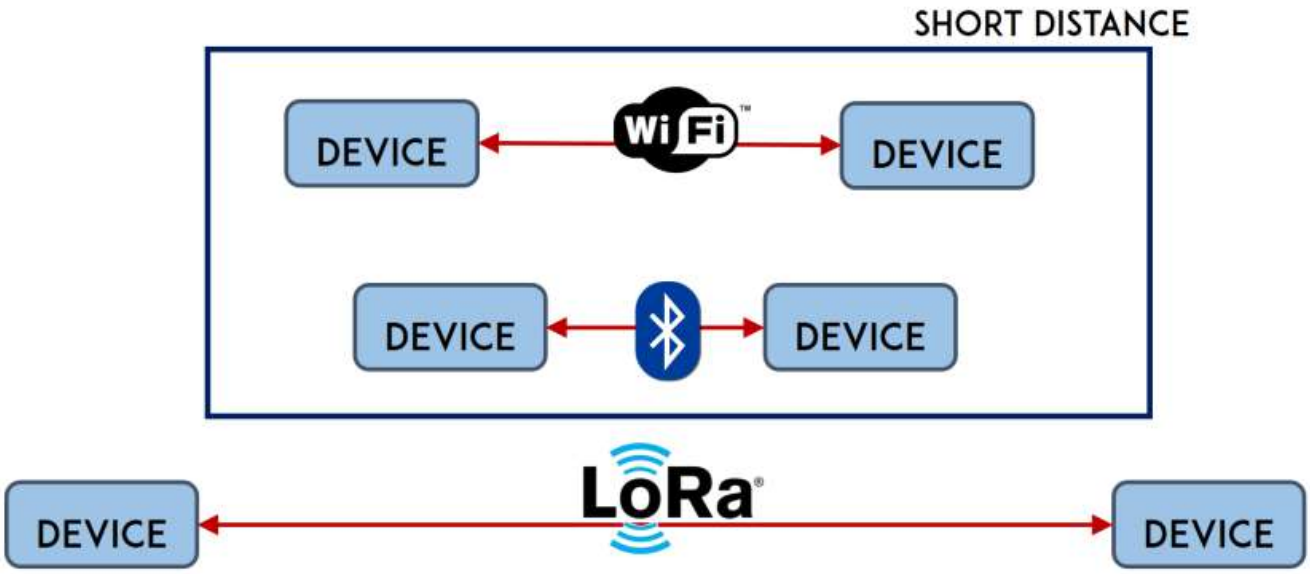
Point to Point Communication

In point to point communication, two LoRa enabled devices talk with each other using RF signals.

For example, this is useful to exchange data between two ESP32 boards equipped with LoRa transceiver chips that are relatively far from each other or in environments without Wi-Fi coverage.



Unlike Wi-Fi or Bluetooth that only support short distance communication, two LoRa devices with a proper antenna can exchange data over a long distance.



You can easily configure your ESP32 with a LoRa chip to transmit and receive data reliably at more than 200 meters distance (you can get better results depending on your enviroment and LoRa settings). There are also other LoRa solutions that easily have a range of more than 30Km.

LoRaWAN

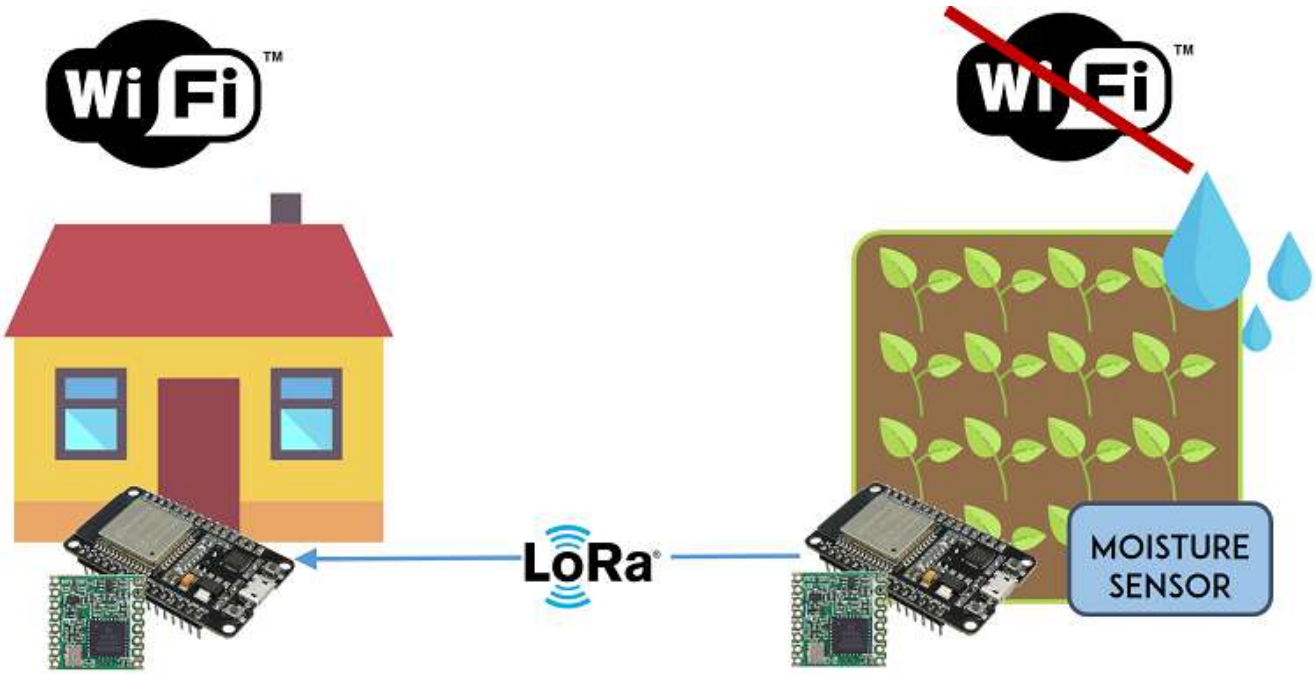
You can also build a LoRa network using LoRaWAN.

The LoRaWAN protocol is a Low Power Wide Area Network (LPWAN) specification derived from LoRa technology standardized by the LoRa Alliance. We won't explore LoRaWAN in this tutorial, but for more information you can check the LoRa Alliance and The Things Network websites.

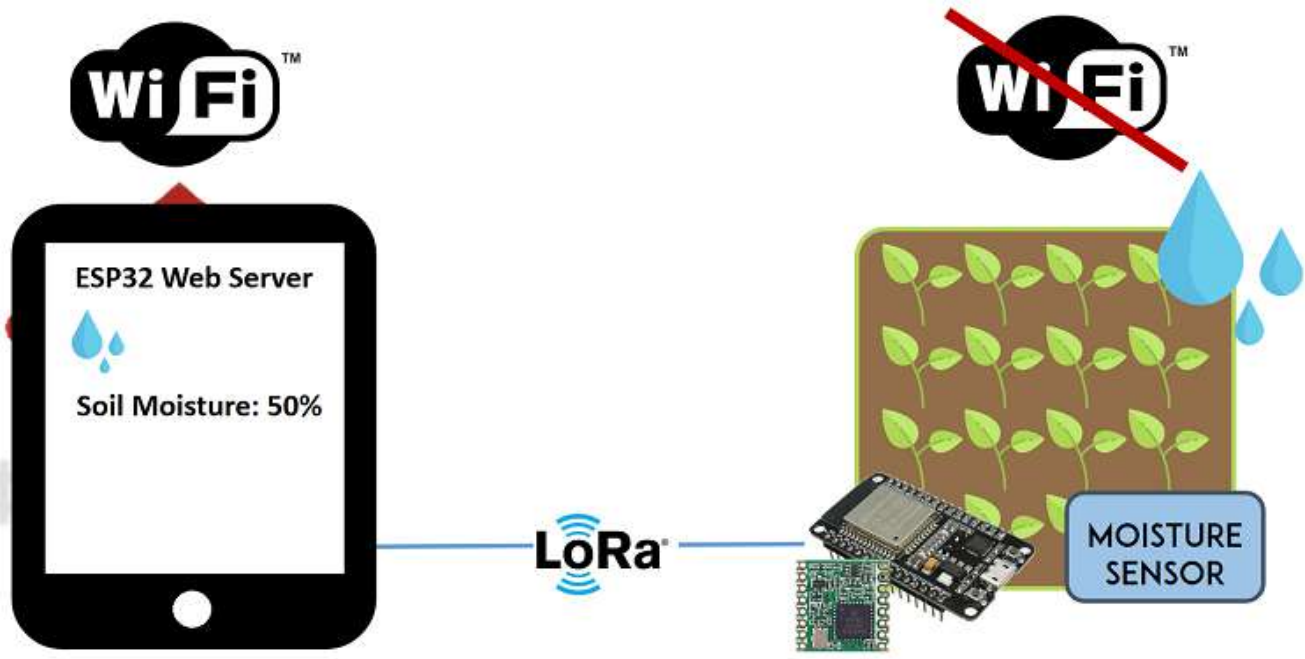
How can LoRa be useful in your home automation projects?

Let's take a look at a practical application.

Imagine that you want to measure the moisture in your field. Although, it is not far from your house, it probably doesn't have Wi-Fi coverage. So, you can build a sensor node with an ESP32 and a moisture sensor, that sends the moisture readings once or twice a day to another ESP32 using LoRa.



The later ESP32 has access to Wi-Fi, and it can run a web server that displays the moisture readings.



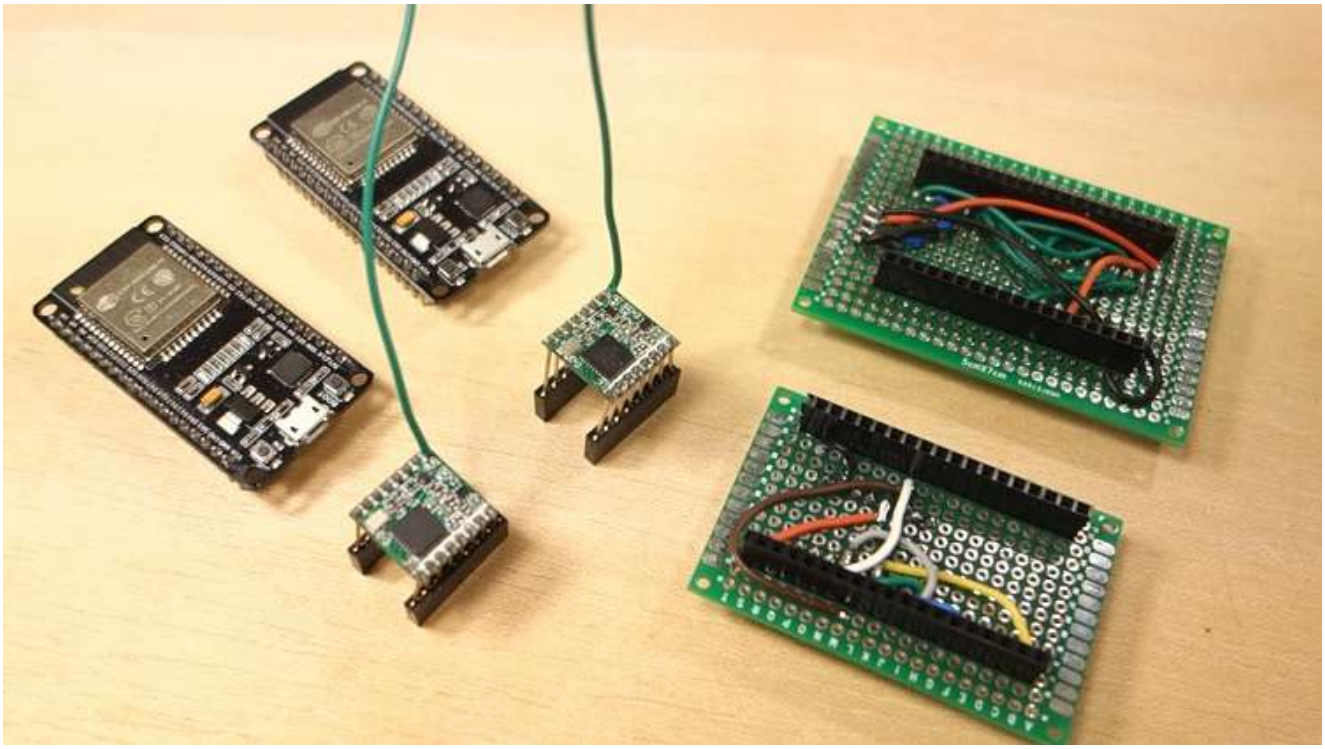
This is just an example that illustrates how you can use the LoRa technology in your ESP32 projects.

Note: we teach how to build this project on our "Learn ESP32 with Arduino IDE" course. It is Project 4 on the Table of Contents: **LoRa Long Range Sensor Monitoring – Reporting Sensor Readings from Outside: Soil Moisture and Temperature.** Check the course page for more details.

ESP32 with LoRa

In this section we'll show you how to get started with LoRa with your ESP32 using Arduino IDE. As an example, we'll build a simple LoRa Sender and a LoRa Receiver.

The LoRa Sender will be sending a "hello" message followed by a counter for testing purposes. This message can be easily replaced with useful data like sensor readings or notifications.



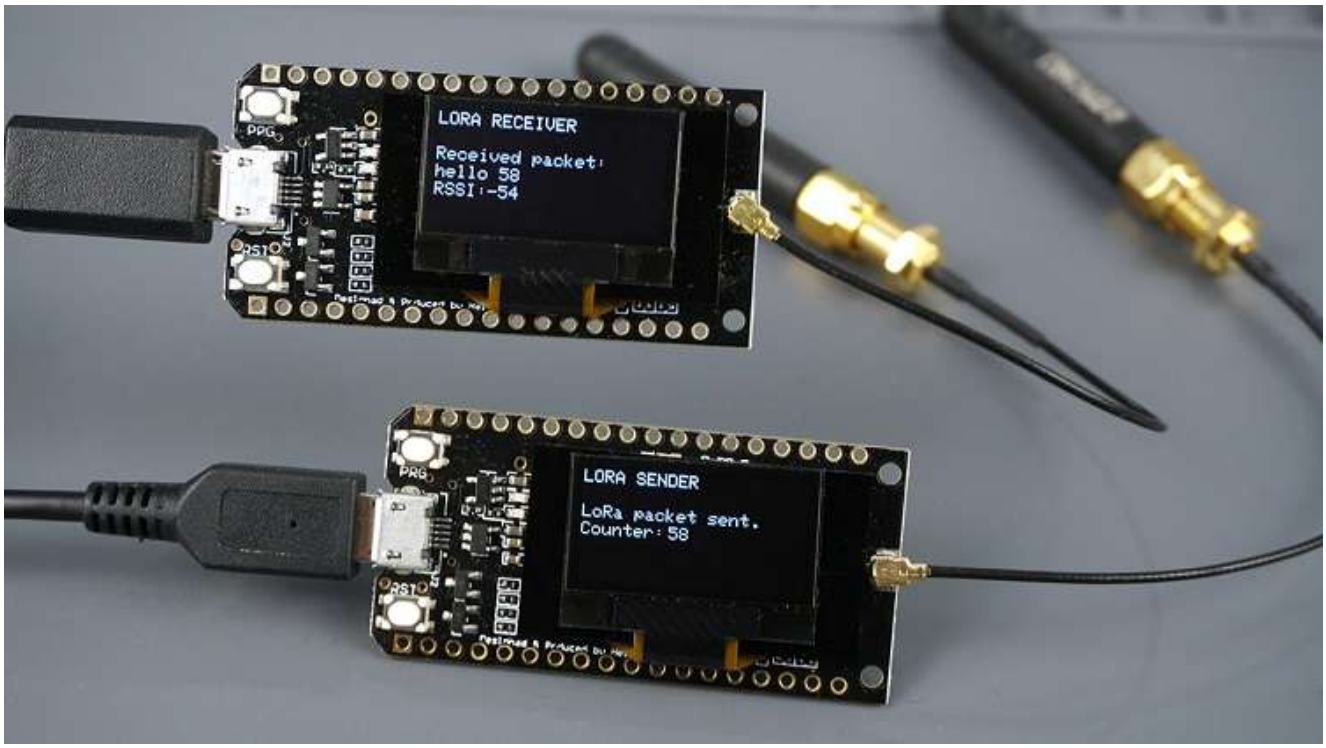
To follow this part you need the following components:

- 2x ESP32 DOIT DEVKIT V1 Board
- 2x LoRa Transceiver modules (RFM95)
- RFM95 LoRa breakout board (optional)
- Jumper wires
- Breadboard or stripboard

Alternative:

- 2x TTGO LoRa32 SX1276 OLED

Instead of using an ESP32 and a separated LoRa transceiver module, there are ESP32 development boards with a LoRa chip and an OLED built-in, which makes wiring much simpler. If you have one of those boards, you can follow: TTGO LoRa32 SX1276 OLED Board: Getting Started with Arduino IDE.



You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!

Preparing the Arduino IDE

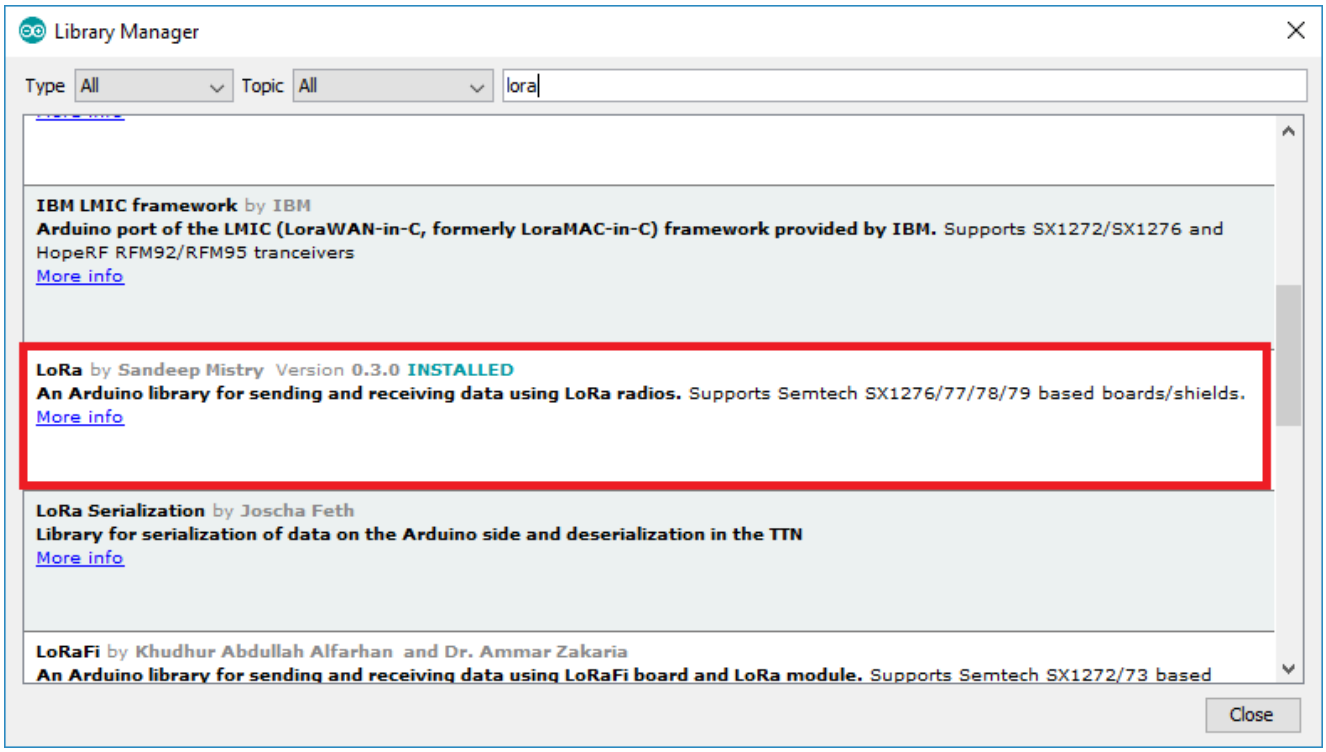
There's an add-on for the Arduino IDE that allows you to program the ESP32 using the Arduino IDE and its programming language. Follow one of the next tutorials to prepare your Arduino IDE to work with the ESP32, if you haven't already.

- **Windows** instructions – ESP32 Board in Arduino IDE
- **Mac and Linux** instructions – ESP32 Board in Arduino IDE

Installing the LoRa Library

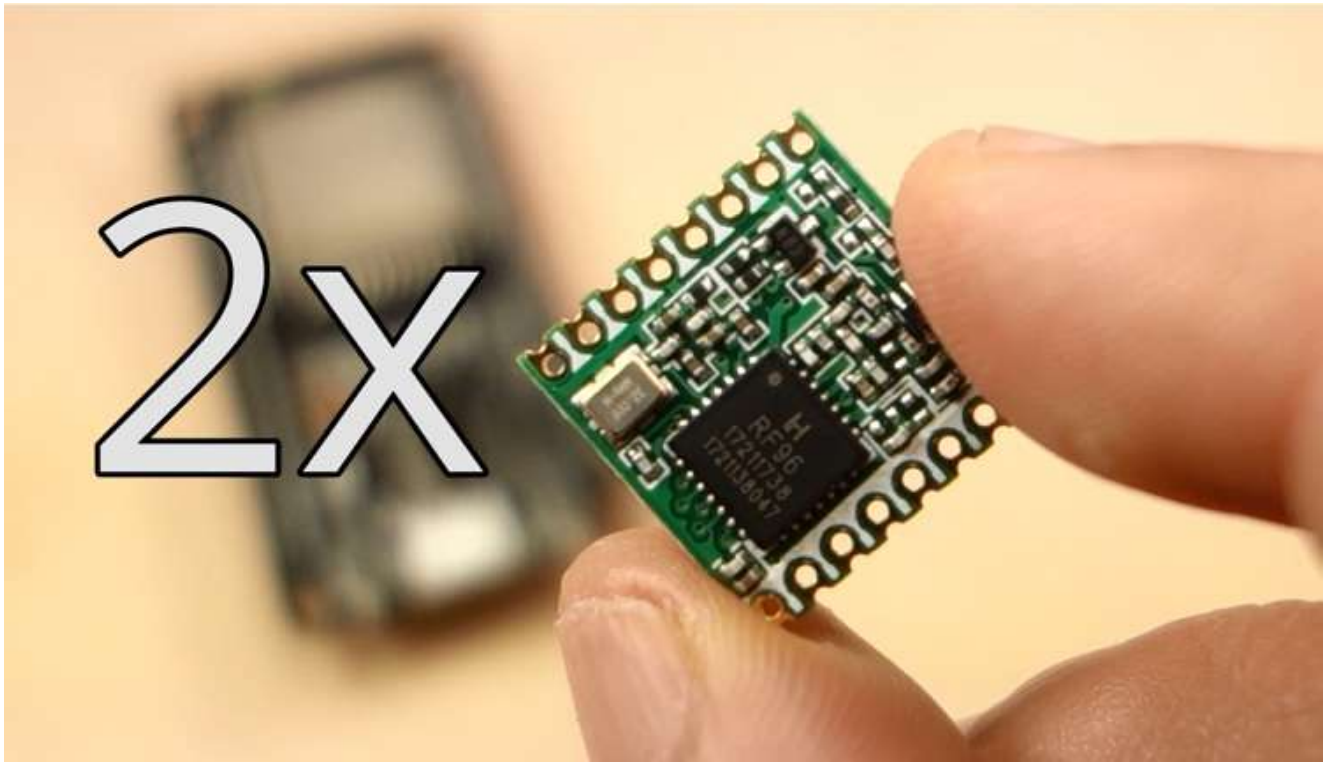
There are several libraries available to easily send and receive LoRa packets with the ESP32. In this example we'll be using the arduino-LoRa library by sandeep mistry.

Open your Arduino IDE, and go to **Sketch** > **Include Library** > **Manage Libraries** and search for "**LoRa**". Select the LoRa library highlighted in the figure below, and install it.



Getting LoRa Tranceiver Modules

To send and receive LoRa messages with the ESP32 we'll be using the RFM95 transceiver module. All LoRa modules are transceivers, which means they can send and receive information. You'll need 2 of them.



You can also use other compatible modules like Semtech SX1276/77/78/79 based boards including: RFM96W, RFM98W, etc…

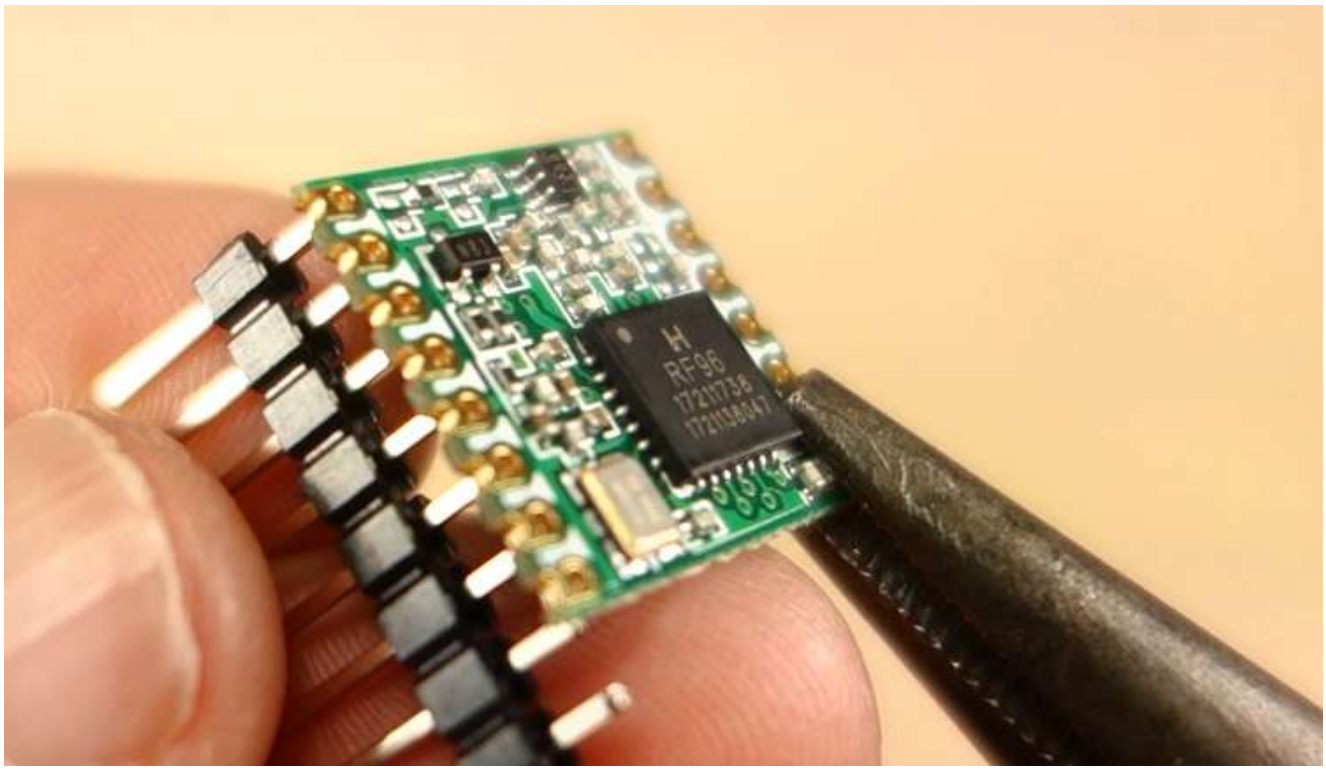
Alternatively, there are ESP32 boards with LoRa and OLED display built-in like the ESP32 Heltec Wifi Module, or the TTGO LoRa32 board.

Before getting your LoRa transceiver module, make sure you check the correct frequency for your location. You can visit the following web page to learn more about RF signals and regulations according to each country. For example, in Portugal we can use a frequency between 863 and 870 MHz or we can use 433MHz. For this project, we'll be using an RFM95 that operates at 868 MHz.

Preparing the RFM95 Transceiver Module

If you have an ESP32 development board with LoRa built-in, you can skip this step.

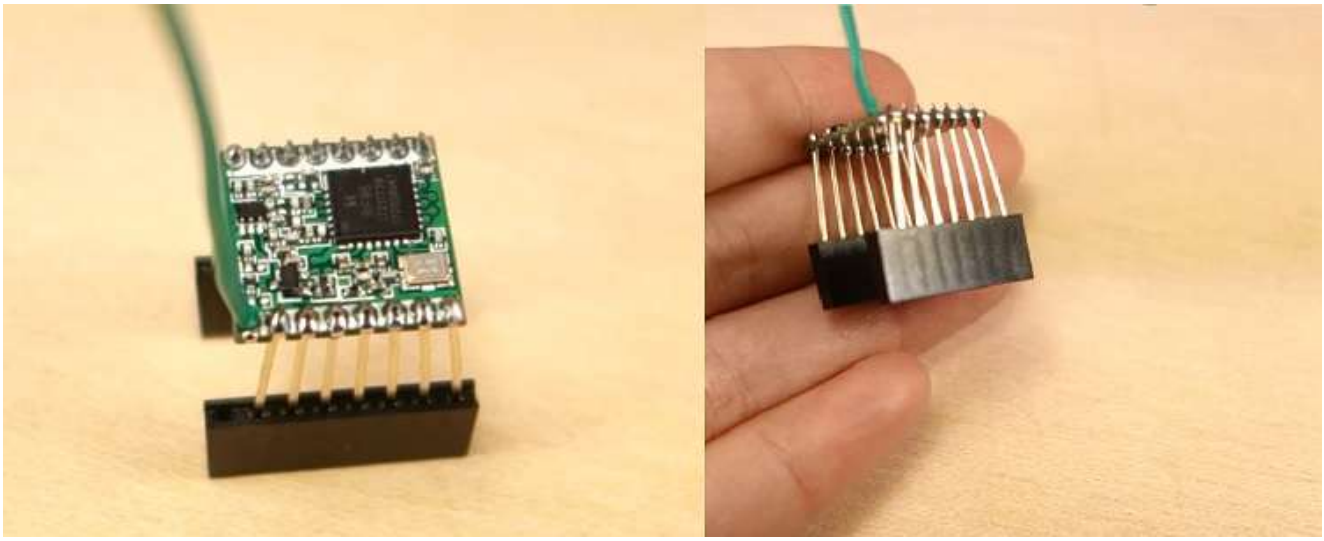
The RFM95 transceiver isn't breadboard friendly. A common row of 2.54mm header pins won't fit on the transceiver pins. The spaces between the connections are shorter than usual.



There are a few options that you can use to access the transceiver pins.

- You may solder some wires directly to the transceiver;
- Break header pins and solder each one separately;
- Or you can buy a breakout board that makes the pins breadboard friendly.

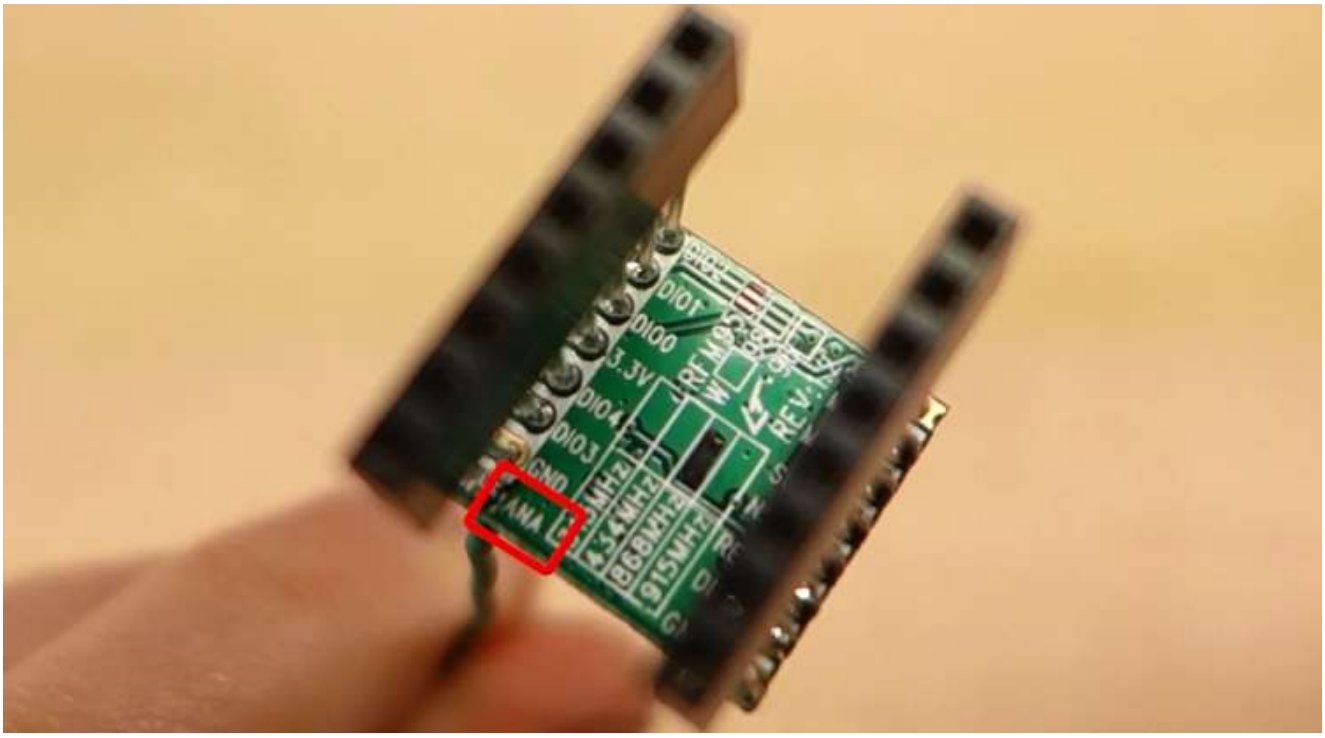
We've soldered a header to the module as shown in the figure below.



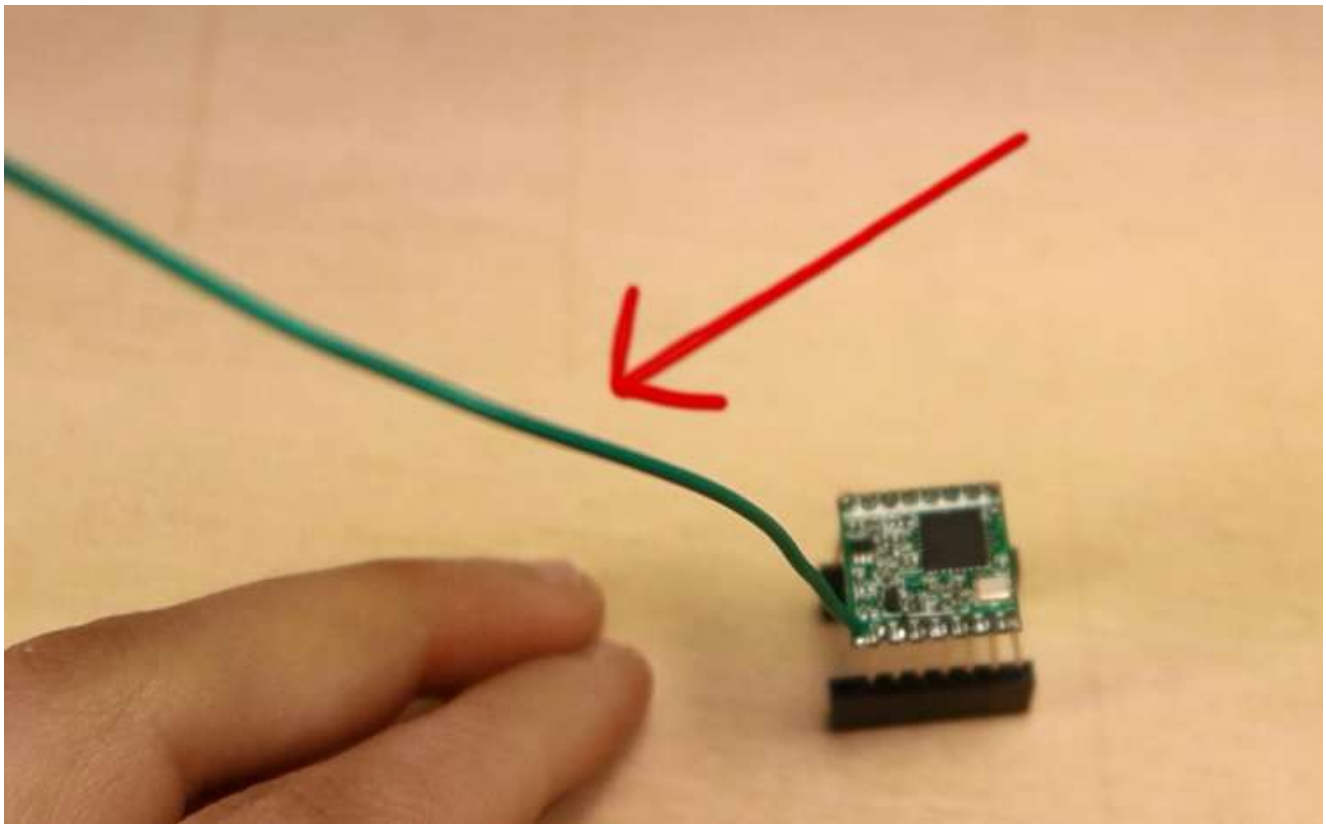
This way you can access the module's pins with regular jumper wires, or even put some header pins to connect them directly to a stripboard or breadboard.

Antenna

The RFM95 transceiver chip requires an external antenna connected to the ANA pin.



You can connect a "real" antenna, or you can make one yourself by using a conductive wire as shown in the figure below. Some breakout boards come with a special connector to add a proper antenna.



The wire length depends on the frequency:

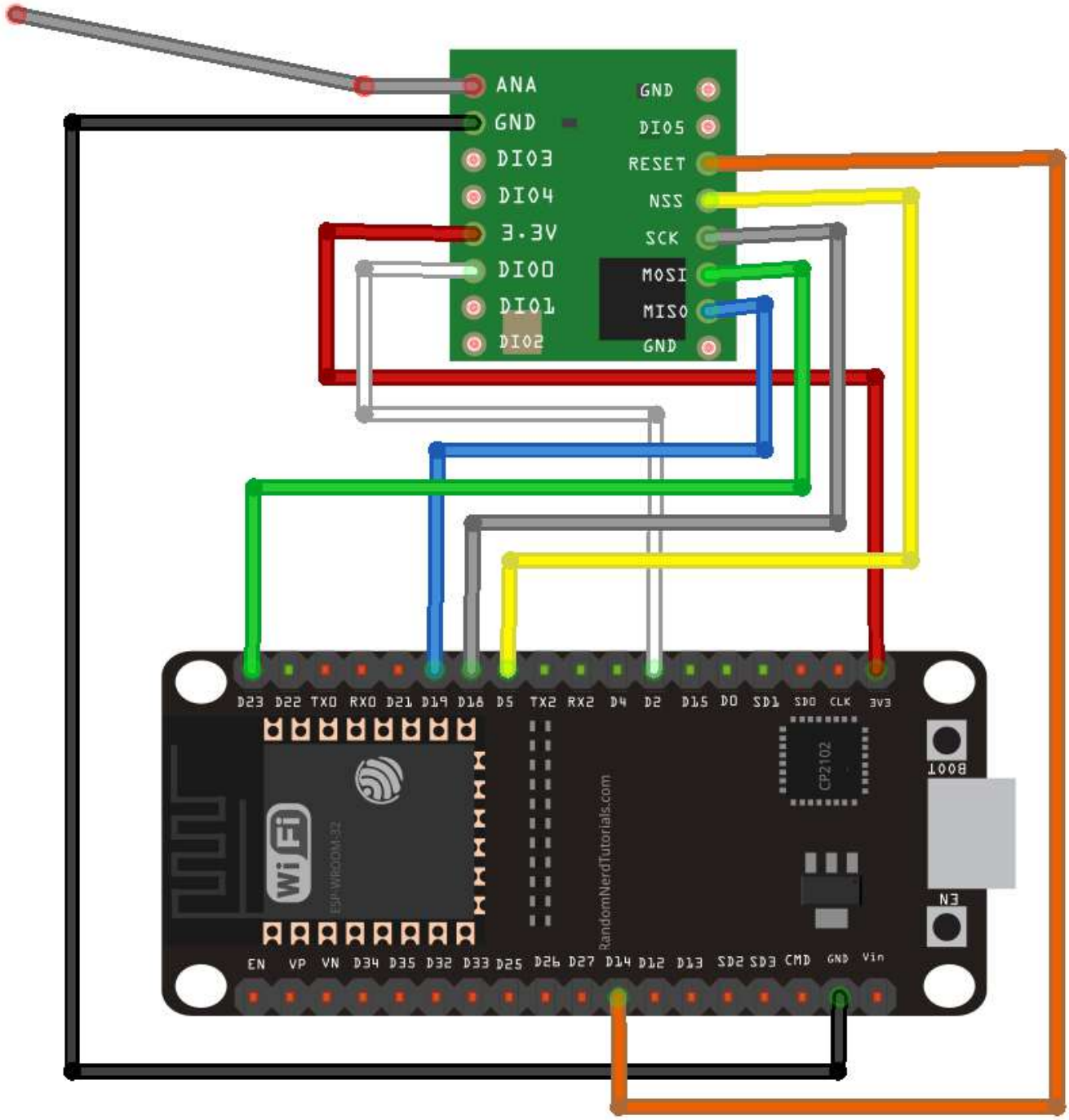
- 868 MHz: 86,3 mm (3.4 inch)
- 915 MHz: 81,9 mm (3.22 inch)
- 433 MHz: 173,1 mm (6.8 inch)

For our module we need to use a 86,3 mm wire soldered directly to the transceiver's ANA pin. Note that using a proper antenna will extend the communication range.

Important: you MUST attach an antenna to the module.

Wiring the RFM95 LoRa Transceiver Module

The RFM95 LoRa transceiver module communicates with the ESP32 using SPI communication protocol. So, we'll use the ESP32 default SPI pins. Wire both ESP32 boards to the corresponding transceiver modules as shown in the next schematic diagram:



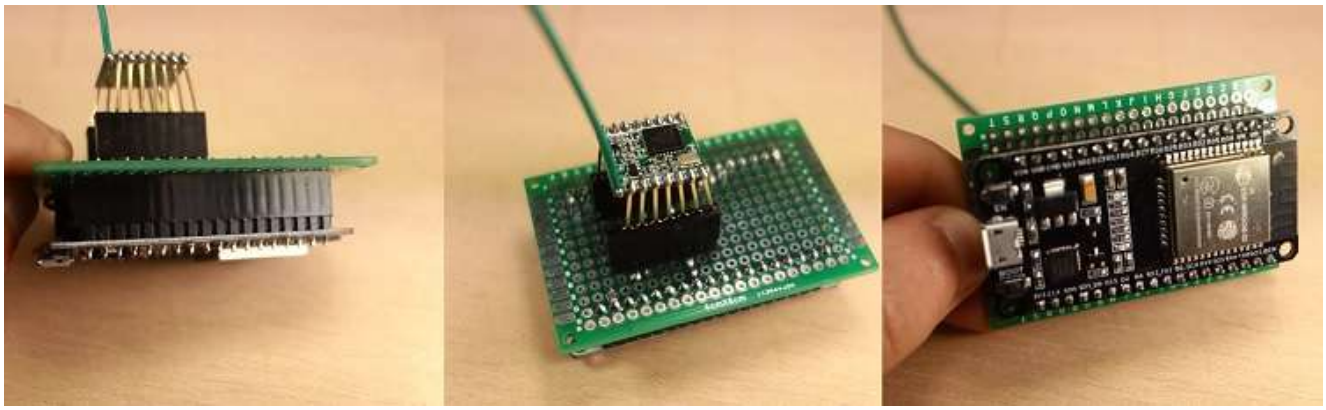
Here’s the connections between the RFM95 LoRa transceiver module and the ESP32:

- ANA: Antenna
- GND: GND
- DIO3: don’t connect
- DIO4: don’t connect
- 3.3V: 3.3V
- DIO0: GPIO 2
- DIO1: don’t connect
- DIO2: don’t connect
- GND: don’t connect

- DIO5: don't connect
- RESET: GPIO 14
- NSS: GPIO 5
- SCK: GPIO 18
- MOSI: GPIO 23
- MISO: GPIO 19
- GND: don't connect

Note: the RFM95 transceiver module has 3 GND pins. It doesn't matter which one you use, but you need to connect at least one.

For practical reasons we've made this circuit on a stripboard. It's easier to handle, and the wires don't disconnect. You may use a breadboard if you prefer.



The LoRa Sender Sketch

Open your Arduino IDE and copy the following code. This sketch is based on an example from the LoRa library. It transmits messages every 10 seconds using LoRa. It sends a "hello" followed by a number that is incremented in every message.

```
/*********
  Modified from the examples of the Arduino LoRa library
  More resources: https://randomnerdtutorials.com
*********/

#include <SPI.h>
#include <LoRa.h>

//define the pins used by the transceiver module
```

```
#define ss 5
#define rst 14
#define dio0 2

int counter = 0;

void setup() {
  //initialize Serial Monitor
  Serial.begin(115200);
  while (!Serial);
  Serial.println("LoRa Sender");

  //setup LoRa transceiver module
  LoRa.setPins(ss, rst, dio0);

  //replace the LoRa.begin(---E-) argument with your location's
  //433E6 for Asia
  //866E6 for Europe
```

View raw code

Let's take a quick look at the code.

It starts by including the needed libraries.

```
#include <SPI.h>
#include <LoRa.h>
```

Then, define the pins used by your LoRa module. If you've followed the previous schematic, you can use the pin definition used in the code. If you're using an ESP32 board with LoRa built-in, check the pins used by the LoRa module in your board and make the right pin assignment.

```
#define ss 5
#define rst 14
#define dio0 2
```

You initialize the `counter` variable that starts at 0;

```
int counter = 0;
```

In the `setup()`, you initialize a serial communication.

```
Serial.begin(115200);
while (!Serial);
```

Set the pins for the LoRa module.

```
LoRa.setPins(ss, rst, dio0);
```

And initialize the transceiver module with a specified frequency.

```
while (!LoRa.begin(866E6)) {
  Serial.println(".");
  delay(500);
}
```

You might need to change the frequency to match the frequency used in your location. Choose one of the following options:

- 433E6
- 866E6
- 915E6

LoRa transceiver modules listen to packets within its range. It doesn't matter where the packets come from. To ensure you only receive packets from your sender, you can set a sync word (ranges from 0 to 0xFF).

```
LoRa.setSyncWord(0xF3);
```

Both the receiver and the sender need to use the same sync word. This way, the receiver ignores any LoRa packets that don't contain that sync word.

Next, in the `loop()` you send the LoRa packets. You initialize a packet with the `beginPacket()` method.

```
LoRa.beginPacket();
```

You write data into the packet using the `print()` method. As you can see in the following two lines, we're sending a hello message followed by the counter.

```
LoRa.print("hello ");
LoRa.print(counter);
```

Then, close the packet with the `endPacket()` method.

```
LoRa.endPacket();
```

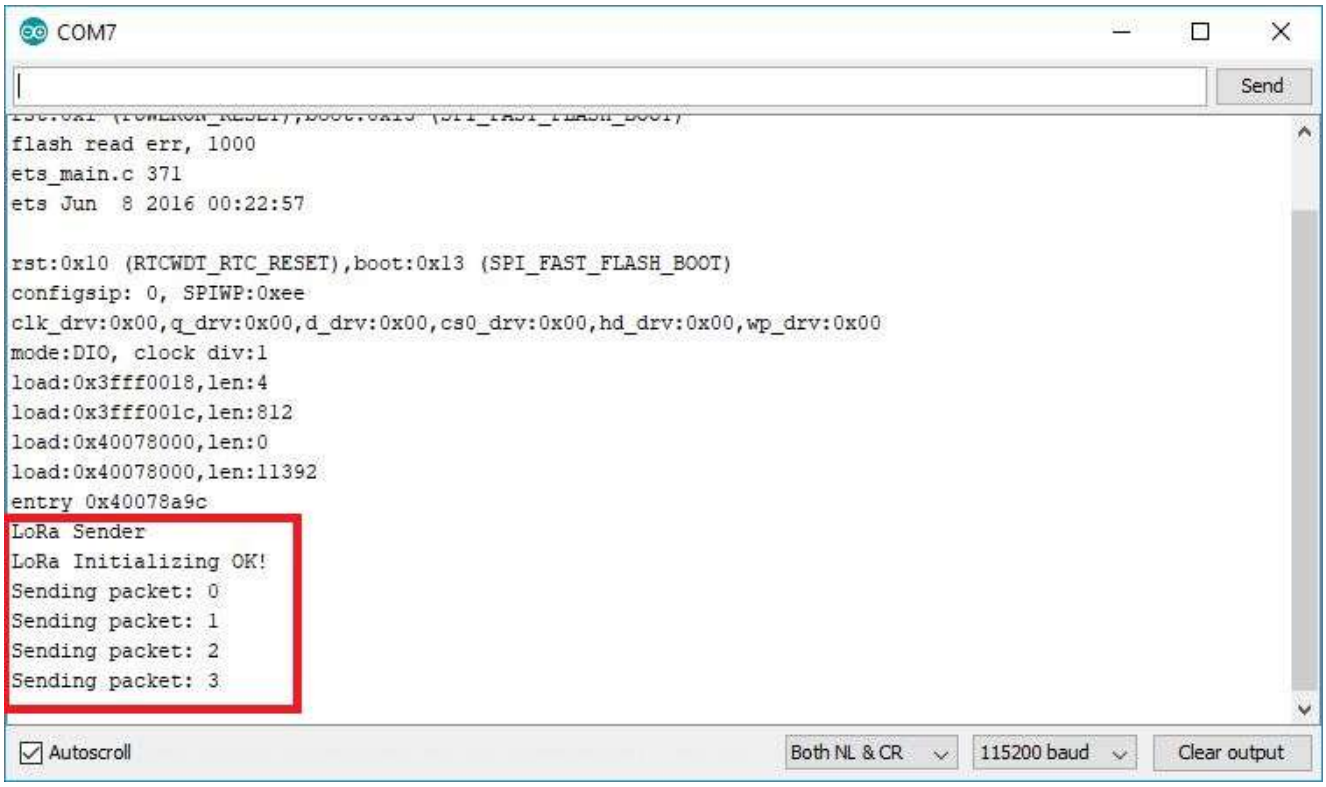
After this, the counter message is incremented by one in every loop, which happens every 10 seconds.

```
counter++;
delay(10000);
```

Testing the Sender Sketch

Upload the code to your ESP32 board. Make sure you have the right board and COM port selected.

After that, open the Serial Monitor, and press the ESP32 enable button. You should see a success message as shown in the figure below. The counter should be incremented every 10 seconds.



The LoRa Receiver Sketch

Now, grab another ESP32 and upload the following sketch (the LoRa receiver sketch). This sketch listens for LoRa packets with the sync word you've defined and prints the content of the packets on the Serial Monitor, as well as the RSSI. The RSSI measures the relative received signal strength.

```
/*********
  Modified from the examples of the Arduino LoRa library
  More resources: https://randomnerdtutorials.com
*********/

#include <SPI.h>
#include <LoRa.h>

//define the pins used by the transceiver module
#define ss 5
#define rst 14
#define dio0 2

void setup() {
  //initialize Serial Monitor
```

```
  Serial.begin(115200);
  while (!Serial);
  Serial.println("LoRa Receiver");

  //setup LoRa transceiver module
  LoRa.setPins(ss, rst, dio0);

  //replace the LoRa.begin(---E-) argument with your location's
  //433E6 for Asia
  //866E6 for Europe
  //915E6 for North America
```

View raw code

This sketch is very similar to the previous one. Only the `loop()` is different.

You might need to change the frequency and the sycnword to match the one used in the sender sketch.

In the `loop()` the code checks if a new packet has been received using the `parsePacket()` method.

```
int packetSize = LoRa.parsePacket();
```

If there's a new packet, we'll read its content while it is available.

To read the incoming data you use the `readString()` method.

```
while (LoRa.available()) {
  String LoRaData = LoRa.readString();
  Serial.print(LoRaData);
}
```

The incoming data is saved on the `LoRaData` variable and printed in the Serial Monitor.

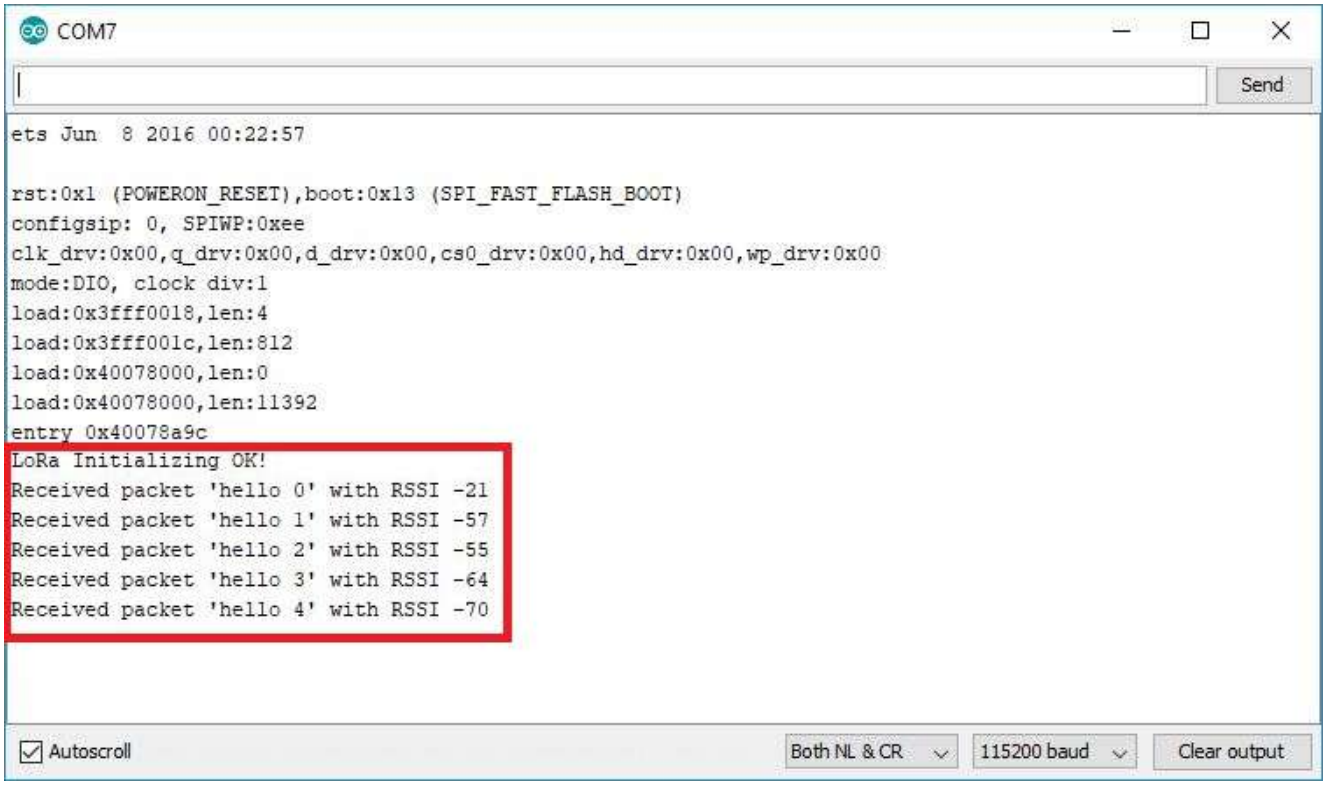
Finally, the next two lines of code print the RSSI of the received packet in dB.

```
  Serial.print("' with RSSI ");
  Serial.println(LoRa.packetRssi());
```

Testing the LoRa Receiver Sketch

Upload this code to your ESP32. At this point you should have two ESP32 boards with different sketches: the sender and the receiver.

Open the Serial Monitor for the LoRa Receiver, and press the LoRa Sender enable button. You should start getting the LoRa packets on the receiver.



Congratulations! You've built a LoRa Sender and a LoRa Receiver using the ESP32.

Taking It Further

Now, you should test the communication range between the Sender and the Receiver on your area. The communication range greatly varies depending on your environment (if you live in a rural or urban area with a lot of tall buildings). To test the communication range you can add an OLED display to the LoRa receiver

and go for a walk to see how far you can get a communication (this is a subject for a future tutorial).

In this example we're just sending an hello message, but the idea is to replace that text with useful information.

Wrapping Up

In summary, in this tutorial we've shown you the basics of LoRa technology:

- LoRa is a radio modulation technique;
- LoRa allows long-distance communication of small amounts of data and requires low power;
- You can use LoRa in point to point communication or in a network;
- LoRa can be especially useful if you want to monitor sensors that are not covered by your Wi-Fi network and that are several meters apart.

We've also shown you how to build a simple LoRa sender and LoRa receiver. These are just simple examples to get you started with LoRa. We'll be adding more projects about this subject soon, so stay tuned!

You may also like reading:

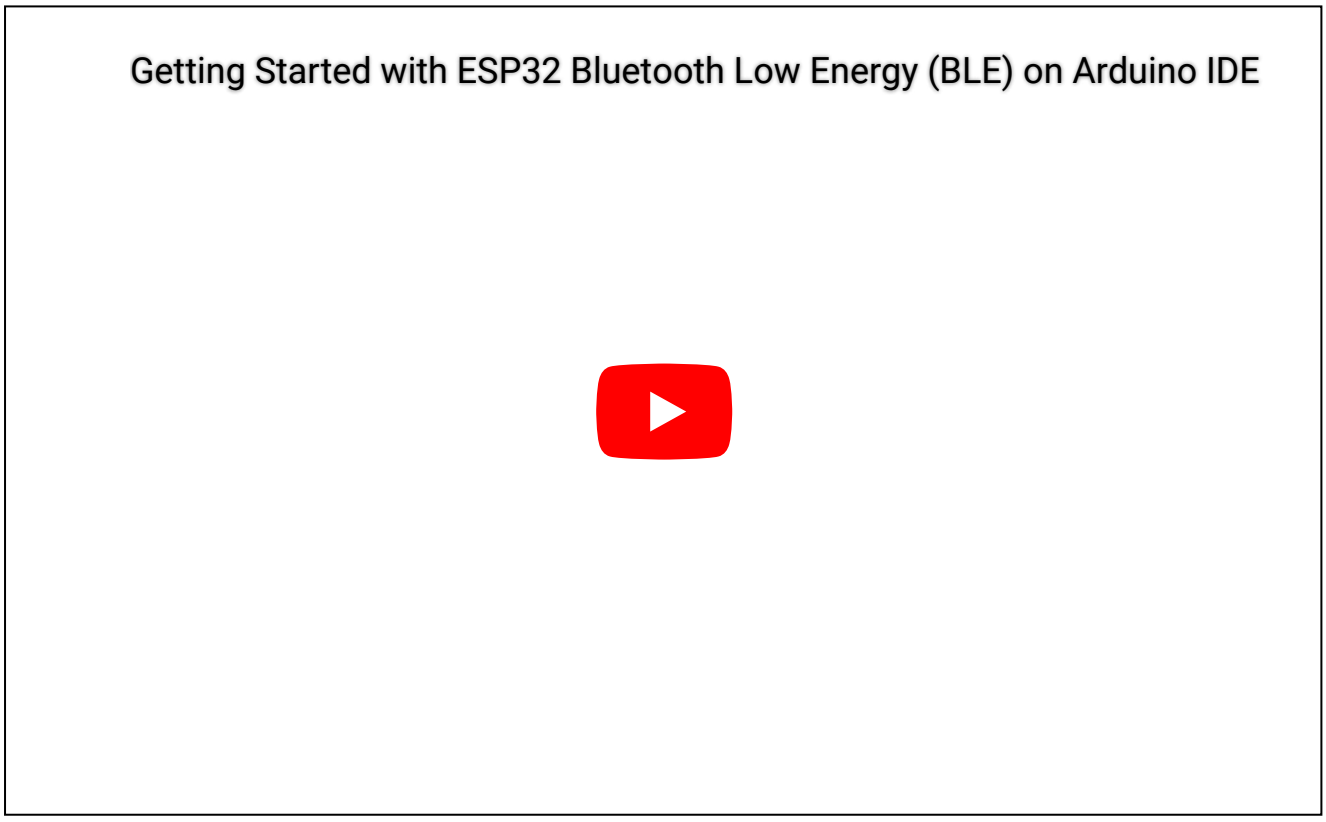
- [Review] TTGO LoRa32 SX1276 OLED: Pinout, Specifications, etc…

Getting Started with ESP32 Bluetooth Low Energy (BLE) on Arduino IDE

The ESP32 comes not only with Wi-Fi but also with Bluetooth and Bluetooth Low Energy (BLE). This post is a quick introduction to BLE with the ESP32. First, we'll explore what's BLE and what it can be used for, and then we'll take a look at some examples with the ESP32 using Arduino IDE. For a simple introduction we'll create an ESP32 BLE server, and an ESP32 BLE scanner to find that server.

Introducing Bluetooth Low Energy

For a quick introduction to BLE, you can watch the video below, or you can scroll down for a written explanation.

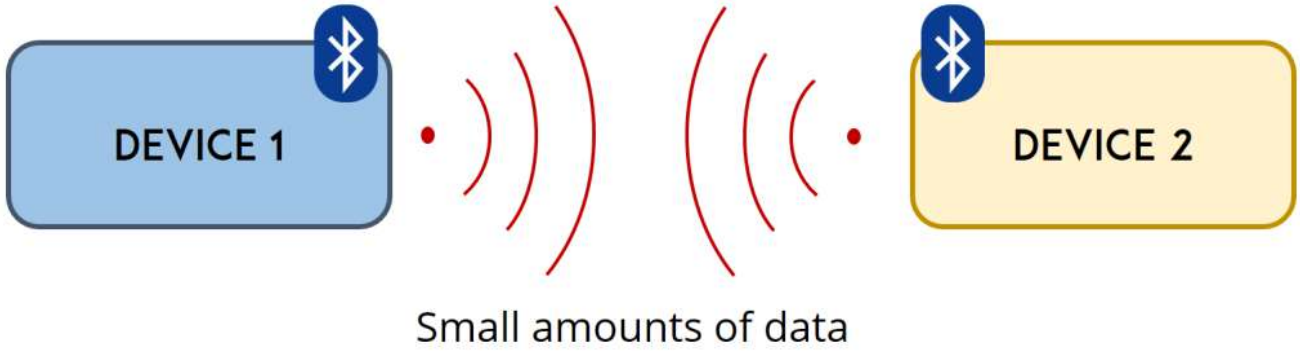


Recommended reading: learn how to use ESP32 Bluetooth Classic with Arduino IDE to exchange data between an ESP32 and an Android smartphone.

What is Bluetooth Low Energy?

Bluetooth Low Energy, BLE for short, is a power-conserving variant of Bluetooth. BLE's primary application is short distance transmission of small amounts of data (low bandwidth). Unlike Bluetooth that is always on, BLE remains in sleep mode constantly except for when a connection is initiated.

This makes it consume very low power. BLE consumes approximately 100x less power than Bluetooth (depending on the use case).



Additionally, BLE supports not only point-to-point communication, but also broadcast mode, and mesh network.

Take a look at the table below that compares BLE and Bluetooth Classic in more detail.

	Bluetooth Low Energy (LE)	Bluetooth Basic Rate/ Enhanced Data Rate (BR/EDR)
Optimized For...	Short burst data transmission	Continuous data streaming
Frequency Band	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)
Channels	40 channels with 2 MHz spacing (3 advertising channels/37 data channels)	79 channels with 1 MHz spacing
Channel Usage	Frequency-Hopping Spread Spectrum (FHSS)	Frequency-Hopping Spread Spectrum (FHSS)
Modulation	GFSK	GFSK, π/4 DQPSK, 8DPSK
Power Consumption	~0.01x to 0.5x of reference (depending on use case)	1 (reference value)
Data Rate	LE 2M PHY: 2 Mb/s LE 1M PHY: 1 Mb/s LE Coded PHY (S=2): 500 Kb/s LE Coded PHY (S=8): 125 Kb/s	EDR PHY (8DPSK): 3 Mb/s EDR PHY (π/4 DQPSK): 2 Mb/s BR PHY (GFSK): 1 Mb/s
Max Tx Power*	Class 1: 100 mW (+20 dBm) Class 1.5: 10 mW (+10 dbm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)	Class 1: 100 mW (+20 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)
Network Topologies	Point-to-Point (including piconet) Broadcast Mesh	Point-to-Point (including piconet)

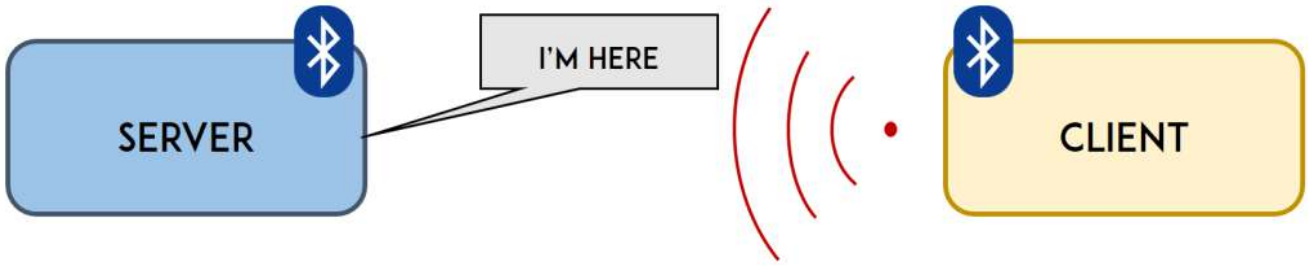
View Image Souce

Due to its properties, BLE is suitable for applications that need to exchange small amounts of data periodically running on a coin cell. For example, BLE is of great use in healthcare, fitness, tracking, beacons, security, and home automation industries.

BLE Server and Client

With Bluetooth Low Energy, there are two types of devices: the server and the client. The ESP32 can act either as a client or as a server.

The server advertises its existence, so it can be found by other devices, and contains the data that the client can read. The client scans the nearby devices, and when it finds the server it is looking for, it establishes a connection and listens for incoming data. This is called point-to-point communication.



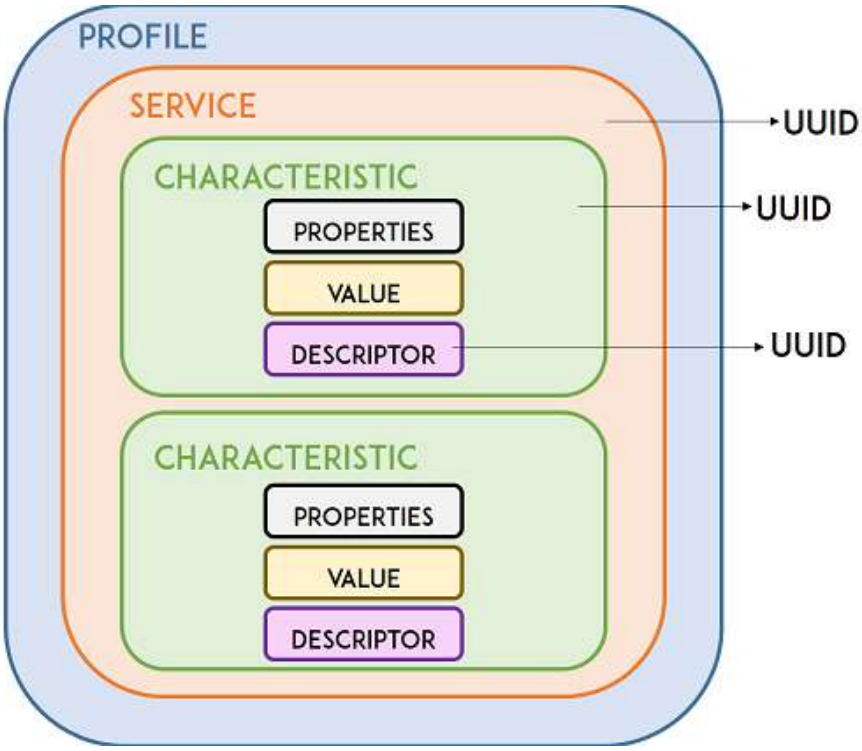
As mentioned previously, BLE also supports broadcast mode and mesh network:

- **Broadcast mode**: the server transmits data to many clients that are connected;
- **Mesh network**: all the devices are connected, this is a many to many connection.

Even though the broadcast and mesh network setups are possible to implement, they were developed very recently, so there aren't many examples implemented for the ESP32 at this moment.

GATT

GATT stands for Generic Attributes and it defines an hierarchical data structure that is exposed to connected BLE devices. This means that GATT defines the way that two BLE devices send and receive standard messages. Understanding this hierarchy is important, because it will make it easier to understand how to use the BLE and write your applications.



BLE Service

The top level of the hierarchy is a profile, which is composed of one or more services. Usually, a BLE device contains more than one service.

Every service contains at least one characteristic, or can also reference other services. A service is simply a collection of information, like sensor readings, for example.

There are predefined services for several types of data defined by the SIG (Bluetooth Special Interest Group) like: Battery Level, Blood Pressure, Heart Rate, Weight Scale, etc. You can check here other defined services.

Name	Uniform Type Identifier	Assigned Number	Specification
Generic Access	org.bluetooth.service.generic_access	0x1800	GSS
Alert Notification Service	org.bluetooth.service.alert_notification	0x1811	GSS
Automation IO	org.bluetooth.service.automation_io	0x1815	GSS
Battery Service	org.bluetooth.service.battery_service	0x180F	GSS
Blood Pressure	org.bluetooth.service.blood_pressure	0x1810	GSS
Body Composition	org.bluetooth.service.body_composition	0x181B	GSS
Bond Management Service	org.bluetooth.service.bond_management	0x181E	GSS
Continuous Glucose Monitoring	org.bluetooth.service.continuous_glucose_monitoring	0x181F	GSS
Current Time Service	org.bluetooth.service.current_time	0x1805	GSS
Cycling Power	org.bluetooth.service.cycling_power	0x1818	GSS
Cycling Speed and Cadence	org.bluetooth.service.cycling_speed_and_cadence	0x1816	GSS
Device Information	org.bluetooth.service.device_information	0x180A	GSS
Environmental Sensing	org.bluetooth.service.environmental_sensing	0x181A	GSS
Fitness Machine	org.bluetooth.service.fitness_machine	0x1826	GSS
Generic Attribute	org.bluetooth.service.generic_attribute	0x1801	GSS

View Image Souce

BLE Characteristic

The characteristic is always owned by a service, and it is where the actual data is contained in the hierarchy (value). The characteristic always has two attributes: characteristic declaration (that provides metadata about the data) and the characteristic value.

Additionally, the characteristic value can be followed by descriptors, which further expand on the metadata contained in the characteristic declaration.

The properties describe how the characteristic value can be interacted with. Basically, it contains the operations and procedures that can be used with the characteristic:

- Broadcast
- Read
- Write without response
- Write
- Notify

- Indicate
- Authenticated Signed Writes
- Extended Properties

UUID

Each service, characteristic and descriptor have an UUID (Universally Unique Identifier). An UUID is a unique 128-bit (16 bytes) number. For example:

```
55072829-bc9e-4c53-938a-74a6d4c78776
```

There are shortened UUIDs for all types, services, and profiles specified in the SIG (Bluetooth Special Interest Group).

But if your application needs its own UUID, you can generate it using this UUID generator website.

In summary, the UUID is used for uniquely identifying information. For example, it can identify a particular service provided by a Bluetooth device.

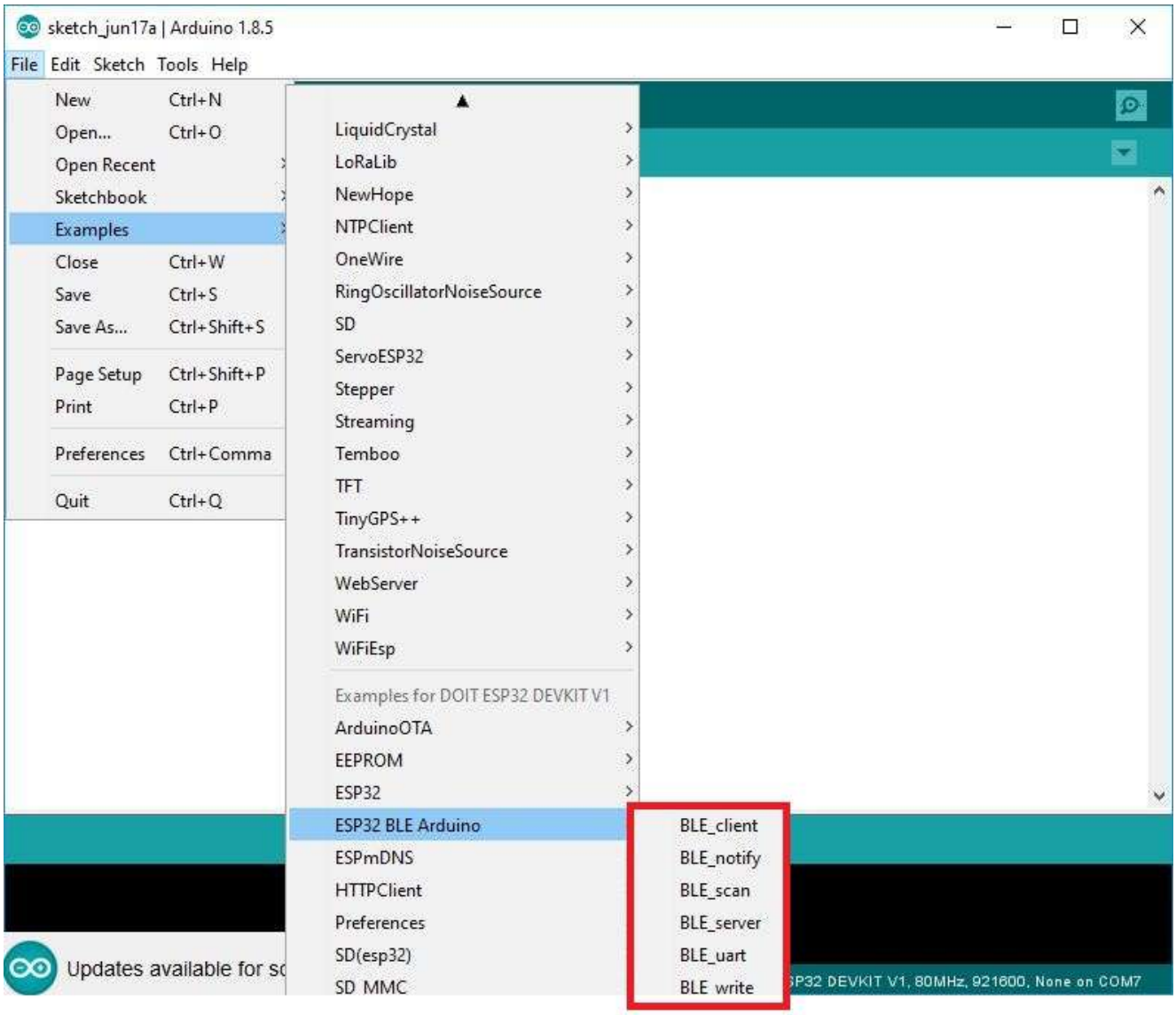
BLE with ESP32

The ESP32 can act as a BLE server or as a BLE client. There are several BLE examples for the ESP32 in the ESP32 BLE library for Arduino IDE. This library comes installed by default when you install the ESP32 on the Arduino IDE.

Note: You need to have the ESP32 add-on installed on the Arduino IDE. Follow one of the next tutorials to prepare your Arduino IDE to work with the ESP32, if you haven't already.

- **Windows** instructions – ESP32 Board in Arduino IDE
- **Mac and Linux** instructions – ESP32 Board in Arduino IDE

In your Arduino IDE, you can go to **File** > **Examples** > **ESP32 BLE Arduino** and explore the examples that come with the BLE library.



Note: to see the ESP32 examples, you must have the ESP32 board selected on **Tools** > **Board**.

For a brief introduction to the ESP32 with BLE on the Arduino IDE, we'll create an ESP32 BLE server, and then an ESP32 BLE scanner to find that server. We'll use and explain the examples that come with the BLE library.

To follow this example, you need two ESP32 development boards. We'll be using the ESP32 DOIT DEVKIT V1 Board.

ESP32 BLE Server

To create an ESP32 BLE Server, open your Arduino IDE and go to **File** > **Examples** > **ESP32 BLE Arduino** and select the **BLE_server** example. The following code should load:

```
/*
    Based on Neil Kolban example for IDF: https://github.com/nk
    Ported to Arduino ESP32 by Evandro Copercini
    updates by chegewara
*/

#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>

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

#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c33191
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26

void setup() {
  Serial.begin(115200);
  Serial.println("Starting BLE work!");

  BLEDevice::init("Long name works now");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  BLECharacteristic *pCharacteristic = pService->createCharacte
                                         CHARACTERISTIC_UUID,
                                         BLECharacteristic::PRO
                                         BLECharacteristic::PRO
```

View raw code

For creating a BLE server, the code should follow the next steps:

1. Create a BLE Server. In this case, the ESP32 acts as a BLE server.
2. Create a BLE Service.
3. Create a BLE Characteristic on the Service.
4. Create a BLE Descriptor on the Characteristic.
5. Start the Service.
6. Start advertising, so it can be found by other devices.

How the code works

Let's take a quick look at how the BLE server example code works.

It starts by importing the necessary libraries for the BLE capabilities.

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
```

Then, you need to define a UUID for the Service and Characteristic.

```
#define SERVICE_UUID "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8
```

You can leave the default UUIDs, or you can go to uuidgenerator.net to create random UUIDs for your services and characteristics.

In the `setup()`, it starts the serial communication at a baud rate of 115200.

```
Serial.begin(115200);
```

Then, you create a BLE device called "**MyESP32**". You can change this name to whatever you like.

```
// Create the BLE Device
BLEDevice::init("MyESP32");
```

In the following line, you set the BLE device as a server.

```
BLEServer *pServer = BLEDevice::createServer();
```

After that, you create a service for the BLE server with the UUID defined earlier.

```
BLEService *pService = pServer->createService(SERVICE_UUID);
```

Then, you set the characteristic for that service. As you can see, you also use the UUID defined earlier, and you need to pass as arguments the characteristic's properties. In this case, it's: READ and WRITE.

```
BLECharacteristic *pCharacteristic = pService->createCharacterist
                                     CHARACTERISTIC_UUID,
                                     BLECharacteristic::PROPERTY_
                                     BLECharacteristic::PROPERTY_
                                     );
```

After creating the characteristic, you can set its value with the `setValue()` method.

```
pCharacteristic->setValue("Hello World says Neil");
```

In this case we're setting the value to the text "Hello World says Neil". You can change this text to whatever your like. In future projects, this text can be a sensor reading, or the state of a lamp, for example.

Finally, you can start the service, and the advertising, so other BLE devices can scan and find this BLE device.

```
BLEAdvertising *pAdvertising = pServer->getAdvertising();
pAdvertising->start();
```

This is just a simple example on how to create a BLE server. In this code nothing is done in the `loop()`, but you can add what happens when a new client connects (check the BLE_notify example for some guidance).

ESP32 BLE Scanner

Creating an ESP32 BLE scanner is simple. Grab another ESP32 (while the other is running the BLE server sketch). In your Arduino IDE, go to **File > Examples > ESP32 BLE Arduino** and select the **BLE_scan** example. The following code should load.

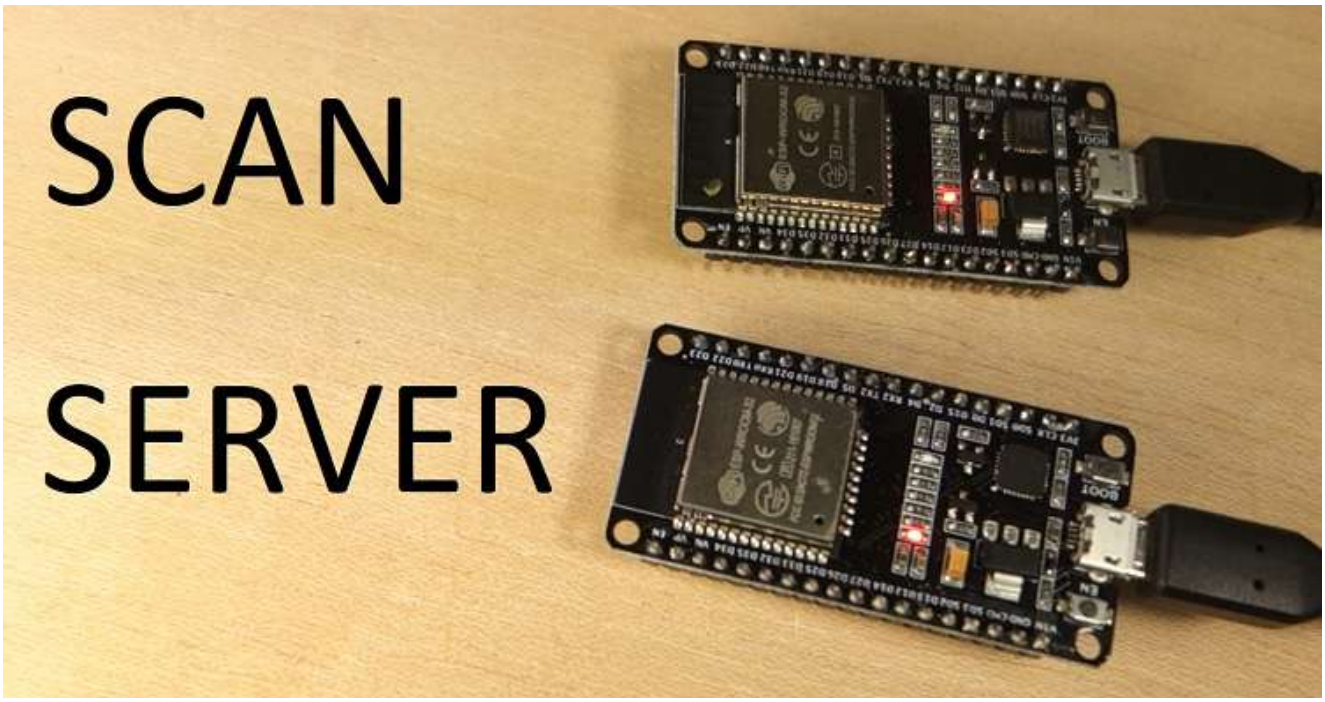
```
/*
   Based on Neil Kolban example for IDF: https://github.com/nko
   Ported to Arduino ESP32 by Evandro Copercini
*/

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

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

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

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

  BLEDevice::init("");
  pBLEScan = BLEDevice::getScan(); //create new scan
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDevice
  pBLEScan->setActiveScan(true); //active scan uses more power,
```

View raw code

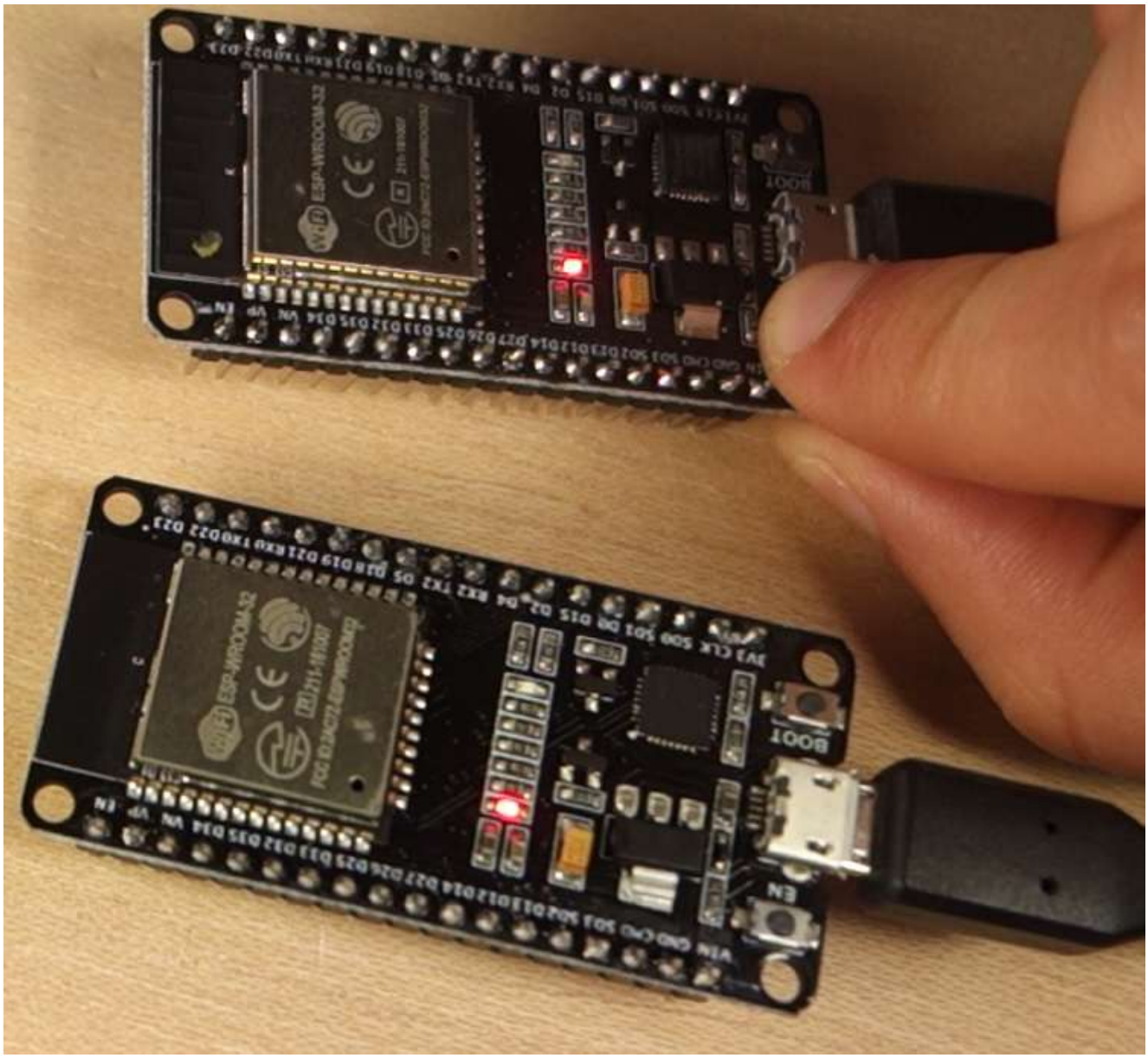
This code initializes the ESP32 as a BLE device and scans for nearby devices. Upload this code to your ESP32. You might want to temporarily disconnect the other ESP32 from your computer, so you're sure that you're uploading the code to the right ESP32 board.

Once the code is uploaded and you should have the two ESP32 boards powered on:

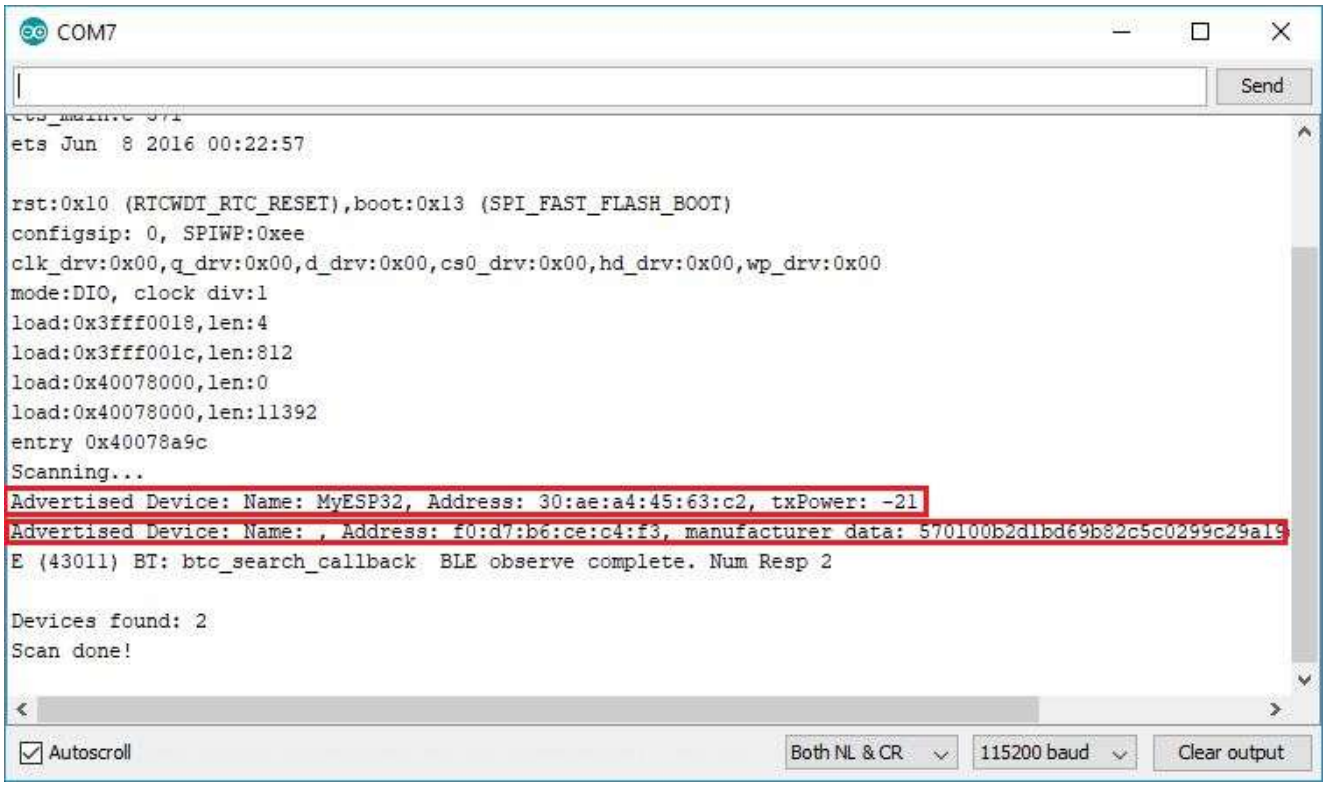
- One ESP32 with the "BLE_server" sketch;
- Other with ESP32 "BLE_scan" sketch.



Go to the Serial Monitor with the ESP32 running the "BLE_scan" example, press the ESP32 (with the "BLE_scan" sketch) ENABLE button to restart and wait a few seconds while it scans.



The scanner found two devices: one is the ESP32 (it has the name “**MyESP32**), and the other is our MiBand2.

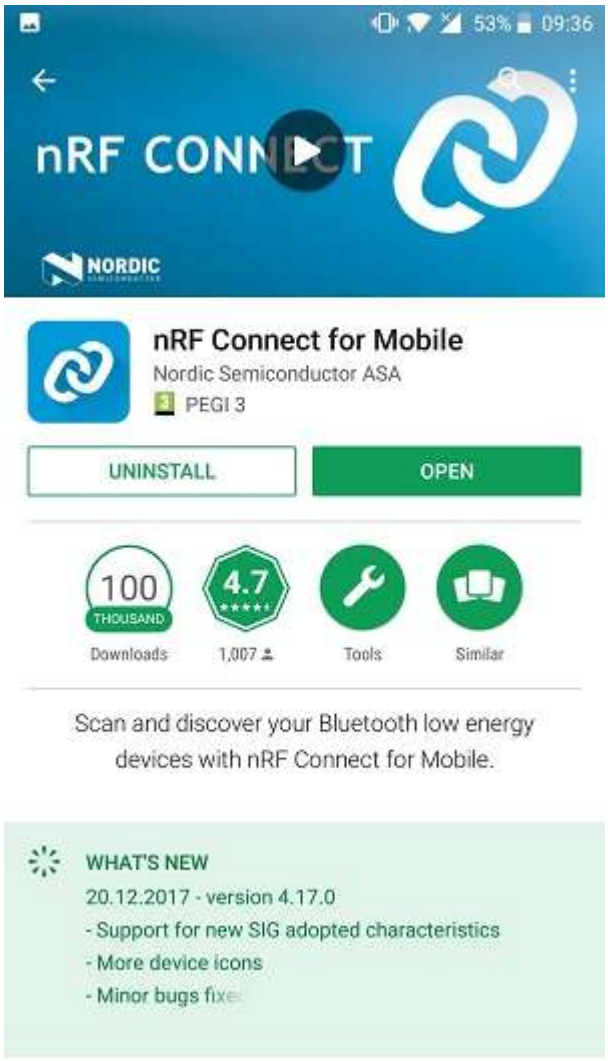


Testing the ESP32 BLE Server with Your Smartphone

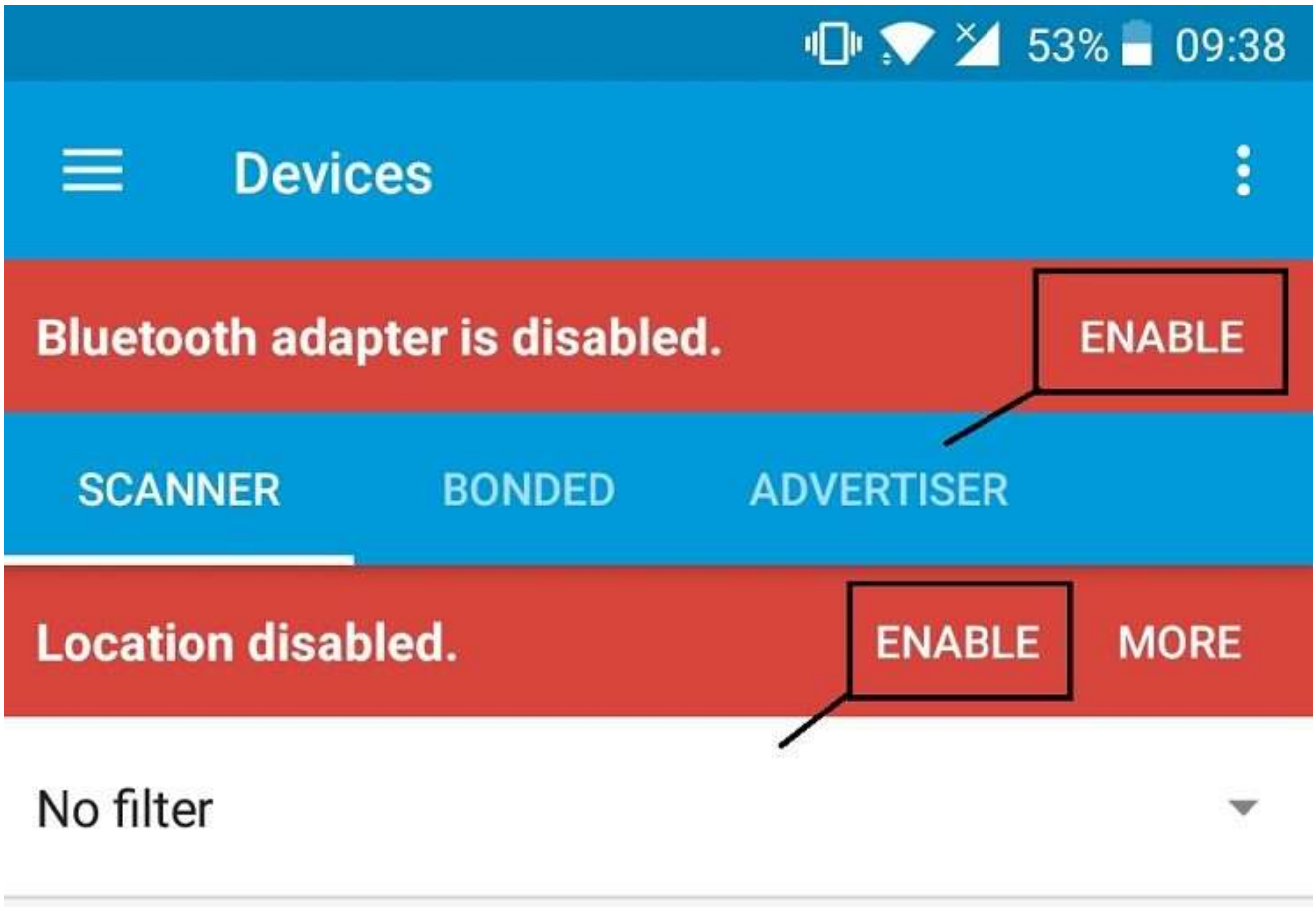
Most modern smartphones should have BLE capabilities. I'm currently using a OnePlus 5, but most smartphones should also work.

You can scan your ESP32 BLE server with your smartphone and see its services and characteristics. For that, we'll be using a free app called **nRF Connect for Mobile** from Nordic, it works on Android (Google Play Store) and iOS (App Store).

Go to Google Play Store or App Store and search for "nRF Connect for Mobile". Install the app and open it.



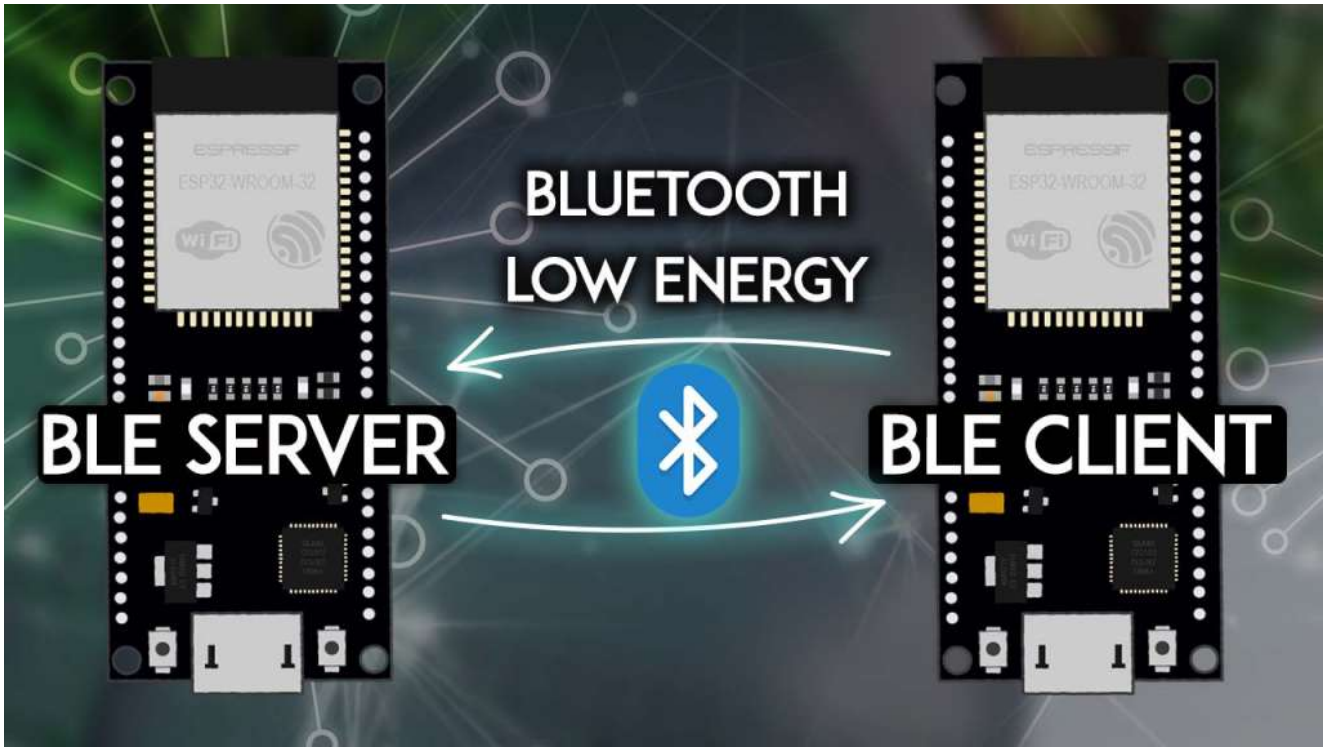
Don’t forget go to the Bluetooth settings and enable Bluetooth adapter in your smartphone. You may also want to make it visible to other devices to test other sketches later on.



Once everything is ready in your smartphone and the ESP32 is running the BLE server sketch, in the app, tap the scan button to scan for nearby devices. You should find an ESP32 with the name “**MyESP32**”.

ESP32 BLE Server and Client (Bluetooth Low Energy)

Learn how to make a BLE (Bluetooth Low Energy) connection between two ESP32 boards. One ESP32 is going to be the server, and the other ESP32 will be the client. The BLE server advertises characteristics that contain sensor readings that the client can read. The ESP32 BLE client reads the values of those characteristics (temperature and humidity) and displays them on an OLED display.



Recommended Reading: Getting Started with ESP32 Bluetooth Low Energy (BLE)

What is Bluetooth Low Energy?

Before going straight to the project, it is important to take a quick look at some essential BLE concepts so that you're able to better understand the project later on. If you're already familiar with BLE, you can skip to the Project Overview section.

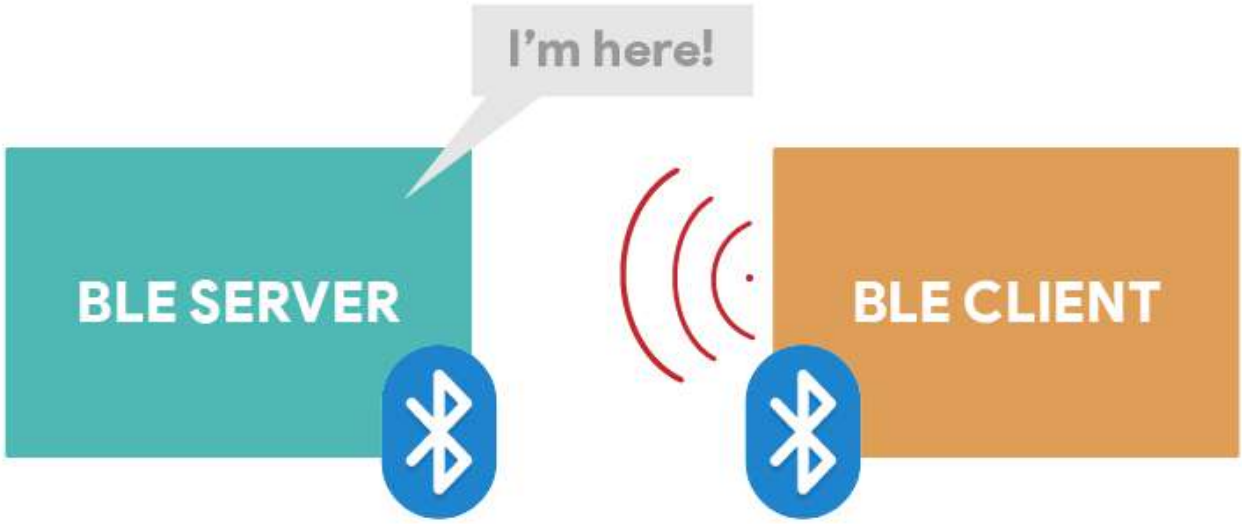
Bluetooth Low Energy, BLE for short, is a power-conserving variant of Bluetooth. BLE's primary application is short-distance transmission of small amounts of data (low bandwidth). Unlike Bluetooth that is always on, BLE remains in sleep mode constantly except for when a connection is initiated.

This makes it consume very low power. BLE consumes approximately 100x less power than Bluetooth (depending on the use case). You can check the main differences between Bluetooth and Bluetooth Low Energy here.

BLE Server and Client

With Bluetooth Low Energy, there are two types of devices: the server and the client. The ESP32 can act either as a client or as a server.

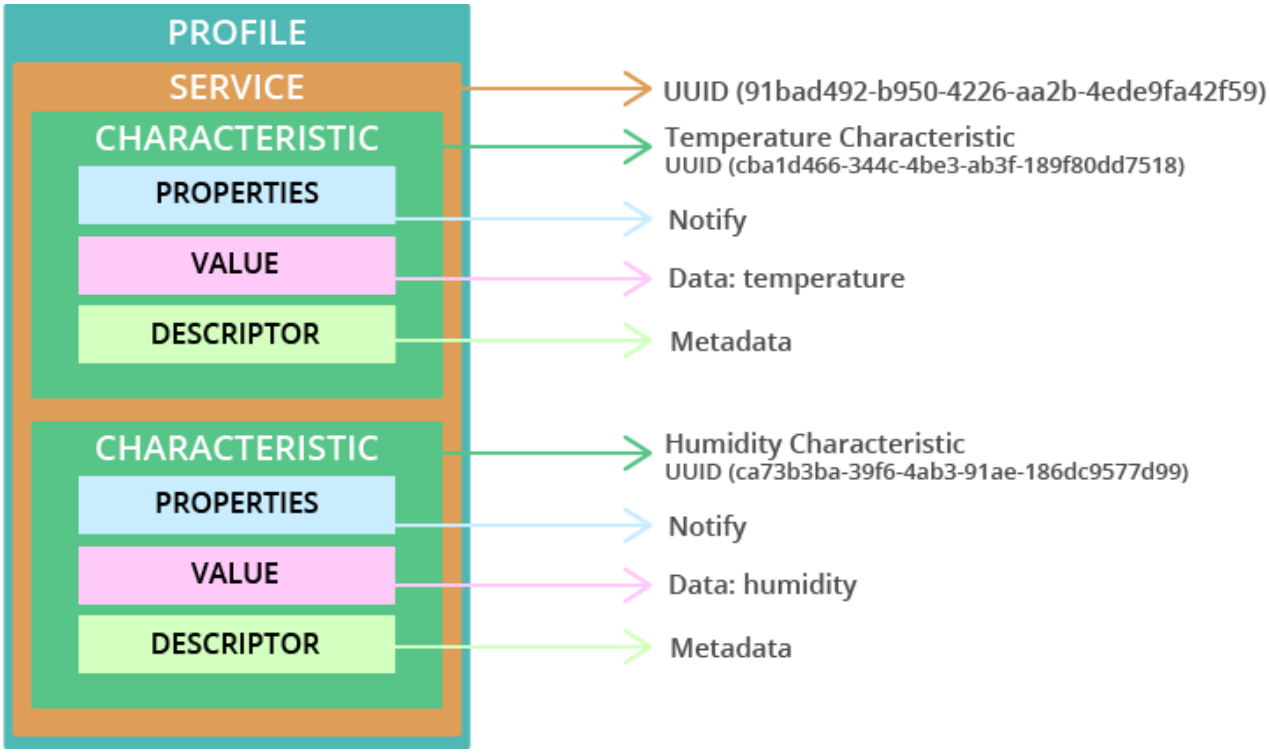
The server advertises its existence, so it can be found by other devices and contains data that the client can read. The client scans the nearby devices, and when it finds the server it is looking for, it establishes a connection and listens for incoming data. This is called point-to-point communication.



There are other possible communication modes like broadcast mode and mesh network (not covered in this tutorial).

GATT

GATT stands for Generic Attributes and it defines a hierarchical data structure that is exposed to connected BLE devices. This means that GATT defines the way that two BLE devices send and receive standard messages. Understanding this hierarchy is important because it will make it easier to understand how to use BLE with the ESP32.



- **Profile**: standard collection of services for a specific use case;
- **Service**: collection of related information, like sensor readings, battery level, heart rate, etc. ;
- **Characteristic**: it is where the actual data is saved on the hierarchy (**value**);
- **Descriptor**: metadata about the data;
- **Properties**: describe how the characteristic value can be interacted with. For example: read, write, notify, broadcast, indicate, etc.

In our example, we'll create a service with two *characteristics*. One for the temperature and another for the humidity. The actual temperature and humidity readings are saved on the *value* under their *characteristics*. Each characteristic has the *notify* property, so that it notifies the client whenever the values change.

UUID

Each service, characteristic, and descriptor have a UUID (Universally Unique Identifier). A UUID is a unique 128-bit (16 bytes) number. For example:

```
55072829-bc9e-4c53-938a-74a6d4c78776
```

There are shortened UUIDs for all types, services, and profiles specified in the SIG (Bluetooth Special Interest Group).

But if your application needs its own UUID, you can generate it using this UUID generator website.

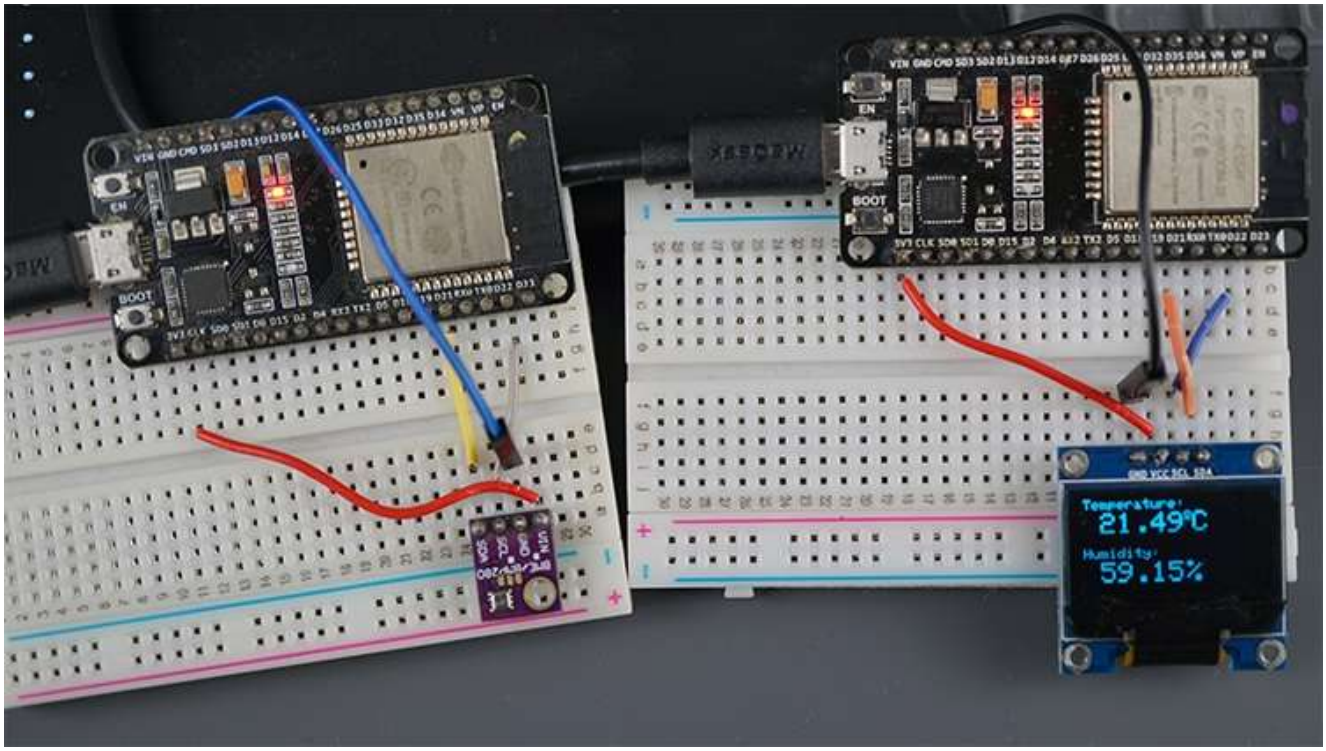
In summary, the UUID is used for uniquely identifying information. For example, it can identify a particular service provided by a Bluetooth device.

For a more detailed introduction about BLE, read our getting started guide:

- Getting Started with ESP32 Bluetooth Low Energy (BLE) on Arduino IDE

Project Overview

In this tutorial, you're going to learn how to make a BLE connection between two ESP32 boards. One ESP32 is going to be the BLE server, and the other ESP32 will be the BLE client.



The ESP32 BLE server is connected to a BME280 sensor and it updates its temperature and humidity characteristic values every 30 seconds.

The ESP32 client connects to the BLE server and it is notified of its temperature and humidity characteristic values. This ESP32 is connected to an OLED display and it prints the latest readings.

This project is divided into two parts:

- **Part 1** – ESP32 BLE server
- **Part 2** – ESP32 BLE client

Parts Required

Here's a list of the parts required to follow this project:

ESP32 BLE Server:

- ESP32 DOIT DEVKIT V1 Board (read Best ESP32 development boards)
- BME280 Sensor
- Jumper wires
- Breadboard
- Smartphone with Bluetooth (optional)

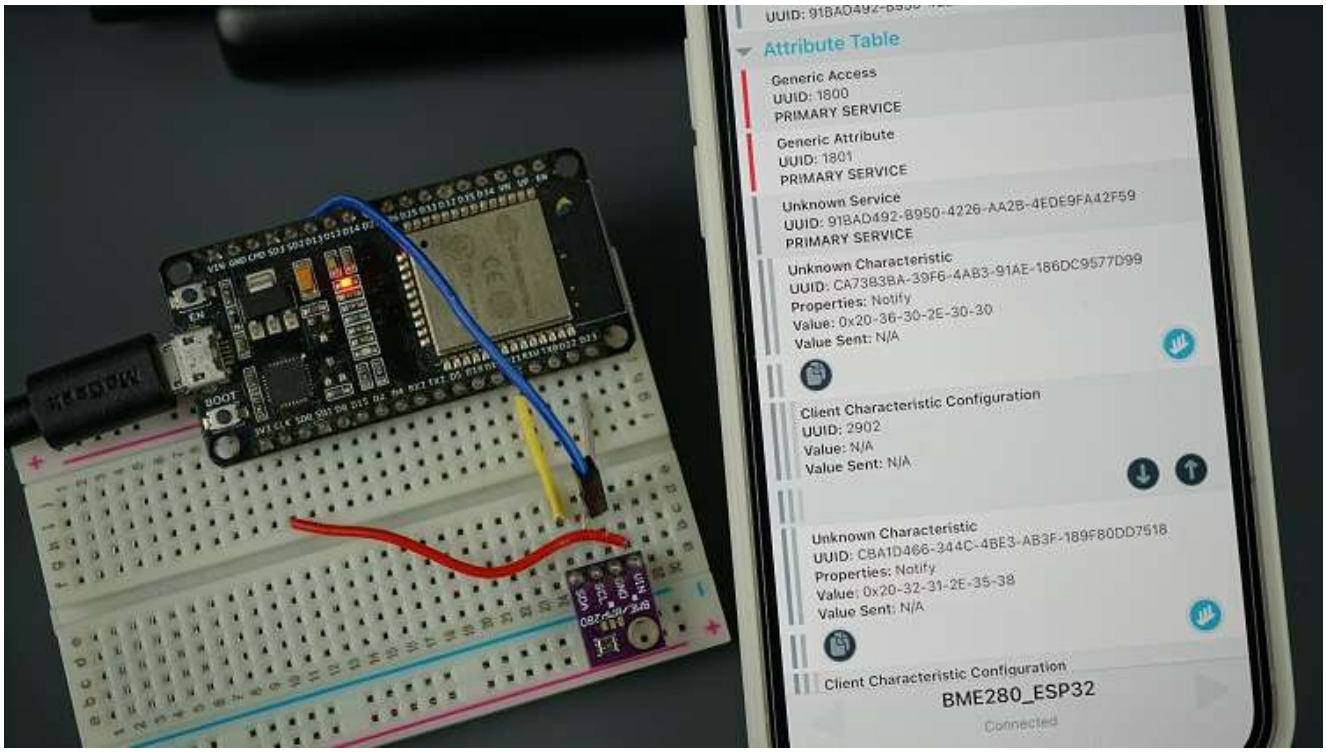
ESP32 BLE Client:

- ESP32 DOIT DEVKIT V1 Board (read Best ESP32 development boards)
- OLED display
- Jumper wires
- Breadboard

You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!

1) ESP32 BLE Server

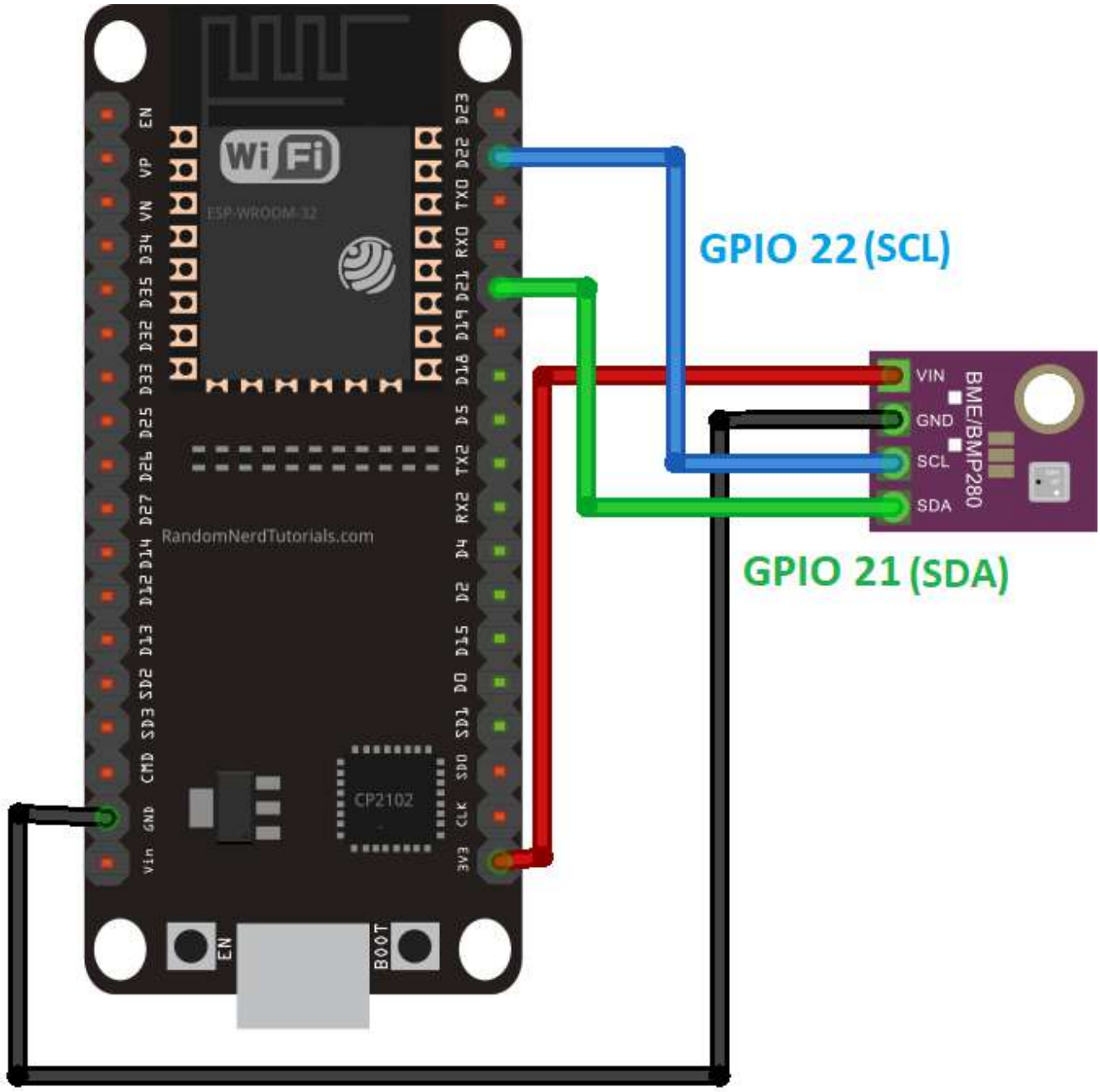
In this part, we'll set up the BLE Server that advertises a service that contains two characteristics: one for temperature and another for humidity. Those characteristics have the *Notify* property to notify new values to the client.



Schematic Diagram

The ESP32 BLE server will advertise characteristics with temperature and humidity from a BME280 sensor. You can use any other sensor as long as you add the required lines in the code.

We’re going to use I2C communication with the BME280 sensor module. For that, wire the sensor to the default ESP32 SCL (GPIO 22) and SDA (GPIO 21) pins, as shown in the following schematic diagram.



Recommended reading: ESP32 Pinout Reference: Which GPIO pins should you use?

Installing BME280 Libraries

As mentioned previously, we'll advertise sensor readings from a BME280 sensor. So, you need to install the libraries to interface with the BME280 sensor.

- Adafruit_BME280 library
- Adafruit_Sensor library

You can install the libraries using the Arduino Library Manager. Go to **Sketch** > **Include Library** > **Manage Libraries** and search for the library name.

Installing Libraries (VS Code + PlatformIO)

If you're using VS Code with the PlatformIO extension, copy the following to the `platformio.ini` file to include the libraries.

```
lib_deps = adafruit/Adafruit Unified Sensor @ ^1.1.4
           adafruit/Adafruit BME280 Library @ ^2.1.2
```

ESP32 BLE Server – Code

With the circuit ready and the required libraries installed, copy the following code to the Arduino IDE, or to the `main.cpp` file if you're using VS Code.

```
/*********
  Rui Santos
  Complete instructions at https://RandomNerdTutorials.com/esp3
  Permission is hereby granted, free of charge, to any person o
  The above copyright notice and this permission notice shall b
*********/

#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

//Default Temperature is in Celsius
//Comment the next line for Temperature in Fahrenheit
#define temperatureCelsius

//BLE server name
#define bleServerName "BME280_ESP32"

Adafruit_BME280 bme; // I2C
```

```
float temp;
float tempF;
```

View raw code

You can upload the code, and it will work straight away advertising its service with the temperature and humidity characteristics. Continue reading to learn how the code works, or skip to the Client section.

There are several examples showing how to use BLE with the ESP32 in the *Examples* section. In your Arduino IDE, go to **File** > **Examples** > **ESP32 BLE Arduino**. This server sketch is based on the *Notify* example.

Importing Libraries

The code starts by importing the required libraries.

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
```

Choosing Temperature Unit

By default, the ESP sends the temperature in Celsius degrees. You can comment the following line or delete it to send the temperature in Fahrenheit degrees.

```
//Comment the next line for Temperature in Fahrenheit
#define temperatureCelsius
```

BLE Server Name

The following line defines a name for our BLE server. Leave the default BLE server name. Otherwise, the server name in the client code also needs to be changed (because they have to match).

```
//BLE server name
#define bleServerName "BME280_ESP32"
```

BME280 Sensor

Create an `Adafruit_BME280` object called `bme` on the default ESP32 I2C pins.

```
Adafruit_BME280 bme; // I2C
```

The `temp`, `tempF` and `hum` variables will hold the temperature in Celsius degrees, the temperature in Fahrenheit degrees, and the humidity read from the BME280 sensor.

```
float temp;
float tempF;
float hum;
```

Other Variables

The following timer variables define how frequently we want to write to the temperature and humidity characteristic. We set the `timerDelay` variable to 30000 milliseconds (30 seconds), but you can change it.

```
// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;
```

The `deviceConnected` boolean variable allows us to keep track if a client is connected to the server.

```
bool deviceConnected = false;
```

BLE UUIDs

In the next lines, we define UUIDs for the service, for the temperature characteristic in celsius, for the temperature characteristic in Fahrenheit, and for the humidity.

```
// https://www.uuidgenerator.net/
#define SERVICE_UUID "91bad492-b950-4226-aa2b-4ede9fa42f59"

// Temperature Characteristic and Descriptor
#ifdef temperatureCelsius
  BLECharacteristic bmeTemperatureCelsiusCharacteristics("cba1d46
  BLEDescriptor bmeTemperatureCelsiusDescriptor(BLEUUID((uint16_t
#else
  BLECharacteristic bmeTemperatureFahrenheitCharacteristics("f78e
  BLEDescriptor bmeTemperatureFahrenheitDescriptor(BLEUUID((uint1
#endif

// Humidity Characteristic and Descriptor
BLECharacteristic bmeHumidityCharacteristics("ca73b3ba-39f6-4ab3-
BLEDescriptor bmeHumidityDescriptor(BLEUUID((uint16_t)0x2903));
```

I recommend leaving all the default UUIDs. Otherwise, you also need to change the code on the client side—so the client can find the service and retrieve the characteristic values.

setup()

In the `setup()`, initialize the Serial Monitor and the BME280 sensor.

```
// Start serial communication
Serial.begin(115200);
```

```
// Init BME Sensor
initBME();
```

Create a new BLE device with the BLE server name you've defined earlier:

```
// Create the BLE Device
BLEDevice::init(bleServerName);
```

Set the BLE device as a server and assign a callback function.

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

The callback function `MyServerCallbacks()` changes the boolean variable `deviceConnected` to `true` or `false` according to the current state of the BLE device. This means that if a client is connected to the server, the state is `true`. If the client disconnects, the boolean variable changes to `false`. Here's the part of the code that defines the `MyServerCallbacks()` function.

```
//Setup callbacks onConnect and onDisconnect
class MyServerCallbacks: public BLEServerCallbacks {
  void onConnect(BLEServer* pServer) {
    deviceConnected = true;
  };
  void onDisconnect(BLEServer* pServer) {
    deviceConnected = false;
  }
};
```

Start a BLE service with the service UUID defined earlier.

```
BLEService *bmeService = pServer->createService(SERVICE_UUID);
```

Then, create the temperature BLE characteristic. If you're using Celsius degrees it sets the following characteristic and descriptor:

```
#ifdef temperatureCelsius
  bmeService->addCharacteristic(&bmeTemperatureCelsiusCharacteris
  bmeTemperatureCelsiusDescriptor.setValue("BME temperature Celsi
  bmeTemperatureCelsiusCharacteristics.addDescriptor(new BLE2902(
```

Otherwise, it sets the Fahrenheit characteristic:

```
#else
  bmeService->addCharacteristic(&dhtTemperatureFahrenheitCharacte
  bmeTemperatureFahrenheitDescriptor.setValue("BME temperature Fa
  bmeTemperatureFahrenheitCharacteristics.addDescriptor(new BLE29
#endif
```

After that, it sets the humidity characteristic:

```
// Humidity
bmeService->addCharacteristic(&bmeHumidityCharacteristics);
bmeHumidityDescriptor.setValue("BME humidity");
bmeHumidityCharacteristics.addDescriptor(new BLE2902());
```

Finally, you start the service, and the server starts the advertising so other devices can find it.

```
// Start the service
bmeService->start();

// Start advertising
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
```

```
  pServer->getAdvertising()->start();
  Serial.println("Waiting a client connection to notify...");
```

loop()

The `loop()` function is fairly straightforward. You constantly check if the device is connected to a client or not. If it's connected, and the `timerDelay` has passed, it reads the current temperature and humidity.

```
  if (deviceConnected) {
    if ((millis() - lastTime) > timerDelay) {
      // Read temperature as Celsius (the default)
      temp = bme.readTemperature();
      // Fahrenheit
      tempF = temp*1.8 +32;
      // Read humidity
      hum = bme.readHumidity();
```

If you're using temperature in Celsius it runs the following code section. First, it converts the temperature to a char variable (`temperatureCTemp` variable). We must convert the temperature to a char variable type to use it in the `setValue()` function.

```
  static char temperatureCTemp[6];
  dtostrf(temp, 6, 2, temperatureCTemp);
```

Then, it sets the `bmeTemperatureCelsiusCharacteristic` value to the new temperature value (`temperatureCTemp`) using the `setValue()` function. After settings the new value, we can notify the connected client using the `notify()` function.

```
  //Set temperature Characteristic value and notify connected clien
  bmeTemperatureCelsiusCharacteristics.setValue(temperatureCTemp);
  bmeTemperatureCelsiusCharacteristics.notify();
```

We follow a similar procedure for the Temperature in Fahrenheit.

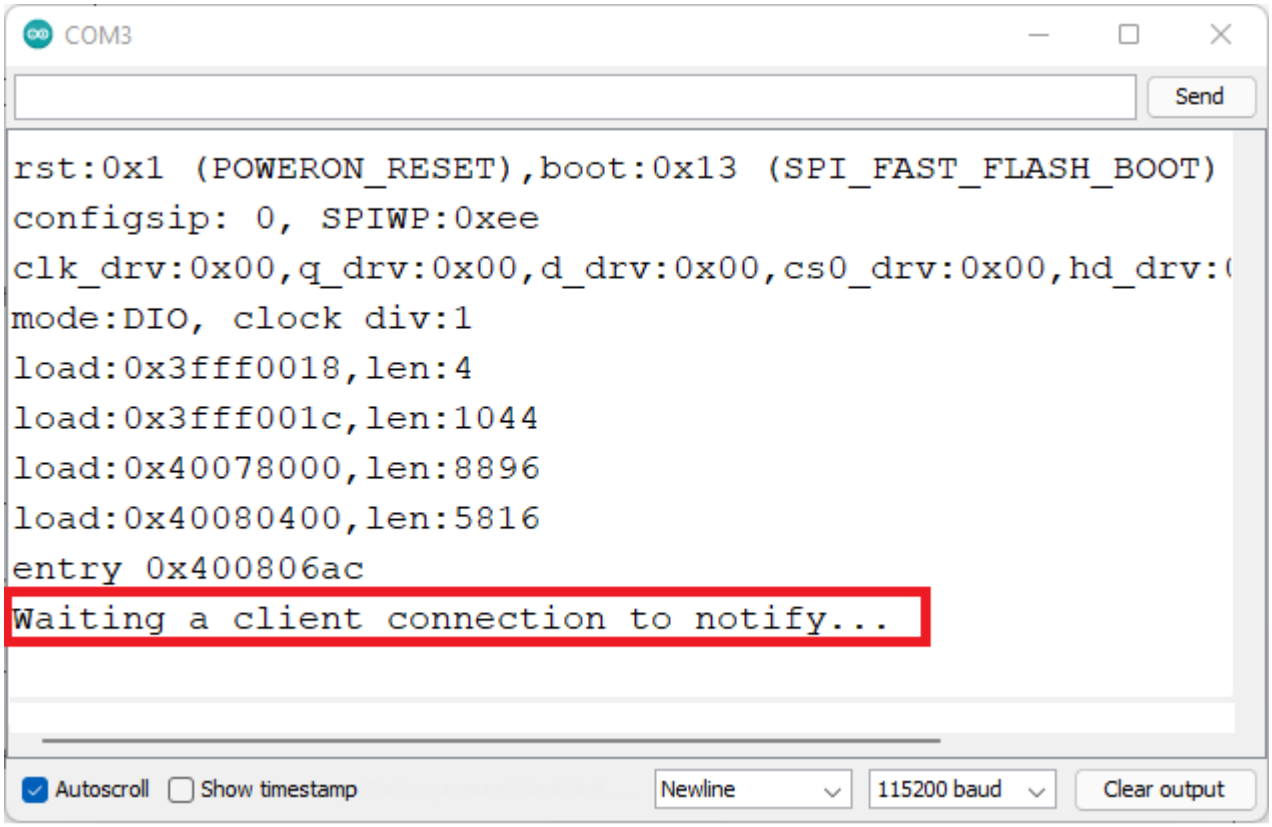
```
#else
    static char temperatureFTemp[6];
    dtostrf(f, 6, 2, temperatureFTemp);
    //Set temperature Characteristic value and notify connected c
    bmeTemperatureFahrenheitCharacteristics.setValue(tempF);
    bmeTemperatureFahrenheitCharacteristics.notify();
    Serial.print("Temperature Fahrenheit: ");
    Serial.print(tempF);
    Serial.print(" *F");
#endif
```

Sending the humidity also uses the same process.

```
//Notify humidity reading from DHT
static char humidityTemp[6];
dtostrf(hum, 6, 2, humidityTemp);
//Set humidity Characteristic value and notify connected client
bmeHumidityCharacteristics.setValue(humidityTemp);
bmeHumidityCharacteristics.notify();
Serial.print(" - Humidity: ");
Serial.print(hum);
Serial.println(" %");
```

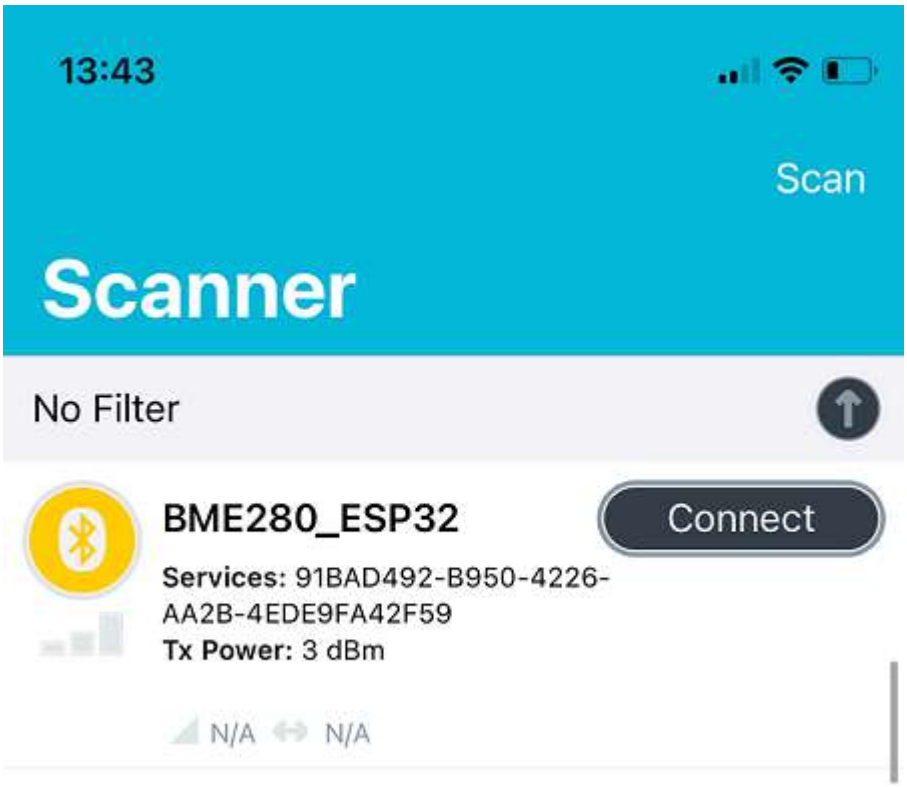
Testing the ESP32 BLE Server

Upload the code to your board and then, open the Serial Monitor. It will display a message as shown below.

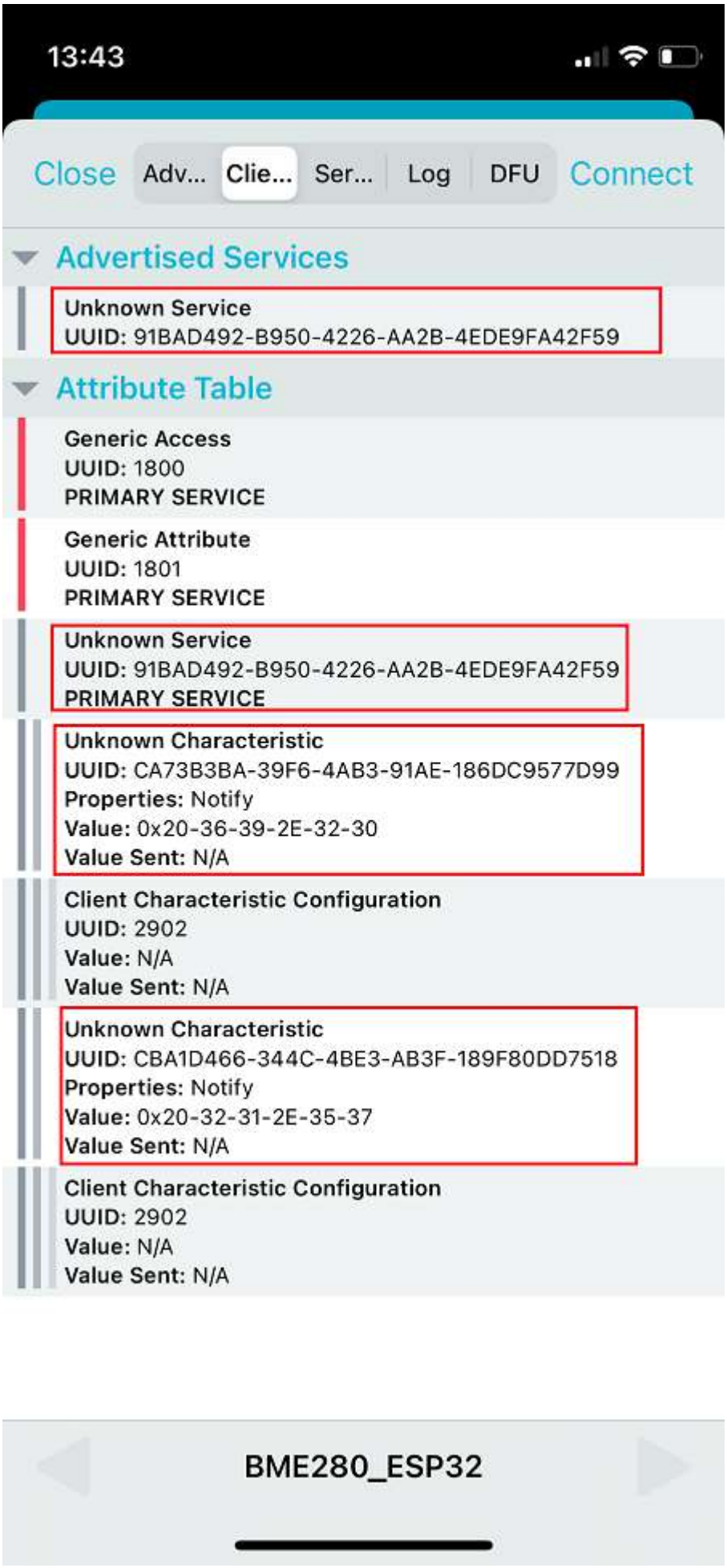


Then, you can test if the BLE server is working as expected by using a BLE scan application on your smartphone like **nRF Connect**. This application is available for Android and iOS.

After installing the application, enable Bluetooth on your smartphone. Open the nRF Connect app and click on the Scan button. It will find all Bluetooth nearby devices, including your **BME280_ESP32** device (it is the BLE server name you defined on the code).



Connect to your BME280_ESP32 device and then, select the client tab (the interface might be slightly different). You can check that it advertises the service with the UUID we defined in the code, as well as the temperature and humidity characteristics. Notice that those characteristics have the *Notify* property.



Your ESP32 BLE Server is ready!

Go to the next section to create an ESP32 client that connects to the server to get access to the temperature and humidity characteristics and get the readings to display them on an OLED display.

---

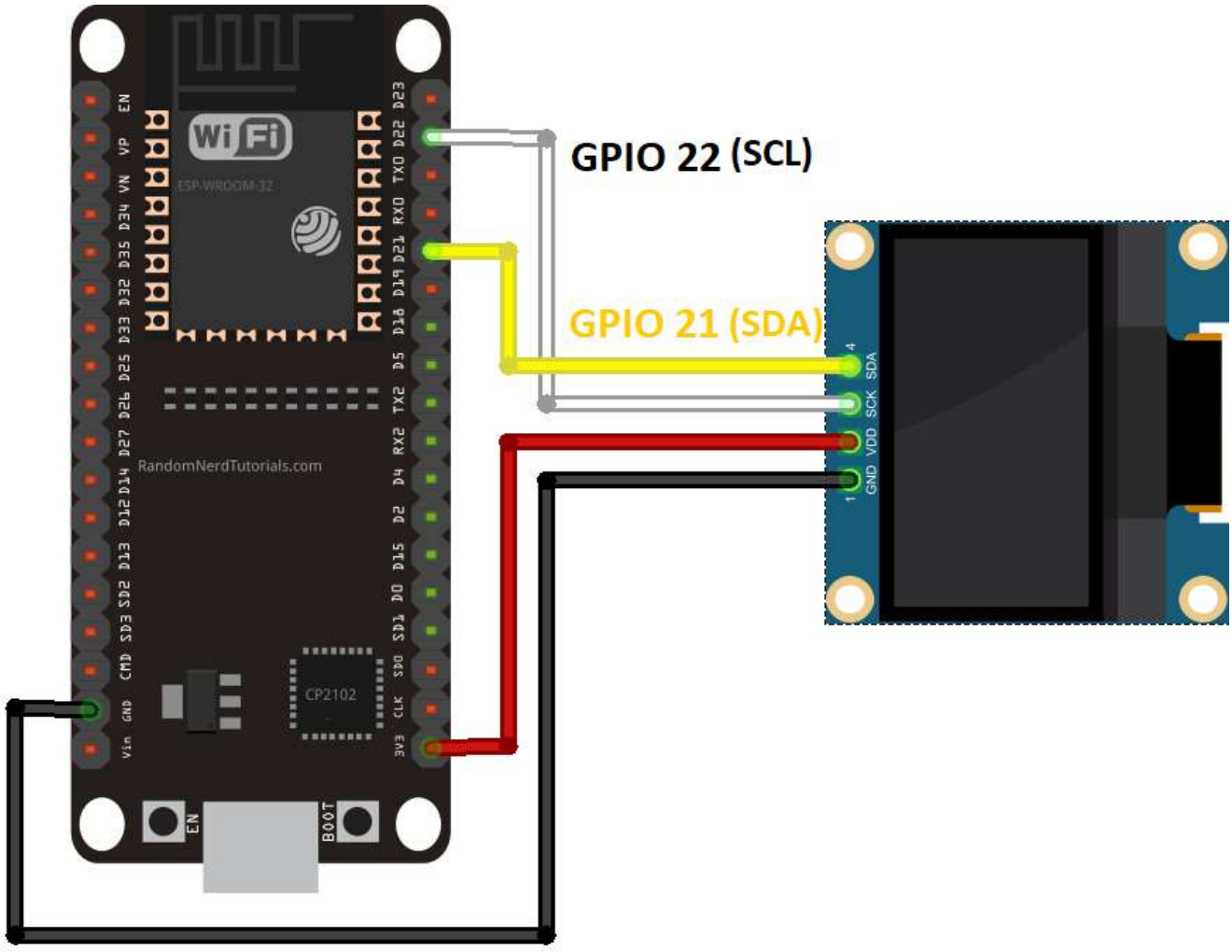
2) ESP32 BLE Client

In this section, we'll create the ESP32 BLE client that will establish a connection with the ESP32 BLE server, and display the readings on an OLED display.

Schematic

The ESP32 BLE client is connected to an OLED display. The display shows the readings received via Bluetooth.

Wire your OLED display to the ESP32 by following the next schematic diagram. The SCL pin connects to GPIO 22 and the SDA pin to GPIO 21 .



Installing the SSD1306, GFX and BusIO Libraries

You need to install the following libraries to interface with the OLED display:

- Adafruit_SSD1306 library
- Adafruit GFX library
- Adafruit BusIO library

To install the libraries, go **Sketch**> **Include Library** > **Manage Libraries**, and search for the libraries' names.

Installing Libraries (VS Code + PlatformIO)

If you're using VS Code with the PlatformIO extension, copy the following to the `platformio.ini` file to include the libraries.

```
lib_deps =
    adafruit/Adafruit GFX Library@^1.10.12
    adafruit/Adafruit SSD1306@^2.4.6
```

ESP32 BLE Client – Code

Copy the BLE client Sketch to your Arduino IDE or to the `main.cpp` file if you're using VS Code with PlatformIO.

```
/*********
  Rui Santos
  Complete instructions at https://RandomNerdTutorials.com/esp3
  Permission is hereby granted, free of charge, to any person o
  The above copyright notice and this permission notice shall b
*********/

#include "BLEDevice.h"
#include <Wire.h>
#include <Adafruit_SSD1306.h>
#include <Adafruit_GFX.h>

//Default Temperature is in Celsius
```

```
//Comment the next line for Temperature in Fahrenheit
#define temperatureCelsius

//BLE Server name (the other ESP32 name running the server sket
#define bleServerName "BME280_ESP32"

/* UUID's of the service, characteristic that we want to read*/
// BLE Service
static BLEUUID bmeServiceUUID("91bad492-b950-4226-aa2b-4ede9fa4

// BLE Characteristics
#ifdef temperatureCelsius
  //Temperature Celsius Characteristic
  static BLEUUID temperatureCharacteristicUUID("cba1d466-344c-4
```

View raw code

Continue reading to learn how the code works or skip to the Demonstration section.

Importing libraries

You start by importing the required libraries:

```
#include "BLEDevice.h"
#include <Wire.h>
#include <Adafruit_SSD1306.h>
#include <Adafruit_GFX.h>
```

Choosing temperature unit

By default the client will receive the temperature in Celsius degrees, if you comment the following line or delete it, it will start receiving the temperature in Fahrenheit degrees.

```
//Default Temperature is in Celsius
//Comment the next line for Temperature in Fahrenheit
```

```
#define temperatureCelsius
```

BLE Server Name and UUIDs

Then, define the BLE server name that we want to connect to and the service and characteristic UUIDs that we want to read. Leave the default BLE server name and UUIDs to match the ones defined in the server sketch.

```
//BLE Server name (the other ESP32 name running the server sketch
#define bleServerName "BME280_ESP32"

/* UUID's of the service, characteristic that we want to read*/
// BLE Service
static BLEUUID bmeServiceUUID("91bad492-b950-4226-aa2b-4ede9fa42f

// BLE Characteristics
#ifdef temperatureCelsius
  //Temperature Celsius Characteristic
  static BLEUUID temperatureCharacteristicUUID("cba1d466-344c-4be
#else
  //Temperature Fahrenheit Characteristic
  static BLEUUID temperatureCharacteristicUUID("f78ebbff-c8b7-410
#endif

// Humidity Characteristic
static BLEUUID humidityCharacteristicUUID("ca73b3ba-39f6-4ab3-91a
```

Declaring variables

Then, you need to declare some variables that will be used later with Bluetooth to check whether we're connected to the server or not.

```
//Flags stating if should begin connecting and if the connection
static boolean doConnect = false;
```

```
static boolean connected = false;
```

Create a variable of type `BLEAddress` that refers to the address of the server we want to connect. This address will be found during scanning.

```
//Address of the peripheral device. Address will be found during
static BLEAddress *pServerAddress;
```

Set the characteristics we want to read (temperature and humidity).

```
//Characteristicd that we want to read
static BLERemoteCharacteristic* temperatureCharacteristic;
static BLERemoteCharacteristic* humidityCharacteristic;
```

OLED Display

You also need to declare some variables to work with the OLED. Define the OLED width and height:

```
#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels
```

Instantiate the OLED display with the width and height defined earlier.

```
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire);
```

Temperature and Humidity Variables

Define char variables to hold the temperature and humidity values received by the server.

```
//Variables to store temperature and humidity
char* temperatureChar;
char* humidityChar;
```

The following variables are used to check whether new temperature and humidity readings are available and if it is time to update the OLED display.

```
//Flags to check whether new temperature and humidity readings ar
boolean newTemperature = false;
boolean newHumidity = false;
```

printReadings()

We created a function called `printReadings()` that displays the temperature and humidity readings on the OLED display.

```
void printReadings(){

  display.clearDisplay();
  // display temperature
  display.setTextSize(1);
  display.setCursor(0,0);
  display.print("Temperature: ");
  display.setTextSize(2);
  display.setCursor(0,10);
  display.print(temperatureChar);
  display.print(" ");
  display.setTextSize(1);
  display.cp437(true);
  display.write(167);
  display.setTextSize(2);
  Serial.print("Temperature:");
  Serial.print(temperatureChar);
  #ifdef temperatureCelsius
    //Temperature Celsius
```

```
    display.print("C");
    Serial.print("C");
  #else
    //Temperature Fahrenheit
    display.print("F");
    Serial.print("F");
  #endif

  //display humidity
  display.setTextSize(1);
  display.setCursor(0, 35);
  display.print("Humidity: ");
  display.setTextSize(2);
  display.setCursor(0, 45);
  display.print(humidityChar);
  display.print("%");
  display.display();
  Serial.print(" Humidity:");
  Serial.print(humidityChar);
  Serial.println("%");
}
```

Recommended reading: ESP32 OLED Display with Arduino IDE

setup()

In the `setup()`, start the OLED display.

```
//OLED display setup
// SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V inte
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C
  Serial.println(F("SSD1306 allocation failed"));
  for(;;); // Don't proceed, loop forever
}
```

Then, print a message in the first line saying “BME SENSOR”.

```
  display.clearDisplay();
  display.setTextSize(2);
  display.setTextColor(WHITE,0);
  display.setCursor(0,25);
  display.print("BLE Client");
  display.display();
```

Start the serial communication at a baud rate of 115200.

```
  Serial.begin(115200);
```

And initialize the BLE device.

```
  //Init BLE device
  BLEDevice::init("");
```

Scan nearby devices

The following methods scan for nearby devices.

```
  // Retrieve a Scanner and set the callback we want to use to be i
  // have detected a new device.  Specify that we want active scann
  // scan to run for 30 seconds.
  BLEScan* pBLEScan = BLEDevice::getScan();
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCall
  pBLEScan->setActiveScan(true);
  pBLEScan->start(30);
```

MyAdvertisedDeviceCallbacks() function

Note that the `MyAdvertisedDeviceCallbacks()` function, upon finding a BLE device, checks if the device found has the right BLE server name. If it has, it stops the scan and changes the `doConnect` boolean variable to `true`. This way we

know that we found the server we're looking for, and we can start establishing a connection.

```
//Callback function that gets called, when another device's adver
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCall
  void onResult(BLEAdvertisedDevice advertisedDevice) {
    if (advertisedDevice.getName() == bleServerName) { //Check if
      advertisedDevice.getScan()->stop(); //Scan can be stopped,
      pServerAddress = new BLEAddress(advertisedDevice.getAddress
      doConnect = true; //Set indicator, stating that we are read
      Serial.println("Device found. Connecting!");
    }
  }
};
```

Connect to the server

If the `doConnect` variable is `true`, it tries to connect to the BLE server. The `connectToServer()` function handles the connection between the client and the server.

```
//Connect to the BLE Server that has the name, Service, and Chara
bool connectToServer(BLEAddress pAddress) {
   BLEClient* pClient = BLEDevice::createClient();

  // Connect to the remove BLE Server.
  pClient->connect(pAddress);
  Serial.println(" - Connected to server");

  // Obtain a reference to the service we are after in the remote
  BLERemoteService* pRemoteService = pClient->getService(bmeServi
  if (pRemoteService == nullptr) {
    Serial.print("Failed to find our service UUID: ");
    Serial.println(bmeServiceUUID.toString().c_str());
    return (false);
  }
```

```
  // Obtain a reference to the characteristics in the service of
  temperatureCharacteristic = pRemoteService->getCharacteristic(t
  humidityCharacteristic = pRemoteService->getCharacteristic(humi

  if (temperatureCharacteristic == nullptr || humidityCharacteris
    Serial.print("Failed to find our characteristic UUID");
    return false;
  }
  Serial.println(" - Found our characteristics");

  //Assign callback functions for the Characteristics
  temperatureCharacteristic->registerForNotify(temperatureNotifyC
  humidityCharacteristic->registerForNotify(humidityNotifyCallbac
  return true;
}
```

It also assigns a callback function responsible to handle what happens when a new value is received.

```
//Assign callback functions for the Characteristics
temperatureCharacteristic->registerForNotify(temperatureNotifyCal
humidityCharacteristic->registerForNotify(humidityNotifyCallback)
```

After the BLE client is connected to the server, you need to active the *notify* property for each characteristic. For that, use the `writeValue()` method on the descriptor.

```
if (connectToServer(*pServerAddress)) {
  Serial.println("We are now connected to the BLE Server.");
  //Activate the Notify property of each Characteristic
  temperatureCharacteristic->getDescriptor(BLEUUID((uint16_t)0x29
  humidityCharacteristic->getDescriptor(BLEUUID((uint16_t)0x2902)
```

Notify new values

When the client receives a new notify value, it will call these two functions: `temperatureNotifyCallback()` and `humidityNotifyCallback()` that are responsible for retrieving the new value, update the OLED with the new readings and print them on the Serial Monitor.

```
//When the BLE Server sends a new temperature reading with the no
static void temperatureNotifyCallback(BLERemoteCharacteristic* pB
                                      uint8_t* pData, size_t le
  //store temperature value
  temperatureChar = (char*)pData;
  newTemperature = true;
}
```

```
//When the BLE Server sends a new humidity reading with the notif
static void humidityNotifyCallback(BLERemoteCharacteristic* pBLER
                                   uint8_t* pData, size_t length
  //store humidity value
  humidityChar = (char*)pData;
  newHumidity = true;
  Serial.print(newHumidity);
}
```

These two previous functions are executed every time the BLE server notifies the client with a new value, which happens every 30 seconds. These functions save the values received on the `temperatureChar` and `humidityChar` variables. These also change the `newTemperature` and `newHumidity` variables to `true`, so that we know we've received new readings.

Display new temperature and humidity readings

In the `loop()`, there is an if statement that checks if new readings are available. If there are new readings, we se the `newTemperature` and `newHumidity` variables

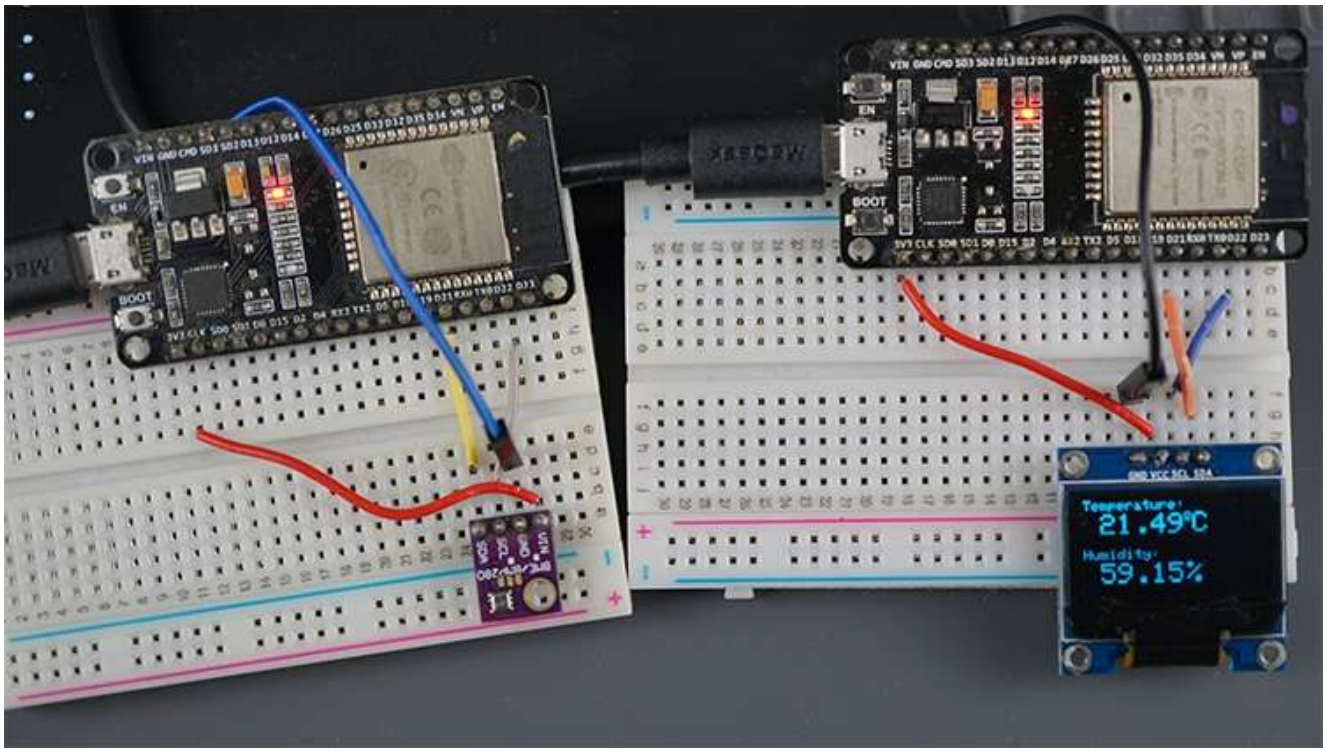
to `false`, so that we are able to receive new readings later on. Then, we call the `printReadings()` function to display the readings on the OLED.

```
//if new temperature readings are available, print in the OLED
if (newTemperature && newHumidity){
  newTemperature = false;
  newHumidity = false;
  printReadings();
}
```

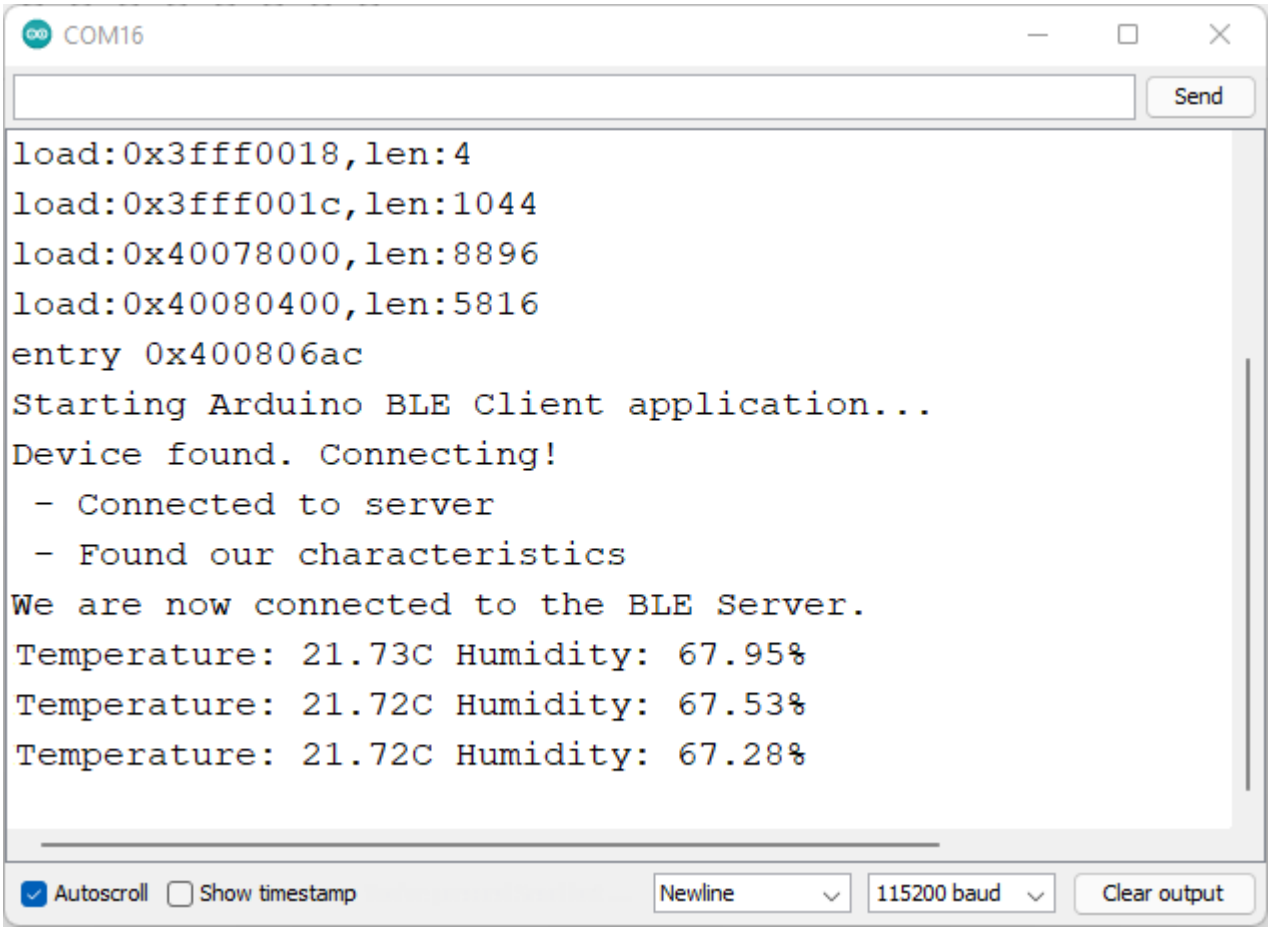
Testing the Project

That's it for the code. You can upload it to your ESP32 board.

Once the code is uploaded. Power the ESP32 BLE server, then power the ESP32 with the client sketch. The client starts scanning nearby devices, and when it finds the other ESP32, it establishes a Bluetooth connection. Every 30 seconds, it updates the display with the latest readings.



Important: don't forget to disconnect your smartphone from the BLE server. Otherwise, the ESP32 BLE Client won't be able to connect to the server.



Wrapping Up

In this tutorial, you learned how to create a BLE Server and a BLE Client with the ESP32. You learned how to set new temperature and humidity values on the BLE server characteristics. Then, other BLE devices (clients) can connect to that server and read those characteristic values to get the latest temperature and humidity values. Those characteristics have the *notify* property, so that the client is notified whenever there's a new value.

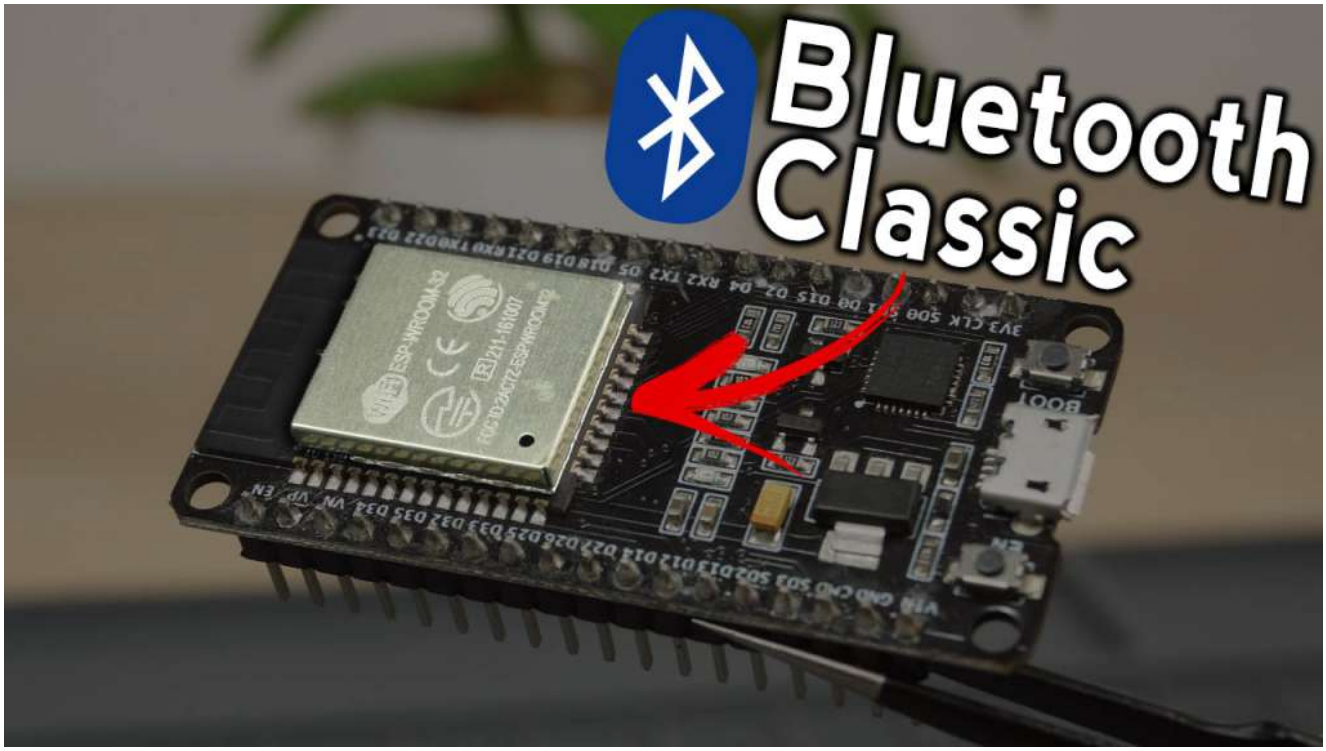
Using BLE is another communication protocol you can use with the ESP32 boards besides Wi-Fi. We hope you found this tutorial useful. We have tutorials for other communication protocols that you may find useful.

- ESP32 Bluetooth Classic with Arduino IDE – Getting Started
- ESP32 Useful Wi-Fi Library Functions (Arduino IDE)
- ESP-MESH with ESP32 and ESP8266: Getting Started (painlessMesh library)
- Getting Started with ESP-NOW (ESP32 with Arduino IDE)

Learn more about the ESP32 with our resources:

ESP32 Bluetooth Classic with Arduino IDE – Getting Started

The ESP32 comes with Wi-Fi, Bluetooth Low Energy and Bluetooth Classic. In this tutorial, you'll learn how to use ESP32 Bluetooth Classic with Arduino IDE to exchange data between an ESP32 and an Android smartphone.

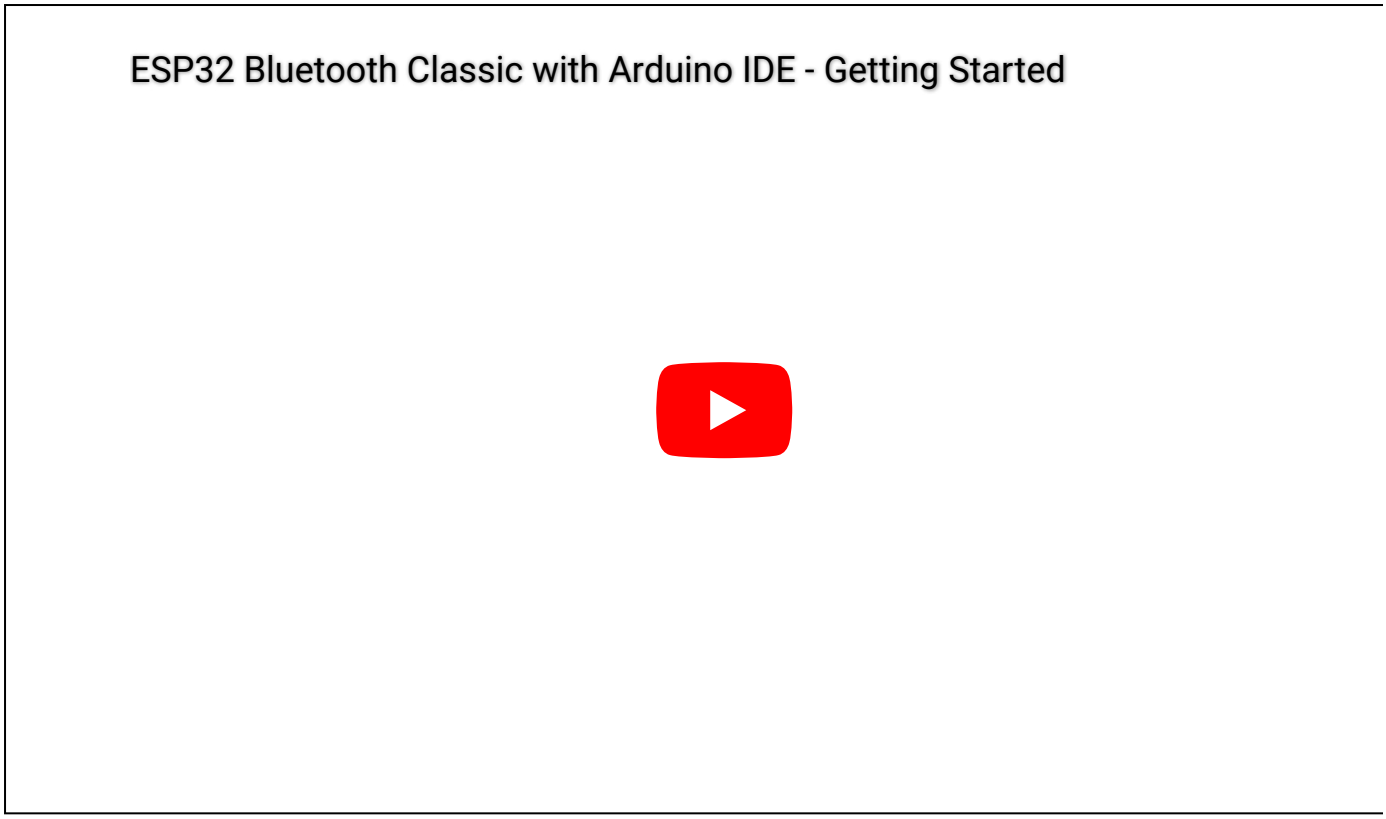


We'll control an ESP32 output, and send sensor readings to an Android smartphone using Bluetooth Classic.

Note: this project is **only** compatible with Android smartphones.

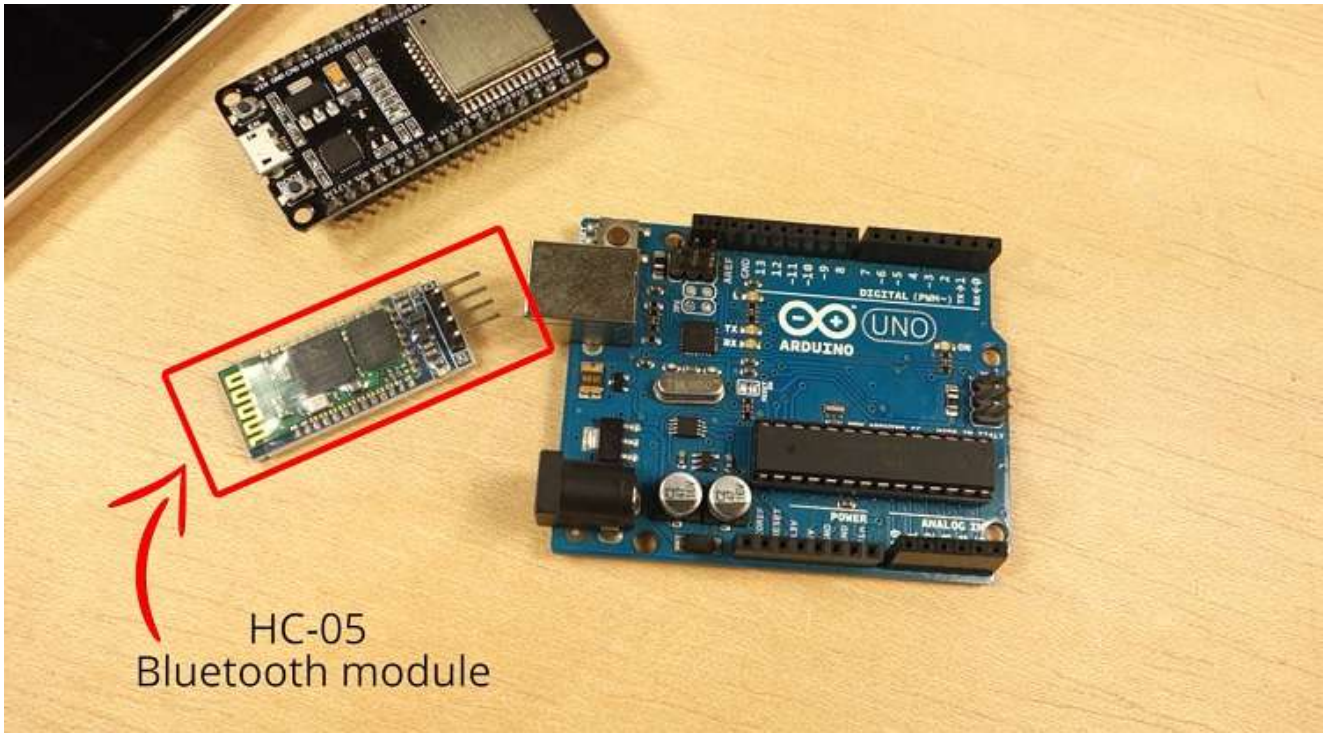
Watch the Video Tutorial

You can watch the video tutorial or keep reading this page for the written instructions.



Bluetooth Classic with ESP32

At the moment, using Bluetooth Classic is much more simpler than Bluetooth Low Energy. If you've already programmed an Arduino with a Bluetooth module like the HC-06, this is very similar. It uses the standard serial protocol and functions.



In this tutorial, we'll start by using an example that comes with the Arduino IDE. Then, we'll build a simple project to exchange data between the ESP32 and your

Android smartphone.

Parts Required

To follow this tutorial, you need the following parts:

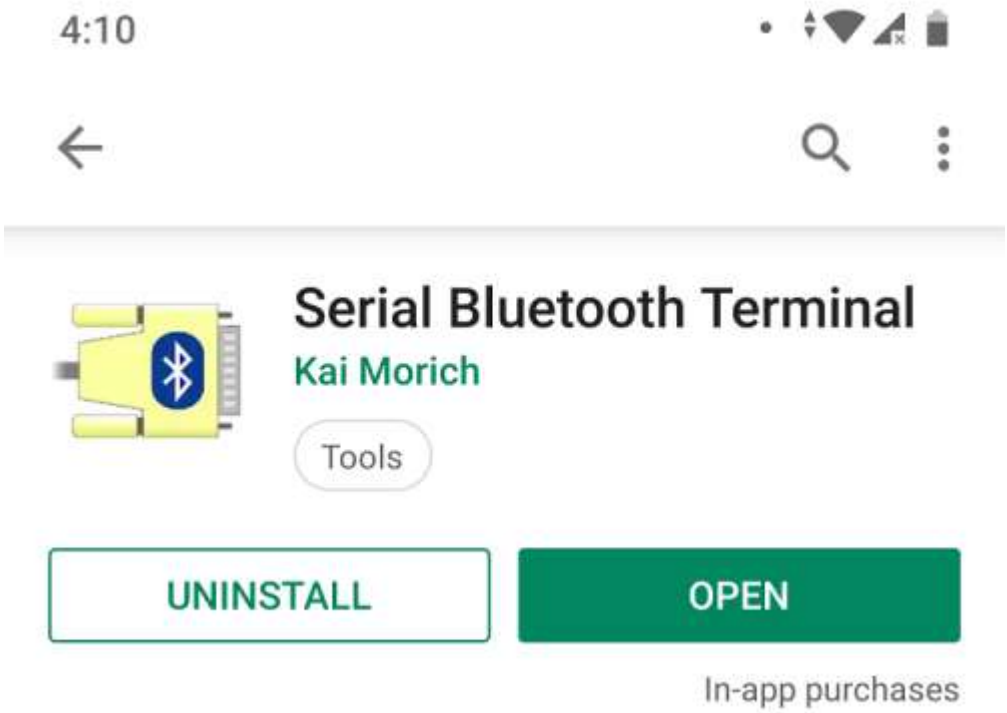
- ESP32 DOIT DEVKIT V1 Board (read Best ESP32 development boards)
- Android Smartphone with Bluetooth
- 5mm LED
- 330 Ohm resistor
- DS18B20 temperature sensor
- 4.7k Ohm resistor
- Jumper wires
- Breadboard

You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!

Bluetooth Terminal Application

To proceed with this tutorial, you need a Bluetooth Terminal application installed in your smartphone.

We recommend using the Android app “Serial Bluetooth Terminal” available in the Play Store.



Serial to Serial Bluetooth

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:

- **Windows:** instructions – ESP32 Board in Arduino IDE
- **Mac and Linux:** instructions – ESP32 Board in Arduino IDE

Open your Arduino IDE, and go to **File** > **Examples** > **BluetoothSerial** > **SerialtoSerialBT**.

The following code should load.

```
//This example code is in the Public Domain (or CC0 licensed, a
//By Evandro Copercini - 2018
//
//This example creates a bridge between Serial and Classical Bl
//and also demonstrate that SerialBT have the same functionalit

#include "BluetoothSerial.h"

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_EN
#error Bluetooth is not enabled! Please run `make menuconfig` t
```

```
#endif

BluetoothSerial SerialBT;

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

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

View raw code

How the Code Works

This code establishes a two-way serial Bluetooth communication between two devices.

The code starts by including the `BluetoothSerial` library.

```
#include "BluetoothSerial.h"
```

The next three lines check if Bluetooth is properly enabled.

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

Then, create an instance of `BluetoothSerial` called `SerialBT`:

```
BluetoothSerial SerialBT;
```

setup()

In the `setup()` initialize a serial communication at a baud rate of 115200.

```
Serial.begin(115200);
```

Initialize the Bluetooth serial device and pass as an argument the Bluetooth Device name. By default it's called **`ESP32test`** but you can rename it and give it a unique name.

```
SerialBT.begin("ESP32test"); //Bluetooth device name
```

loop()

In the `loop()`, send and receive data via Bluetooth Serial.

In the first if statement, we check if there are bytes being received in the serial port. If there are, send that information via Bluetooth to the connected device.

```
if (Serial.available()) {
  SerialBT.write(Serial.read());
}
```

`SerialBT.write()` sends data using bluetooth serial.

`Serial.read()` returns the data received in the serial port.

The next if statement, checks if there are bytes available to read in the Bluetooth Serial port. If there are, we'll write those bytes in the Serial Monitor.

```
  if (SerialBT.available()) {
    Serial.write(SerialBT.read());
  }
```

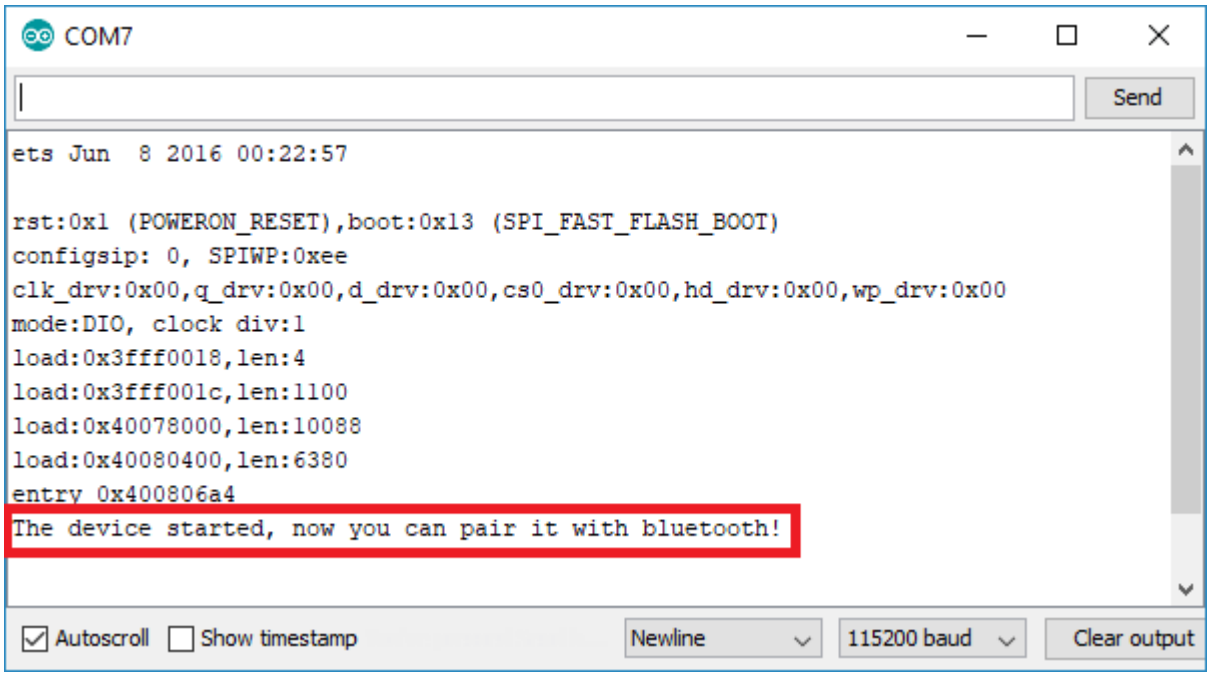
It will be easier to understand exactly how this sketch works in the demonstration.

Testing the Code

Upload the previous code to the ESP32. Make sure you have the right board and COM port selected.

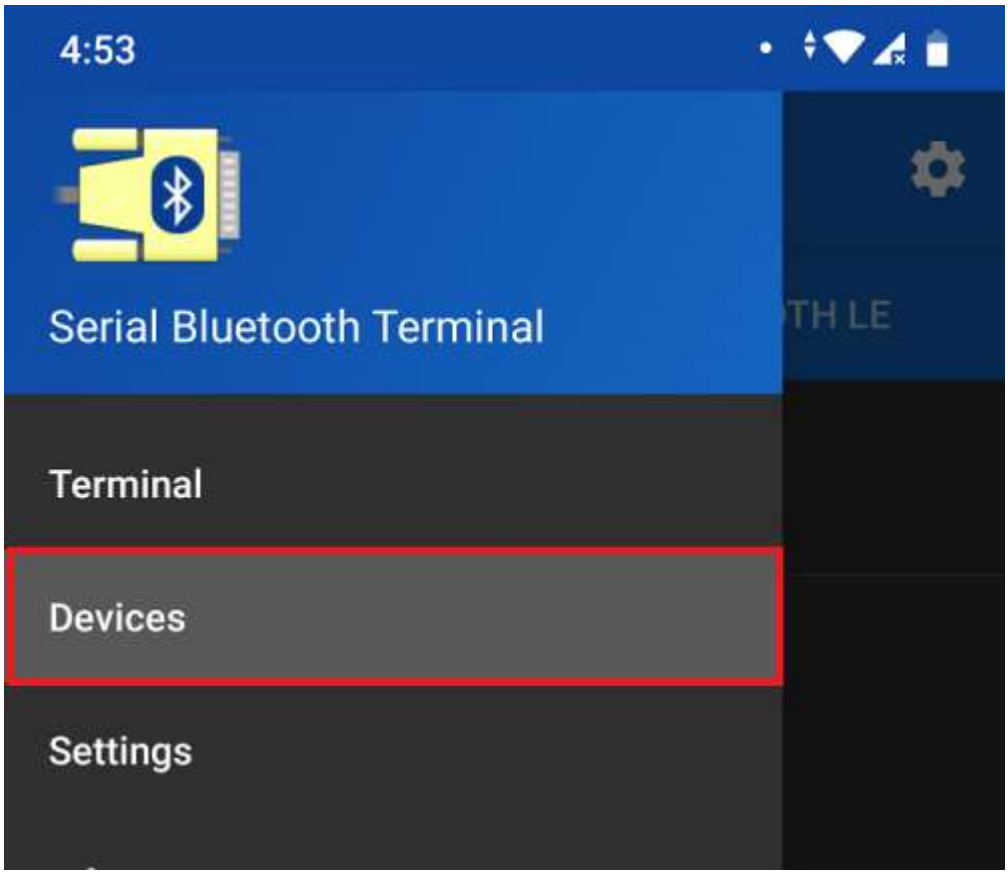
After uploading the code, open the Serial Monitor at a baud rate of 115200. Press the ESP32 Enable button.

After a few seconds, you should get a message saying: "*The device started, now you can pair it with bluetooth!*".

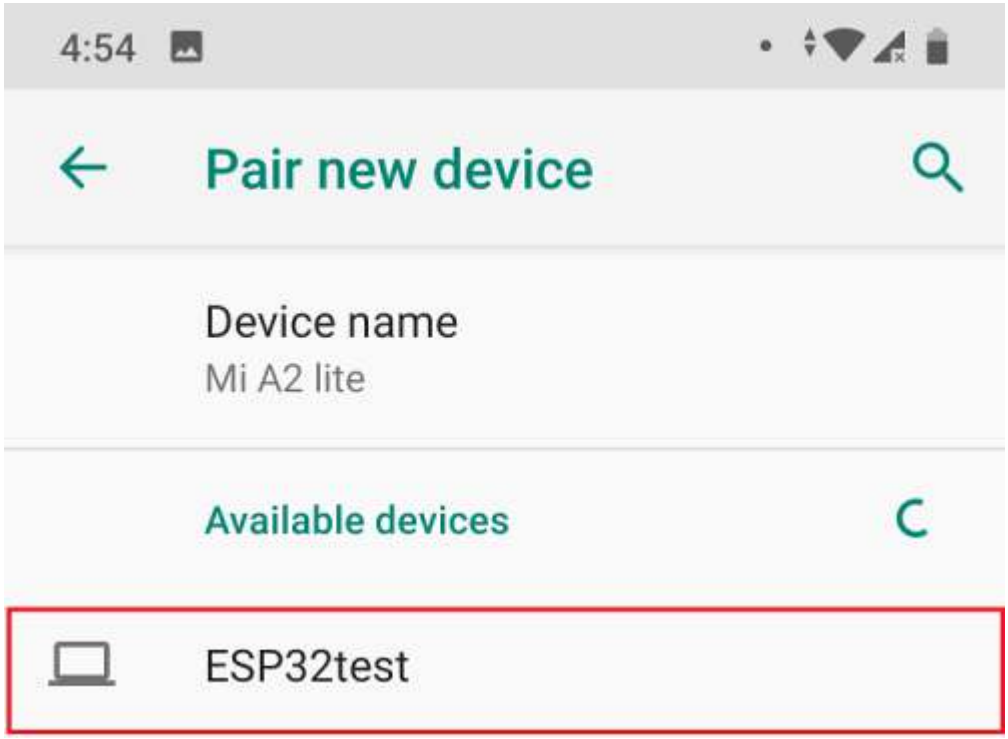


Go to your smartphone and open the "**Serial Bluetooth Terminal**" app. Make sure you've enable your smartphone's Bluetooth.

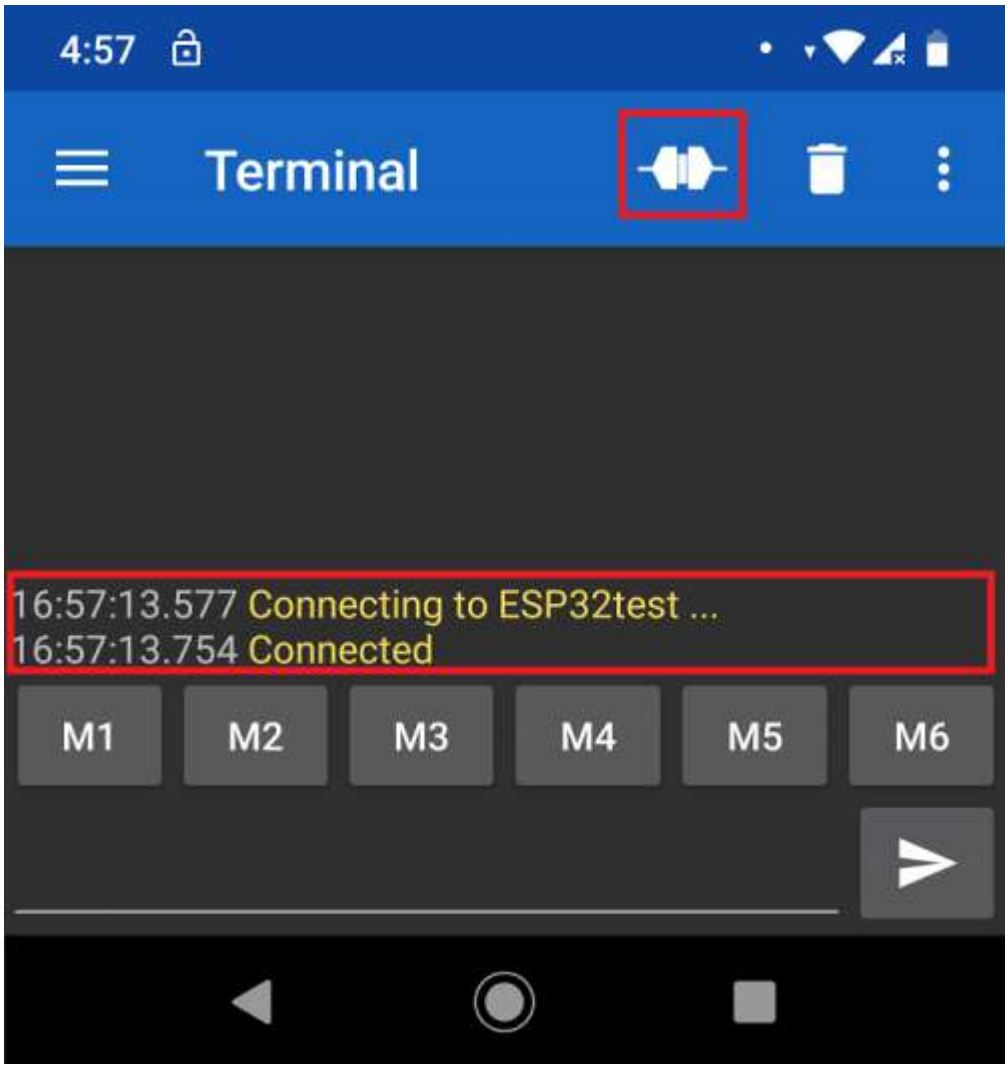
To connect to the ESP32 for the first time, you need to pair a new device.
Go to **Devices.**



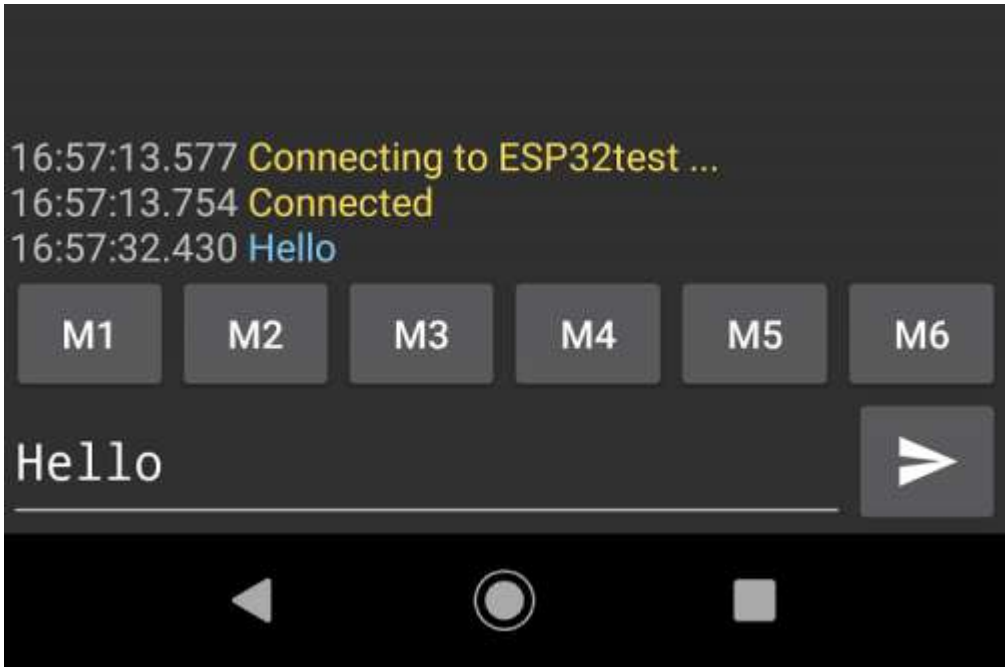
Click the settings icon, and select **Pair new device**. You should get a list with the available Bluetooth devices, including the **ESP32test**. Pair with the **ESP32test**.



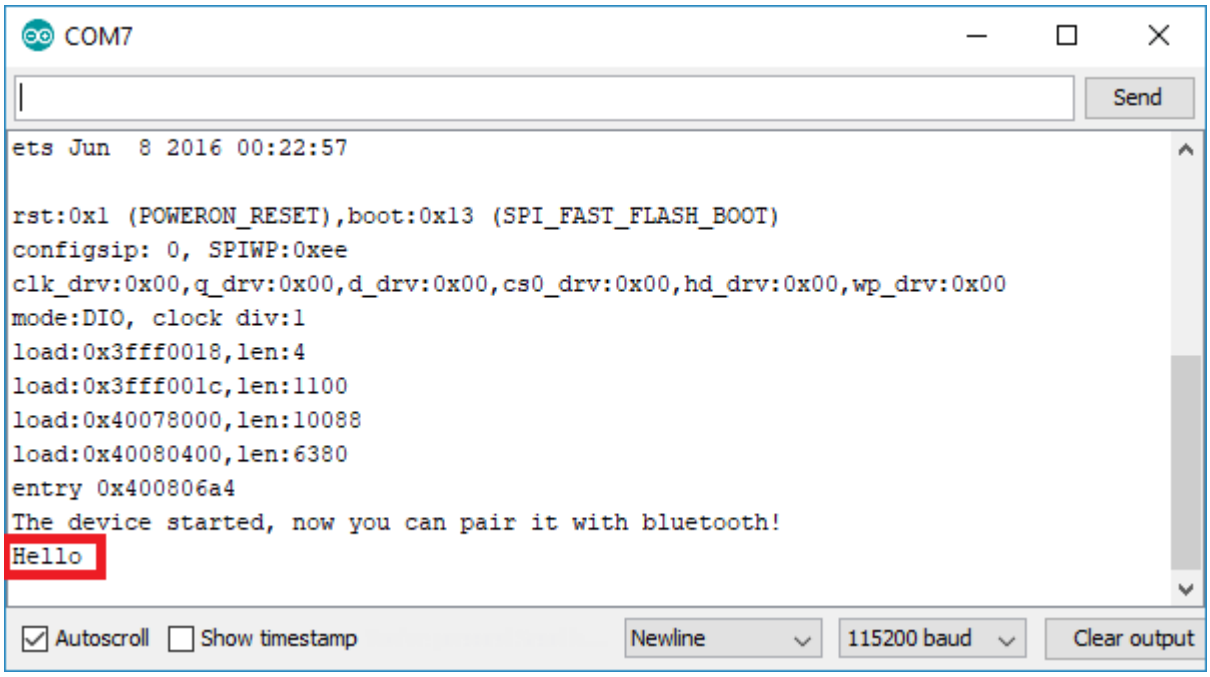
Then, go back to the Serial Bluetooth Terminal. Click the icon at the top to connect to the ESP32. You should get a “**Connected”** message.



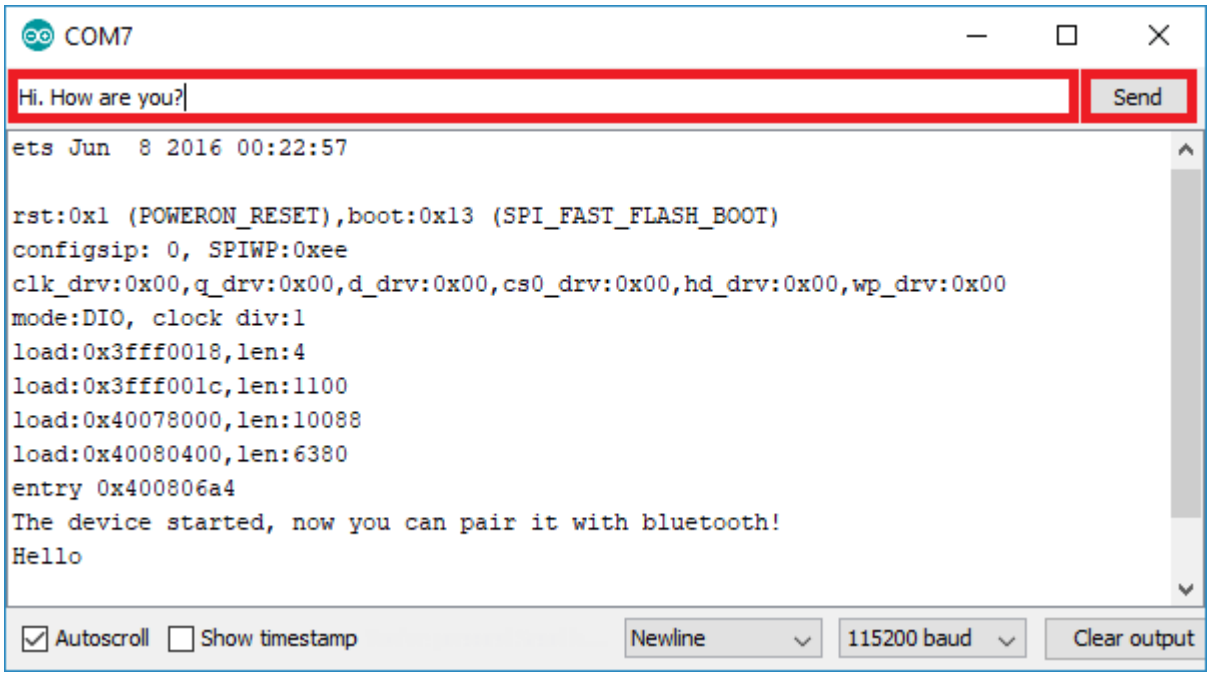
After that, type something in the Serial Bluetooth Terminal app. For example, "**Hello**".



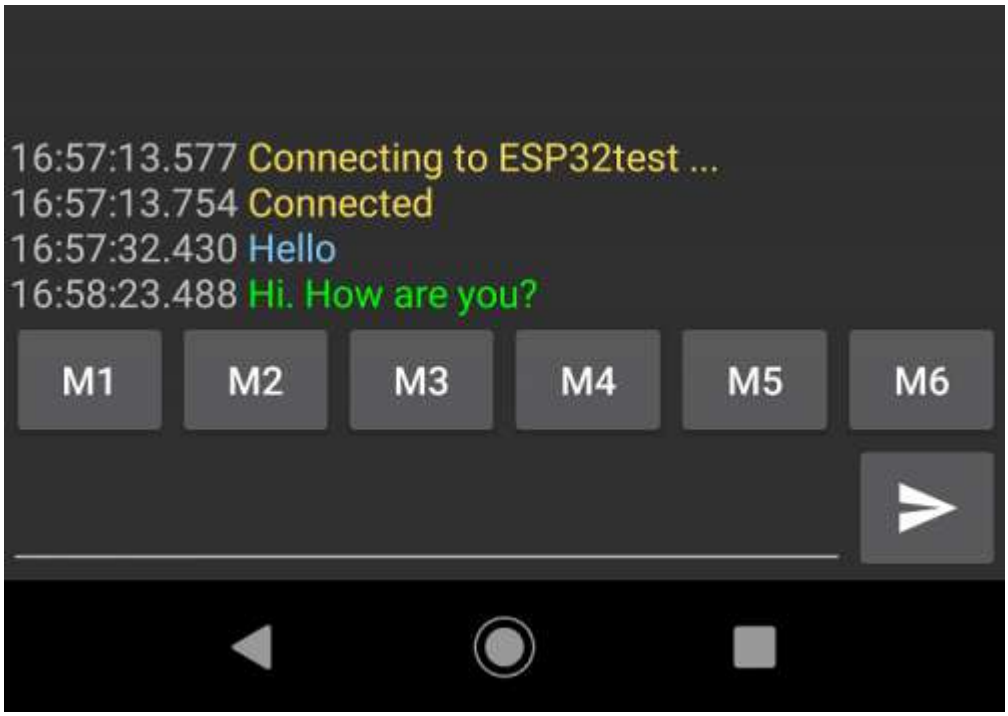
You should instantly receive that message in the Arduino IDE Serial Monitor.



You can also exchange data between your Serial Monitor and your smartphone. Type something in the Serial Monitor top bar and press the "**Send**" button.



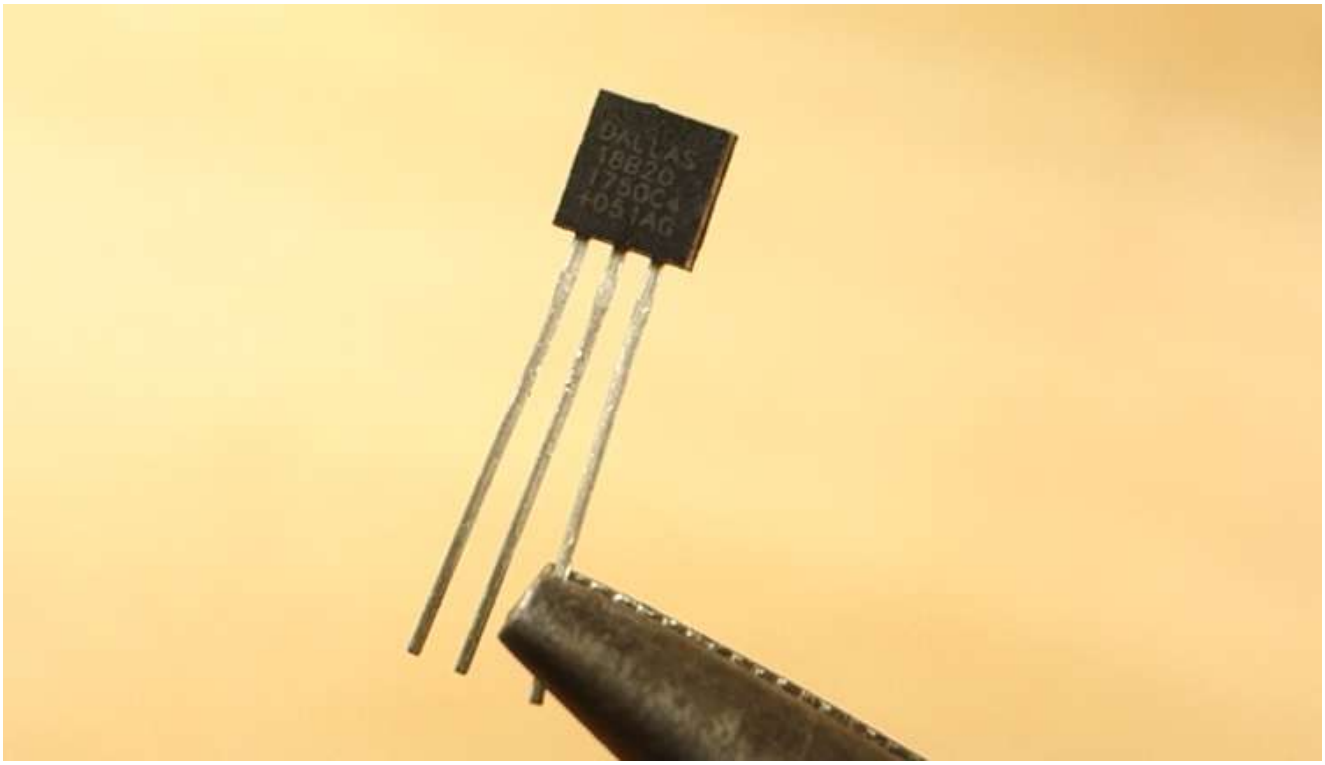
You should instantly receive that message in the Serial Bluetooth Terminal App.



Exchange Data using Bluetooth Serial

Now that you know how to exchange data using Bluetooth Serial, you can modify the previous sketch to make something useful. For example, control the ESP32 outputs when you receive a certain message, or send data to your smartphone like sensor readings.

The project we'll build sends temperature readings every 10 seconds to your smartphone. We'll be using the DS18B20 temperature sensor.

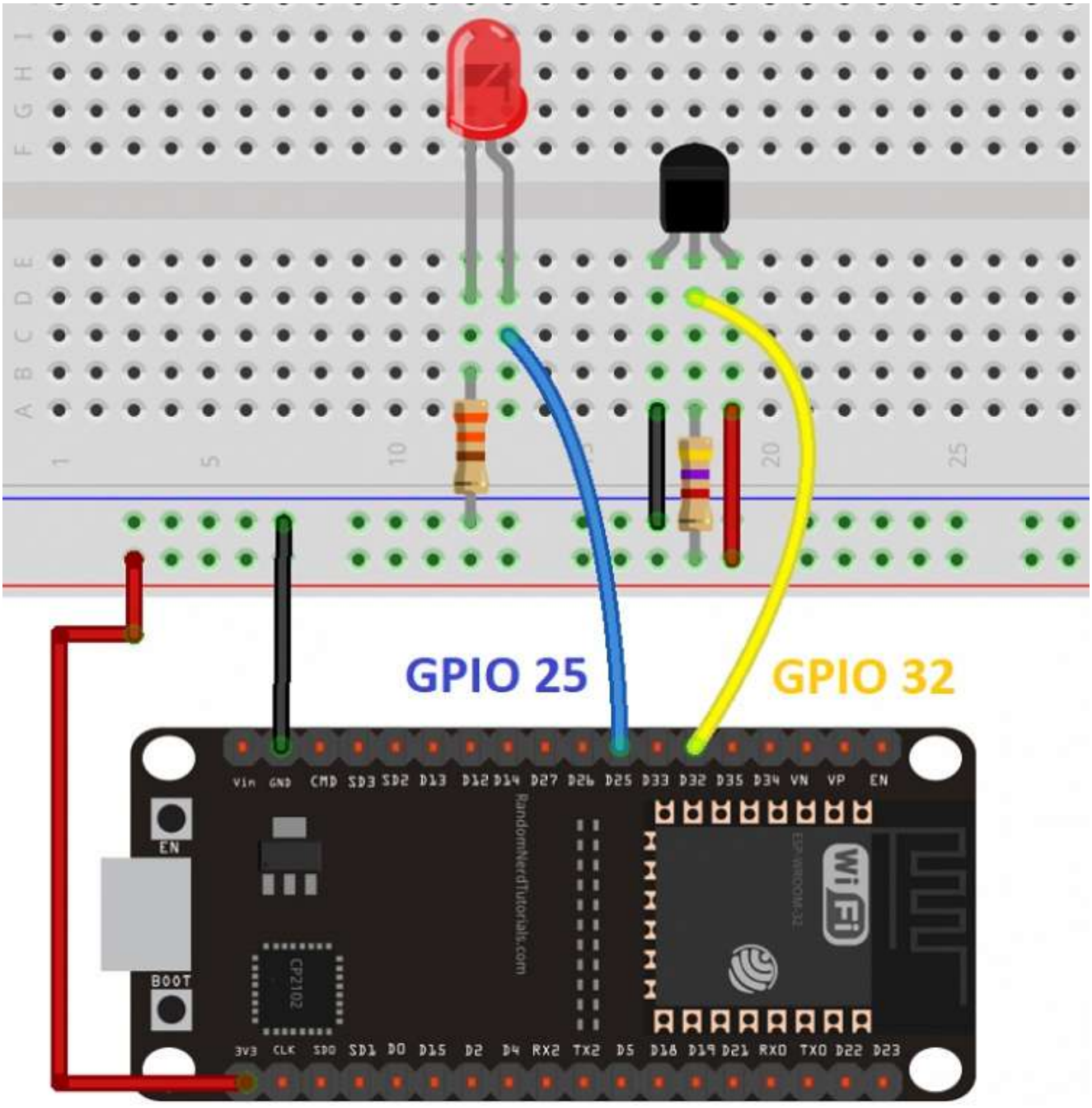


Through the Android app, we'll send messages to control an ESP32 output. When the ESP32 receives the `led_on` message, we'll turn the GPIO on, when it receives the `led_off` message, we'll turn the GPIO off.

Schematic

Before proceeding with this project, assemble the circuit by following the next schematic diagram.

Connect an LED to GPIO25 , and connect the DS18B20 data pin to GPIO32 .



Recommended reading: ESP32 Pinout Reference: Which GPIO pins should you use?

Code

To work with the DS18B20 temperature sensor, you need to install the One Wire library by Paul Stoffregen and the Dallas Temperature library. Follow the next instructions to install these libraries, if you haven't already.

One Wire library

1. Click here to download the One Wire library. You should have a .zip folder in your Downloads
2. Unzip the *.zip* folder and you should get **OneWire-master** folder
3. Rename your folder from ~~**OneWire-master**~~ to **OneWire**
4. Move the **OneWire** folder to your Arduino IDE installation **libraries** folder
5. Finally, re-open your Arduino IDE

Dallas Temperature library

1. Click here to download the Dallas Temperature library. You should have a .zip folder in your Downloads
2. Unzip the *.zip* folder and you should get **Arduino-Temperature-Control-Library-master** folder
3. Rename your folder from ~~**Arduino-Temperature-Control-Library-master**~~ to **DallasTemperature**
4. Move the **DallasTemperature**folder to your Arduino IDE installation **libraries** folder
5. Finally, re-open your Arduino IDE

After assembling the circuit and installing the necessary libraries, copy the following sketch to your Arduino IDE.

```
/*********
  Rui Santos
  Complete project details at https://randomnerdtutorials.com
*********/

// Load libraries
#include "BluetoothSerial.h"
#include <OneWire.h>
```

```
#include <DallasTemperature.h>

// Check if Bluetooth configs are enabled
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_EN
#error Bluetooth is not enabled! Please run `make menuconfig` t
#endif

// Bluetooth Serial object
BluetoothSerial SerialBT;

// GPIO where LED is connected to
const int ledPin =  25;

// GPIO where the DS18B20 is connected to
const int oneWireBus = 32;
// Setup a oneWire instance to communicate with any OneWire dev
OneWire oneWire(oneWireBus);
// Pass our oneWire reference to Dallas Temperature sensor
```

View raw code

How the Code Works

Let's take a quick look at the code and see how it works.

Start by including the necessary libraries. The `BluetoothSerial` library for Bluetooth, and the `OneWire` and `DallasTemperature` for the DS18B20 temperature sensor.

```
#include "BluetoothSerial.h"
#include <OneWire.h>
#include <DallasTemperature.h>
```

Create a `BluetoothSerial` instance called `SerialBT`.

```
BluetoothSerial SerialBT;
```

Create a variable called `ledPin` to hold the GPIO you want to control. In this case, GPIO25 has an LED connected.

```
const int ledPin =  25;
```

Define the DS18B20 sensor pin and create objects to make it work. The temperature sensor is connected to GPIO32 .

```
// GPIO where the DS18B20 is connected to
const int oneWireBus = 32;
// Setup a oneWire instance to communicate with any OneWire devic
OneWire oneWire(oneWireBus);
// Pass our oneWire reference to Dallas Temperature sensor
DallasTemperature sensors(&oneWire);
```

Create an empty string called `message` to store the received messages.

```
String message = "";
```

Create a char variable called `incomingChar` to save the characters coming via Bluetooth Serial.

```
char incomingChar;
```

The `temperatureString` variable holds the temperature readings to be sent via Bluetooth.

```
String temperatureString = "";
```

Create auxiliar timer variables to send readings every 10 seconds.

```
unsigned long previousMillis = 0;   // Stores last time temperatu
const long interval = 10000;        // interval at which to publi
```

setup()

In the `setup()`, set the `ledPin` as an output.

```
pinMode(ledPin, OUTPUT);
```

Initialize the ESP32 as a bluetooth device with the “ESP32” name.

```
SerialBT.begin("ESP32"); //Bluetooth device name
```

loop()

In the `loop()`, send the temperature readings, read the received messages and execute actions accordingly.

The following snippet of code, checks if 10 seconds have passed since the last reading. If it’s time to send a new reading, we get the latest temperature and save it in Celsius and Fahrenheit in the `temperatureString` variable.

```
unsigned long currentMillis = millis();
if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
    sensors.requestTemperatures();
    temperatureString = " " + String(sensors.getTempCByIndex(0))
```

Then, to send the `temperatureString` via bluetooth, use `SerialBT.println()`.

```
  SerialBT.println(temperatureString);
```

The next if statement reads incoming messages. When you receive messages via serial, you receive a character at a time. You know that the message ended, when you receive **\n**.

So, we check if there's data available in the Bluetooth serial port.

```
  if (SerialBT.available()) {
```

If there is, we'll save the characters in the `incomingChar` variable.

```
  char incomingChar = SerialBT.read();
```

If the `incomingChar` is different than `\n`, we'll concatenate that char character to our message.

```
  if (incomingChar!= '\n'){
    message += String(incomingChar);
  }
```

When we're finished reading the characters, we clear the `message` variable. Otherwise all received messages would be appended to each other.

```
  message = "";
```

After that, we have two if statements to check the content of the message. If the message is **led_on**, the LED turns on.

```
  if (message =="led_on"){
     digitalWrite(ledPin, HIGH);
```

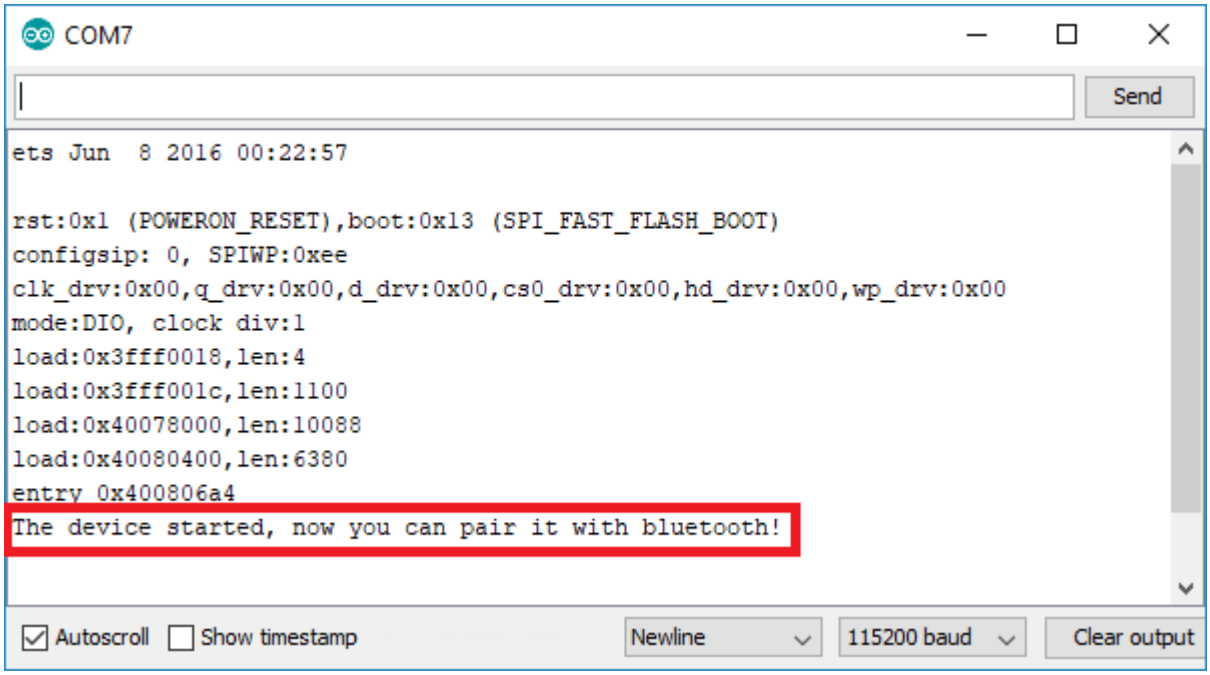
```
}
```

If the message is **`led_off`** , the LED turns off.

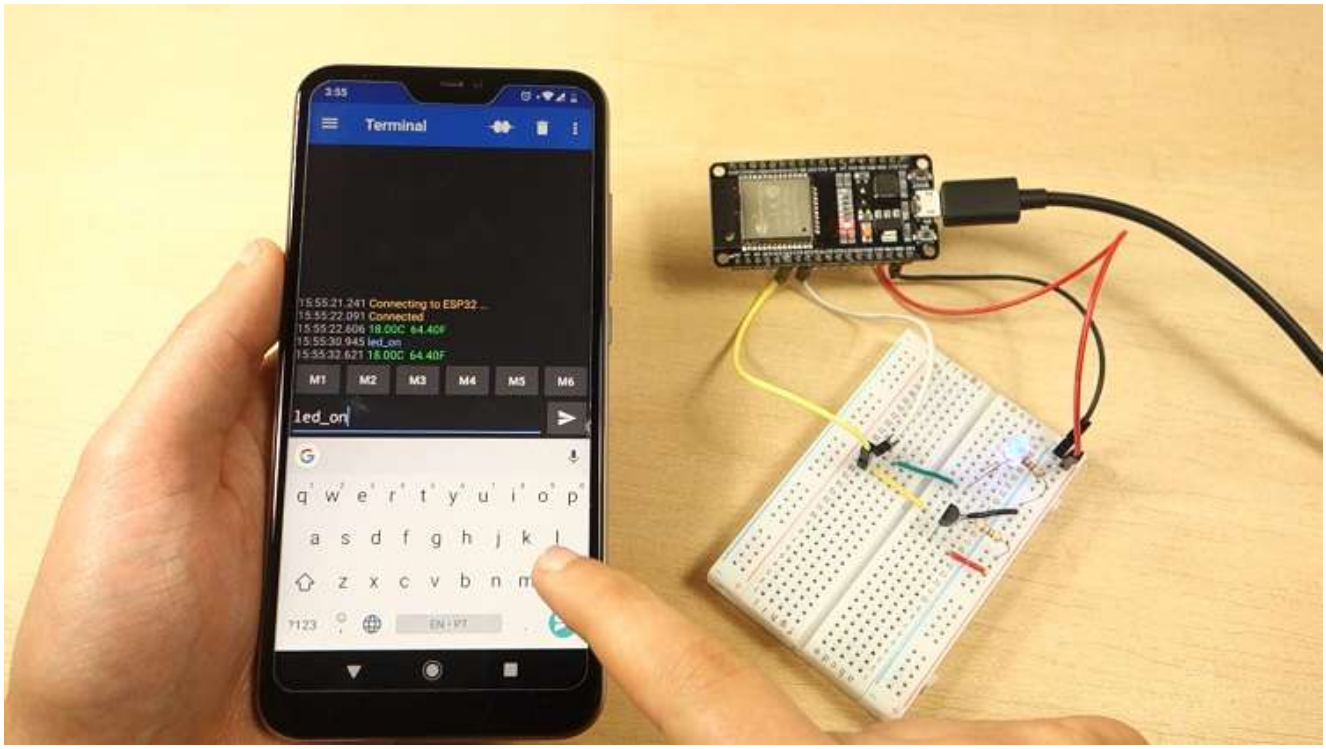
```
else if (message =="led_off"){
  digitalWrite(ledPin, LOW);
}
```

Testing the Project

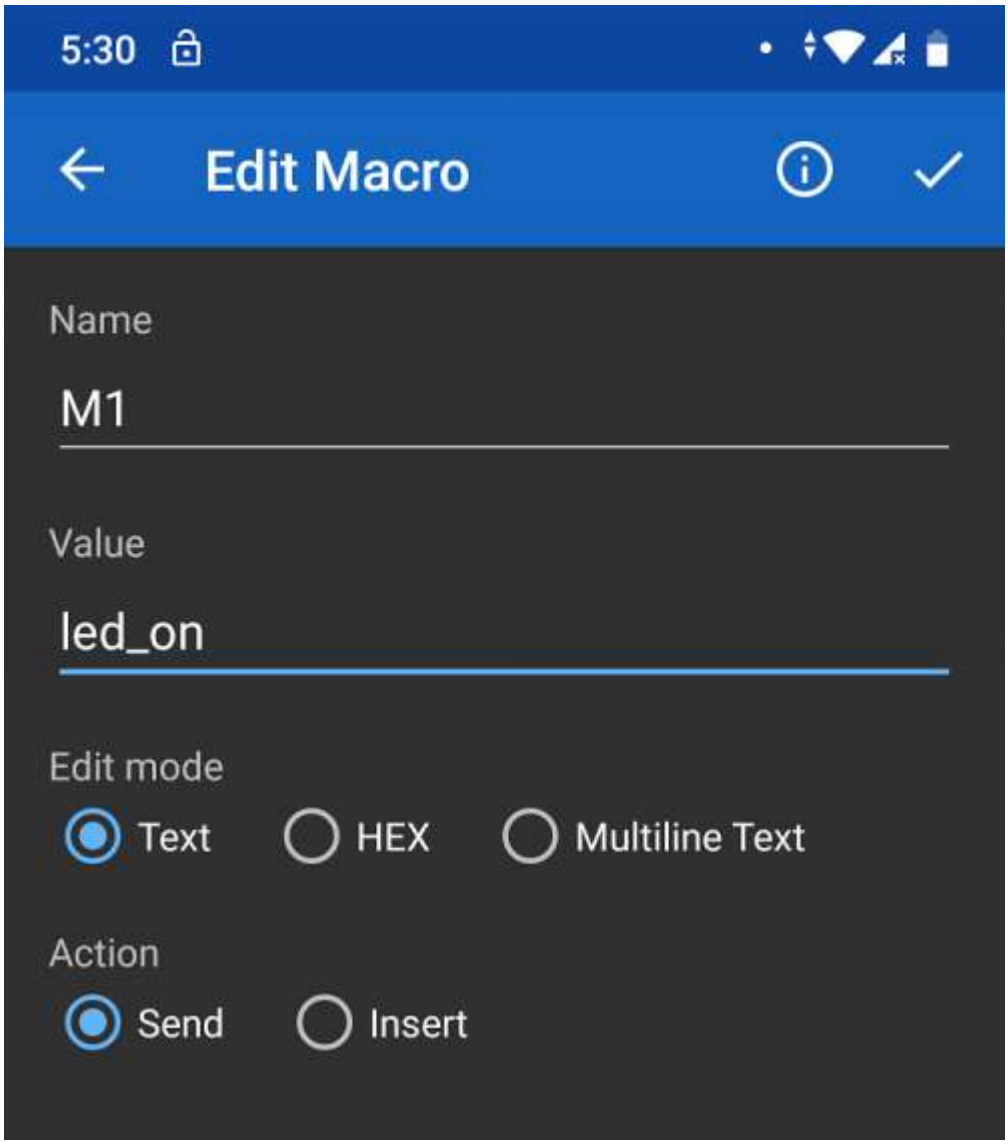
Upload the previous sketch to your ESP32 board. Then, open the Serial Monitor, and press the ESP32 Enable button. When you receive the following message, you can go to your smartphone and connect with the ESP32.



Then, you can write the**“led_on”** and **“led_off”** messages to control the LED.

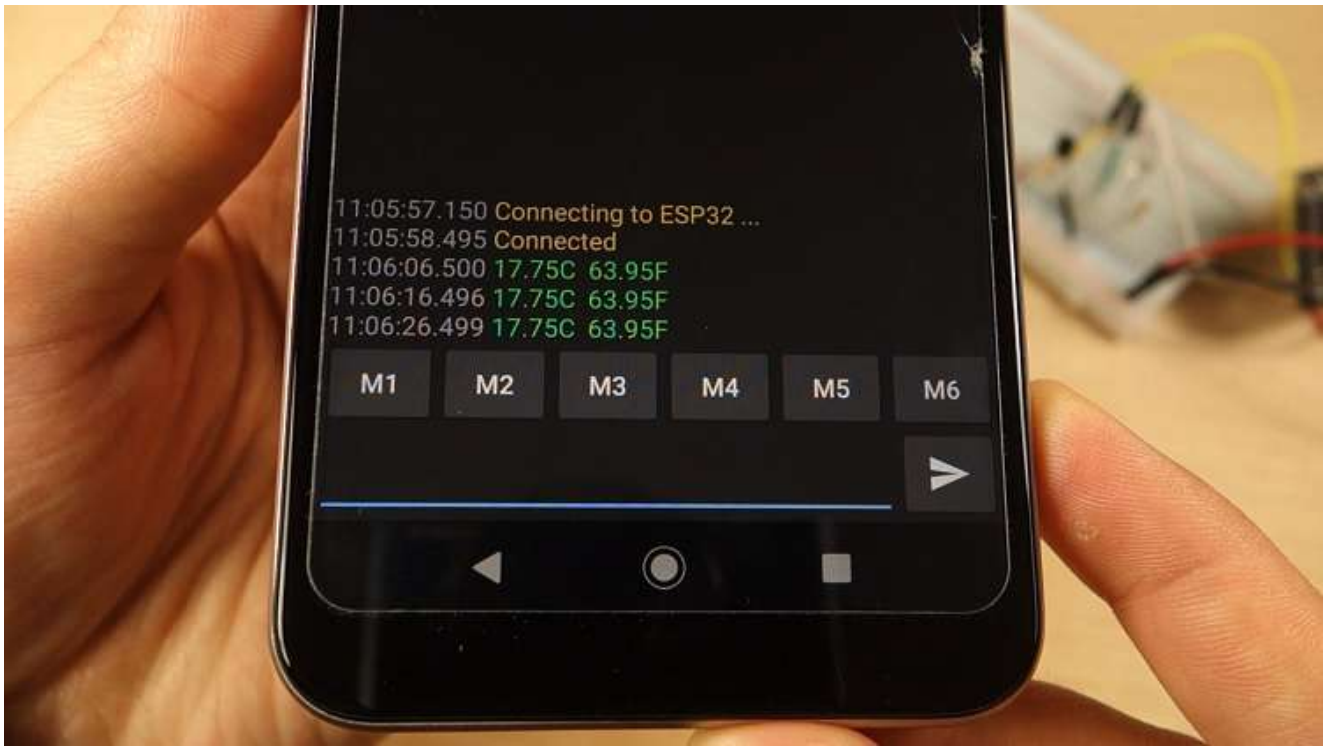


The application has several buttons in which you can save default messages. For example, you can associate **M1** with the “**led_on**” message, and **M2** with the “**led_off**” message.



Now, you are able to control the ESP32 GPIOs.

At the same time, you should be receiving the temperature readings every 10 seconds.



Wrapping Up

In summary, the ESP32 supports BLE and Bluetooth Classic. Using Bluetooth Classic is as simple as using serial communication and its functions.

If you want to learn how to use BLE with the ESP32, you can read our guide:

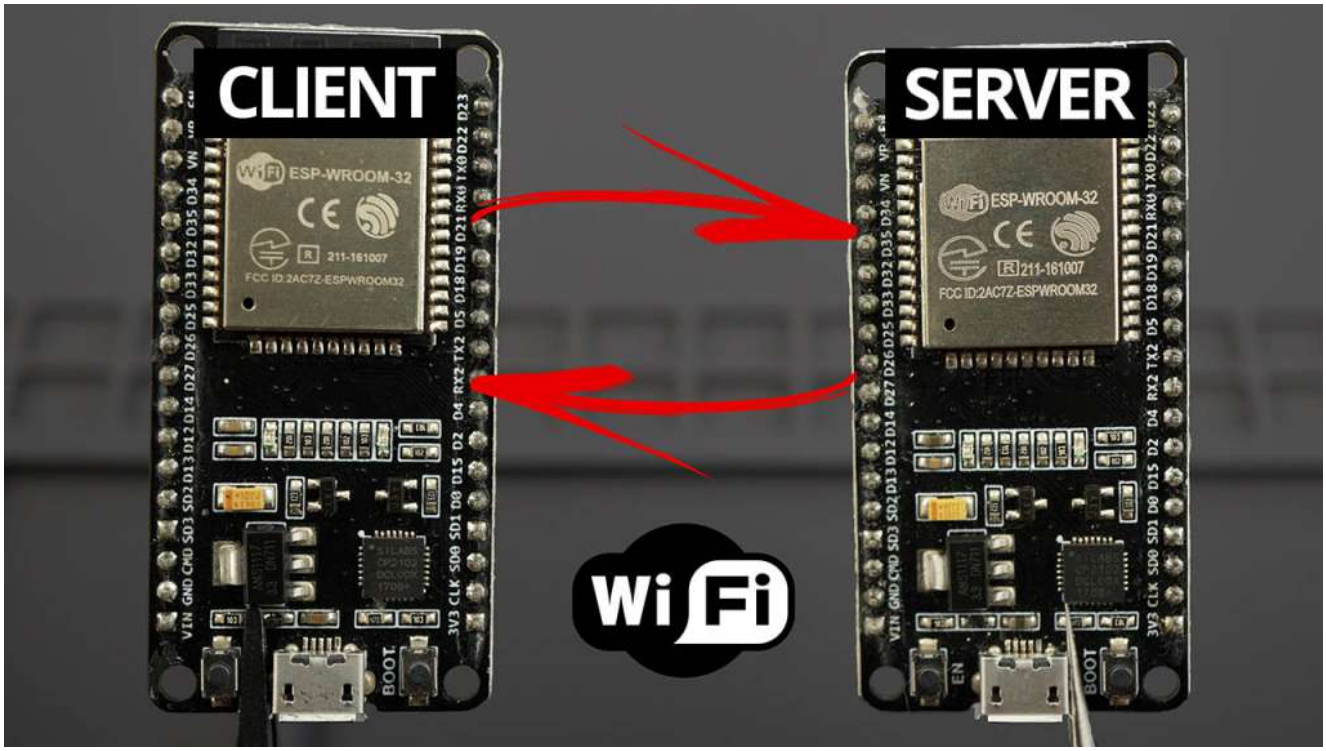
- Getting Started with ESP32 Bluetooth Low Energy (BLE) on Arduino IDE

We hope you've found this tutorial useful. For more projects with the ESP32 you can check our project's compilation: 20+ ESP32 Projects and Tutorials.

> This tutorial is a preview of the "Learn ESP32 with Arduino IDE" course. If you like this project, make sure you take a look at the ESP32 course page where we cover this and a lot more topics with the ESP32.

ESP32 Client-Server Wi-Fi Communication Between Two Boards

This guide shows how to setup an HTTP communication between two ESP32 boards to exchange data via Wi-Fi without an internet connection (router). In simple words, you'll learn how to send data from one board to the other using HTTP requests. The ESP32 boards will be programmed using Arduino IDE.



For demonstration purposes, we'll send BME280 sensor readings from one board to the other. The receiver will display the readings on an OLED display.

If you have an ESP8266 board, you can read this dedicated guide: ESP8266 NodeMCU Client-Server Wi-Fi Communication.

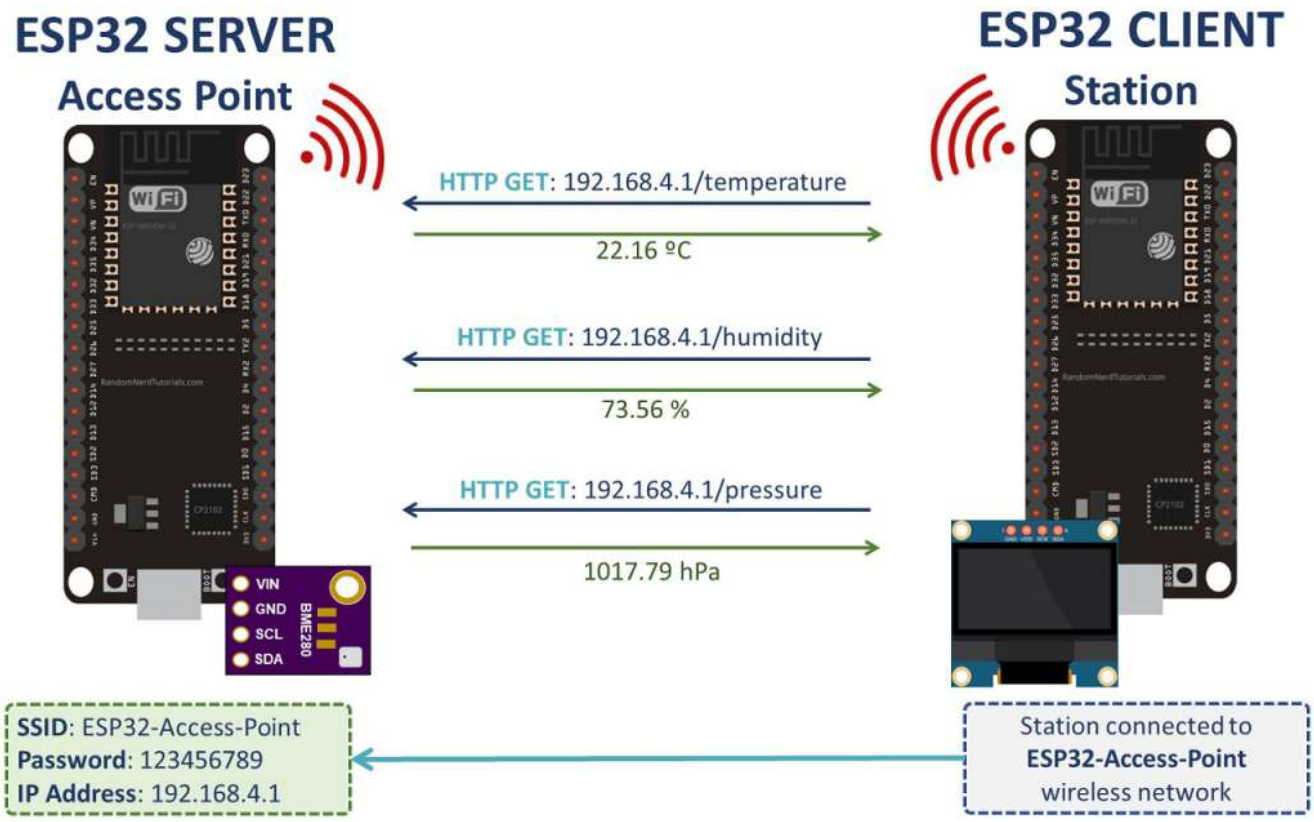
Watch the Video Demonstration

To see how the project works, you can watch the following video demonstration:

ESP32 Client-Server Wi-Fi Communication Between Two Boards (ESP8266 Co...

Project Overview

One ESP32 board will act as a server and the other ESP32 board will act as a client. The following diagram shows an overview of how everything works.

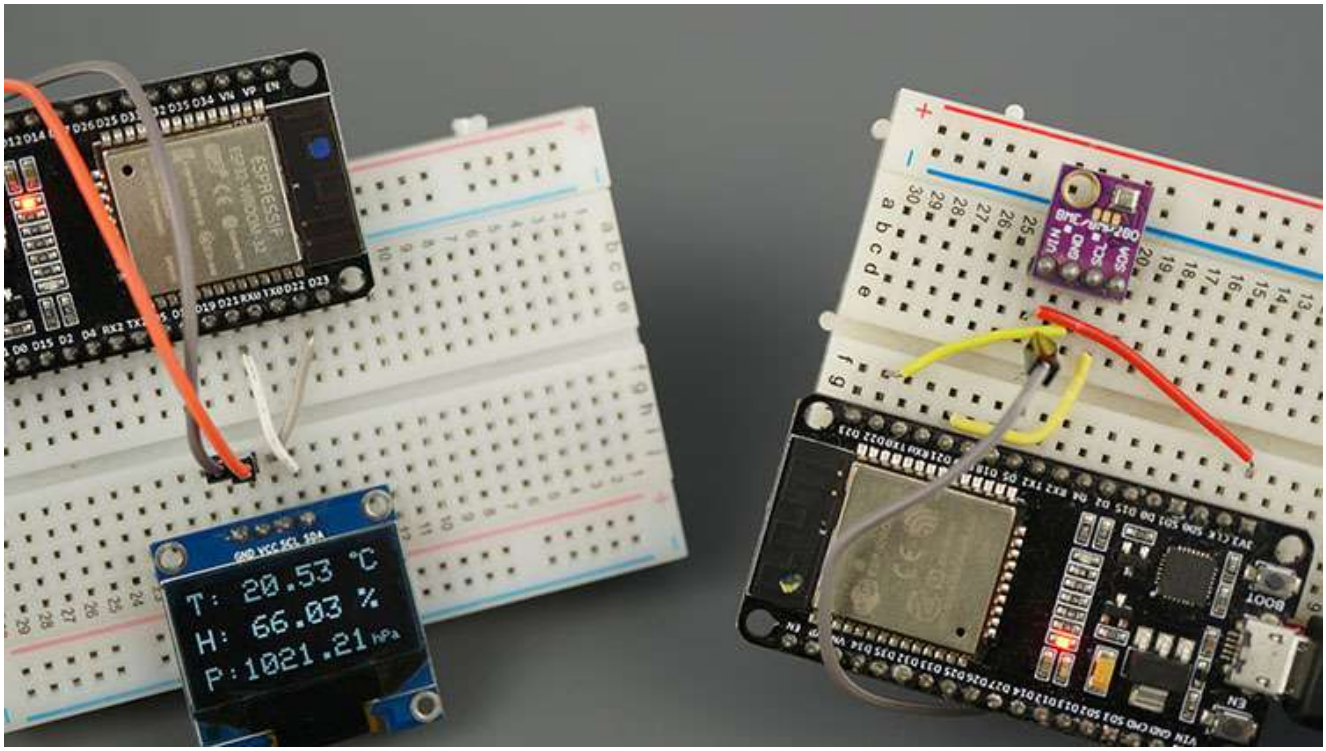


- The ESP32 server creates its own wireless network (ESP32 Soft-Access Point). So, other Wi-Fi devices can connect to that network (**SSID:** ESP32-Access-Point, **Password:** 123456789).
- The ESP32 client is set as a station. So, it can connect to the ESP32 server wireless network.
- The client can make HTTP GET requests to the server to request sensor data or any other information. It just needs to use the IP address of the server to make a request on a certain route: `/temperature`, `/humidity` or `/pressure`.
- The server listens for incoming requests and sends an appropriate response with the readings.
- The client receives the readings and displays them on the OLED display.

As an example, the ESP32 client requests temperature, humidity and pressure to the server by making requests on the server IP address followed by `/temperature`, `/humidity` and `/pressure`, respectively.

The ESP32 server is listening on those routes and when a request is made, it sends the corresponding sensor readings via HTTP response.

Parts Required



For this tutorial, you need the following parts:

- 2x ESP32 Development boards – read Best ESP32 Boards Review
- BME280 sensor
- I2C SSD1306 OLED display
- Jumper Wires
- Breaboard

You can use the preceding links or go directly to MakerAdvisor.com/tools to find all the parts for your projects at the best price!

Installing Libraries

For this tutorial you need to install the following libraries:

Asynchronous Web Server Libraries

We'll use the following libraries to handle HTTP request:

- ESPAsyncWebServer library (download ESPAsyncWebServer library)
- Async TCP library (download AsyncTCP library)

These libraries are not available to install through the Library Manager. So, you need to unzip the libraries and move them to the Arduino IDE installation libraries folder.

Alternatively, you can go to **Sketch** > **Include Library** > **Add .ZIP library…** and select the libraries you've just downloaded.

You may also like: Build an Asynchronous Web Server with the ESP32

BME280 Libraries

The following libraries can be installed through the Arduino Library Manager. Go to **Sketch** > **Include Library**> **Manage Libraries** and search for the library name.

- Adafruit_BME280 library
- Adafruit unified sensor library

You may also like: Interface BME280 with ESP32 (Guide)

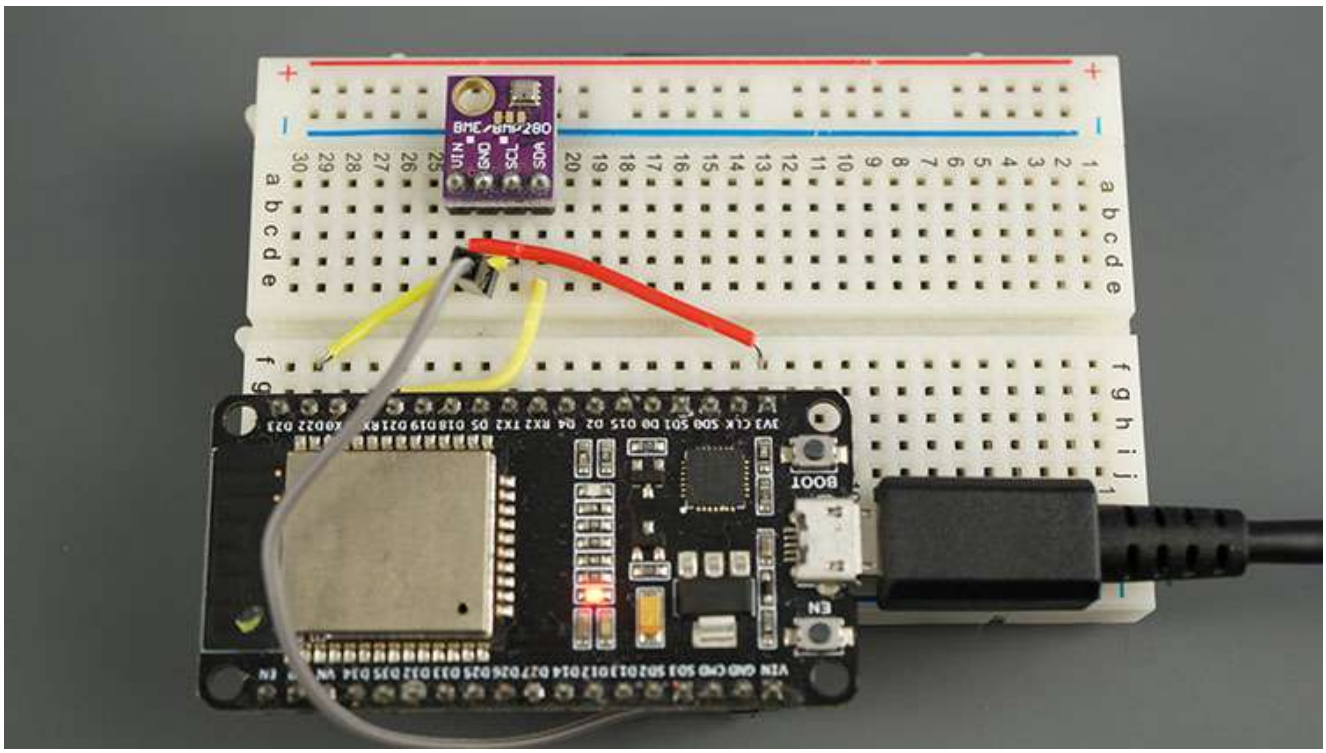
I2C SSD1306 OLED Libraries

To interface with the OLED display you need the following libraries. These can be installed through the Arduino Library Manager. Go to **Sketch** > **Include Library**> **Manage Libraries** and search for the library name.

- Adafruit SSD1306
- Adafruit GFX Library

You may also like: I2C SSD1306 OLED Display with ESP32 (Guide)

#1 ESP32 Server (Access Point)

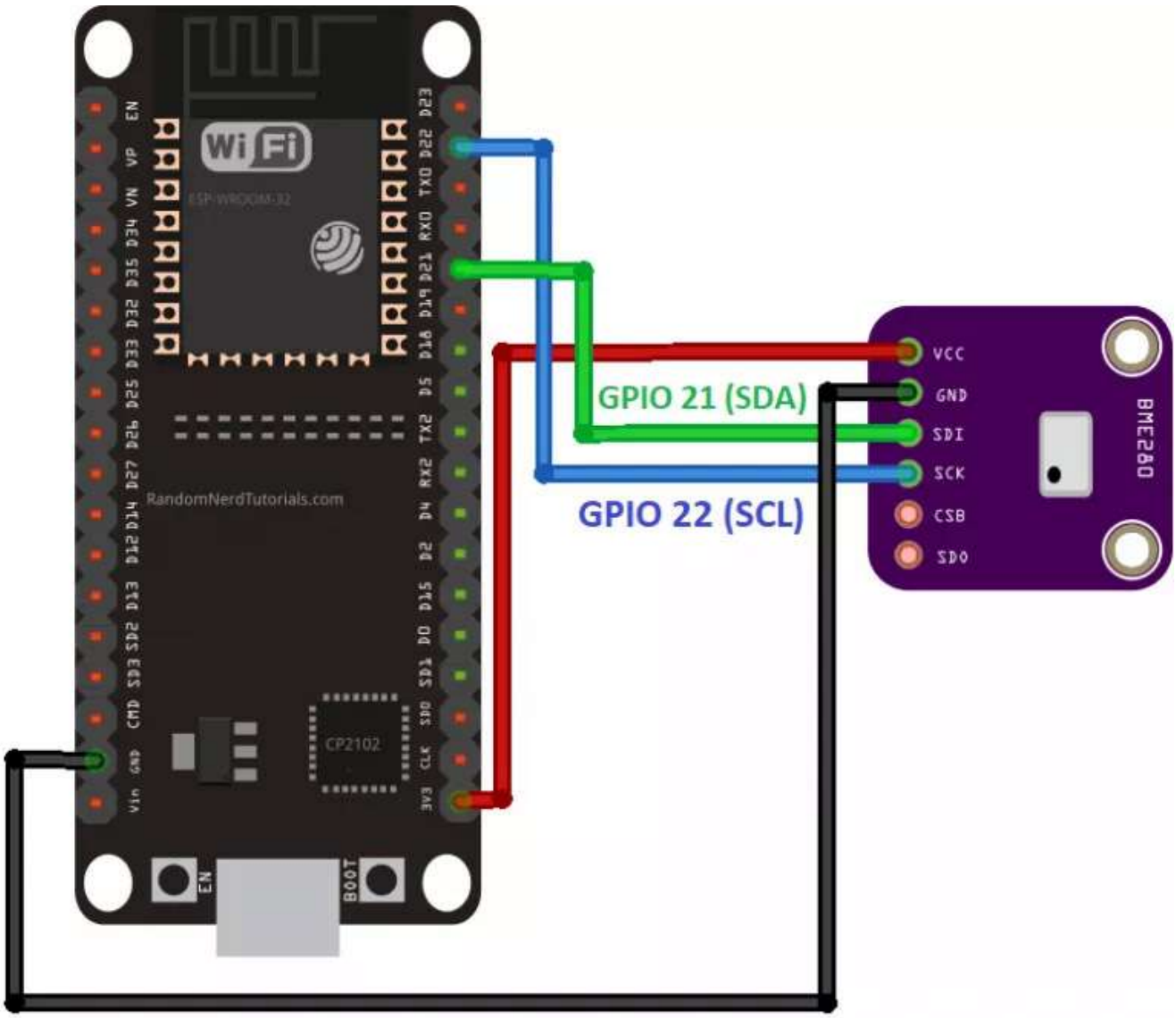


The ESP32 server is an Access Point (AP), that listens for requests on the `/temperature`, `/humidity` and `/pressure` URLs. When it gets requests on those URLs, it sends the latest BME280 sensor readings.

For demonstration purposes, we're using a BME280 sensor, but you can use any other sensor by modifying a few lines of code.

Schematic Diagram

Wire the ESP32 to the BME280 sensor as shown in the following schematic diagram.



BME280	ESP32
VIN/VCC	3.3V
GND	GND
SCL	GPIO 22
SDA	GPIO 21

Arduino Sketch for #1 ESP32 Server

Upload the following code to your board.

```
/*
  Rui Santos
  Complete project details at https://RandomNerdTutorials.com/e

  Permission is hereby granted, free of charge, to any person o
  of this software and associated documentation files.

  The above copyright notice and this permission notice shall b
  copies or substantial portions of the Software.
*/

// Import required libraries
#include "WiFi.h"
#include "ESPAsyncWebServer.h"

#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

// Set your access point network credentials
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";

/*#include <SPI.h>
#define BME_SCK 18
#define BME_MISO 19
#define BME_MOSI 23
```

View raw code

How the code works

Start by including the necessary libraries. Include the `WiFi.h` library and the `ESPAsyncWebServer.h` library to handle incoming HTTP requests.

```
#include "WiFi.h"
#include "ESPAsyncWebServer.h"
```

Include the following libraries to interface with the BME280 sensor.

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
```

In the following variables, define your access point network credentials:

```
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";
```

We're setting the SSID to `ESP32-Access-Point`, but you can give it any other name. You can also change the password. By default, its set to `123456789`.

Create an instance for the BME280 sensor called `bme`.

```
Adafruit_BME280 bme;
```

Create an asynchronous web server on port 80.

```
AsyncWebServer server(80);
```

Then, create three functions that return the temperature, humidity, and pressure as String variables.

```
String readTemp() {
  return String(bme.readTemperature());
  //return String(1.8 * bme.readTemperature() + 32);
}

String readHumi() {
  return String(bme.readHumidity());
}
```

```
String readPres() {
  return String(bme.readPressure() / 100.0F);
}
```

In the `setup()`, initialize the Serial Monitor for demonstration purposes.

```
Serial.begin(115200);
```

Set your ESP32 as an access point with the SSID name and password defined earlier.

```
WiFi.softAP(ssid, password);
```

Then, handle the routes where the ESP32 will be listening for incoming requests.

For example, when the ESP32 server receives a request on the `/temperature` URL, it sends the temperature returned by the `readTemp()` function as a char (that's why we use the `c_str()` method.

```
server.on("/temperature", HTTP_GET, [](AsyncWebServerRequest *req
  request->send_P(200, "text/plain", readTemp().c_str());
});
```

The same happens when the ESP receives a request on the `/humidity` and `/pressure` URLs.

```
server.on("/humidity", HTTP_GET, [](AsyncWebServerRequest *reques
  request->send_P(200, "text/plain", readHumi().c_str());
});
server.on("/pressure", HTTP_GET, [](AsyncWebServerRequest *reques
```

```
  request->send_P(200, "text/plain", readPres().c_str());
});
```

The following lines initialize the BME280 sensor.

```
bool status;

// default settings
// (you can also pass in a Wire library object like &Wire2)
status = bme.begin(0x76);
if (!status) {
  Serial.println("Could not find a valid BME280 sensor, check wir
  while (1);
}
```

Finally, start the server.

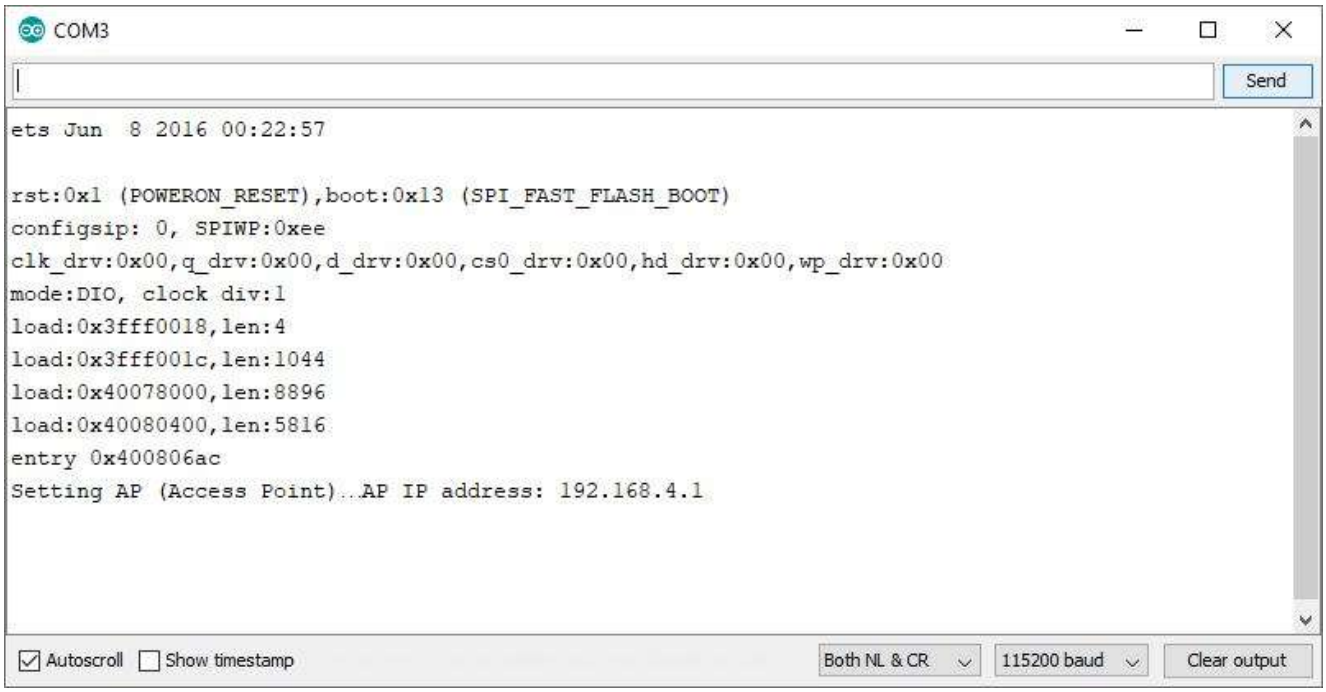
```
server.begin();
```

Because this is an asynchronous web server, there's nothing in the `loop()`.

```
void loop(){

}
```

Testing the ESP32 Server

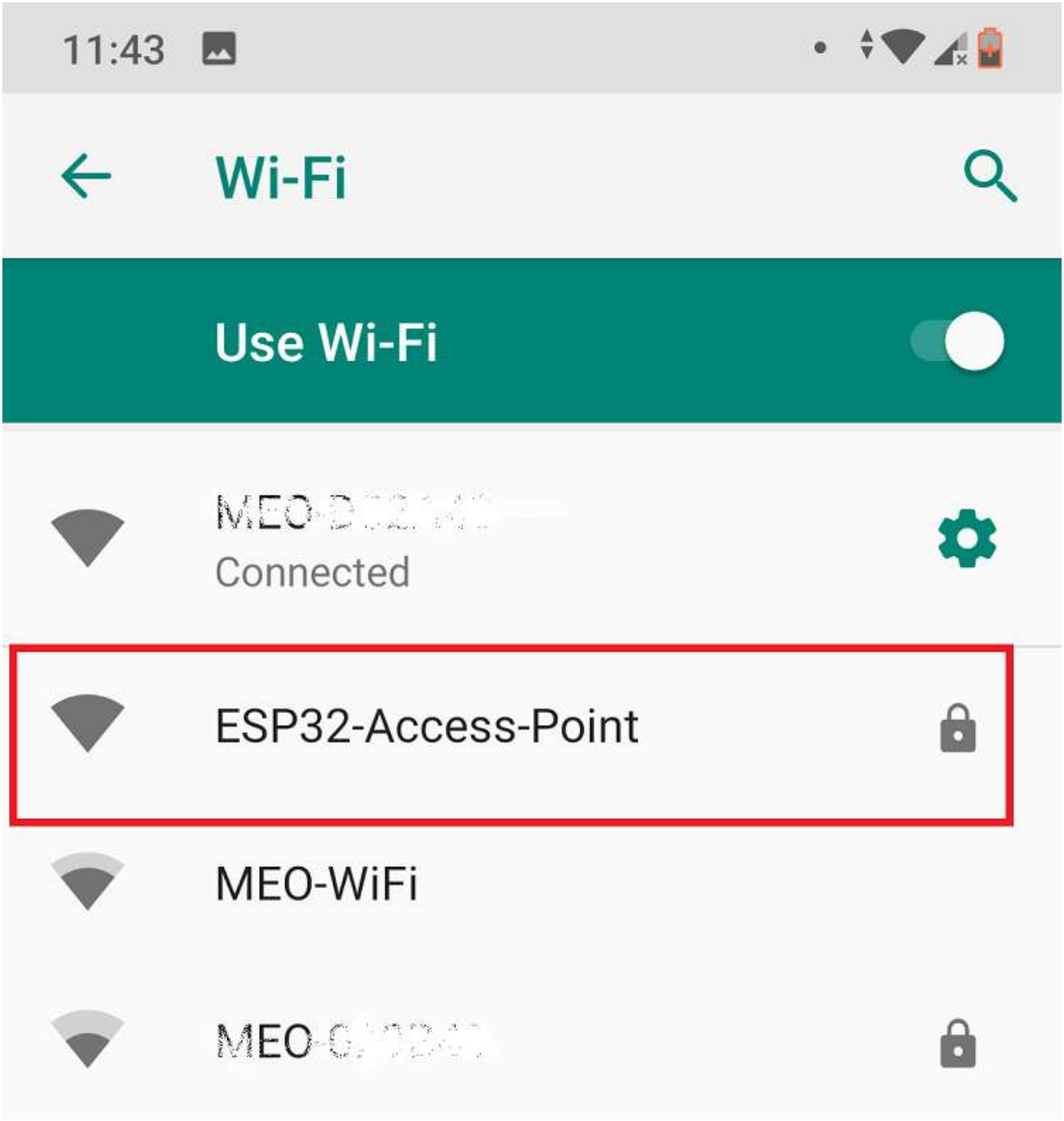
Upload the code to your board and open the Serial Monitor. You should get something as follows:



This means that the access point was set successfully.

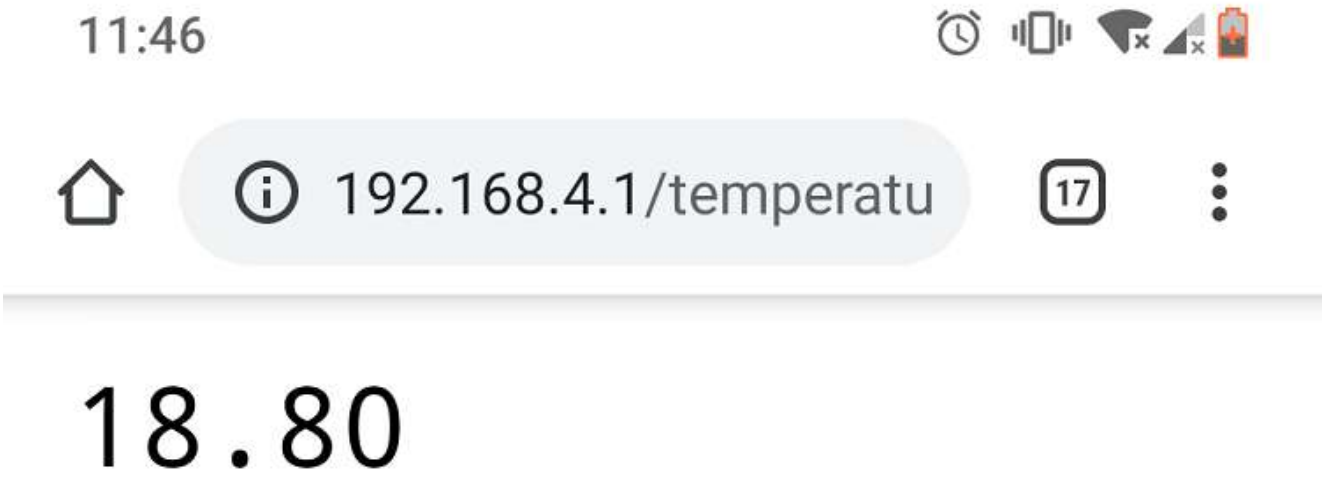
Now, to make sure it is listening for temperature, humidity and pressure requests, you need to connect to its network.

In your smartphone, go to the Wi-Fi settings and connect to the **ESP32-Access-Point**. The password is **123456789**.

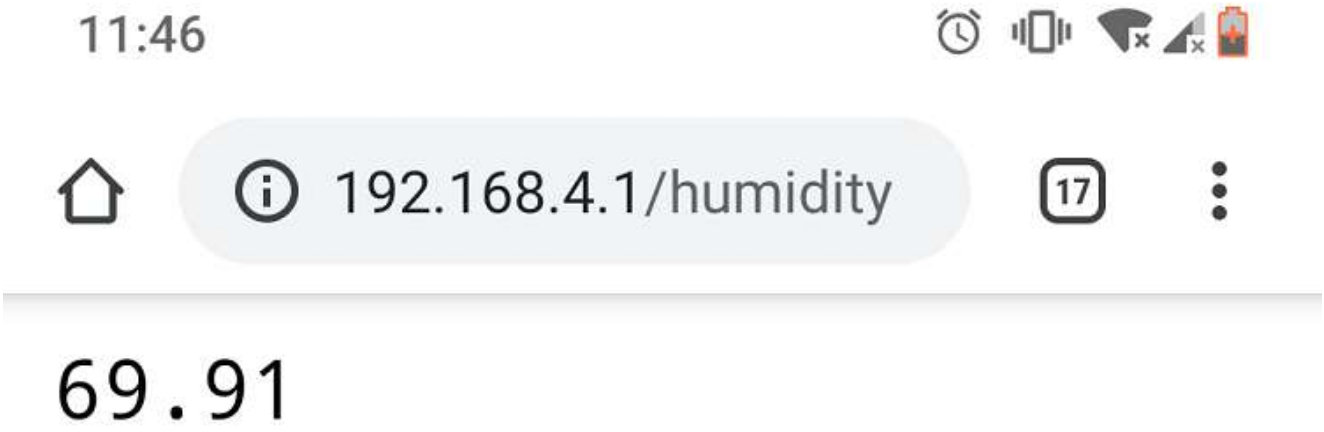


While connected to the access point, open your browser and type `192.168.4.1/temperature`

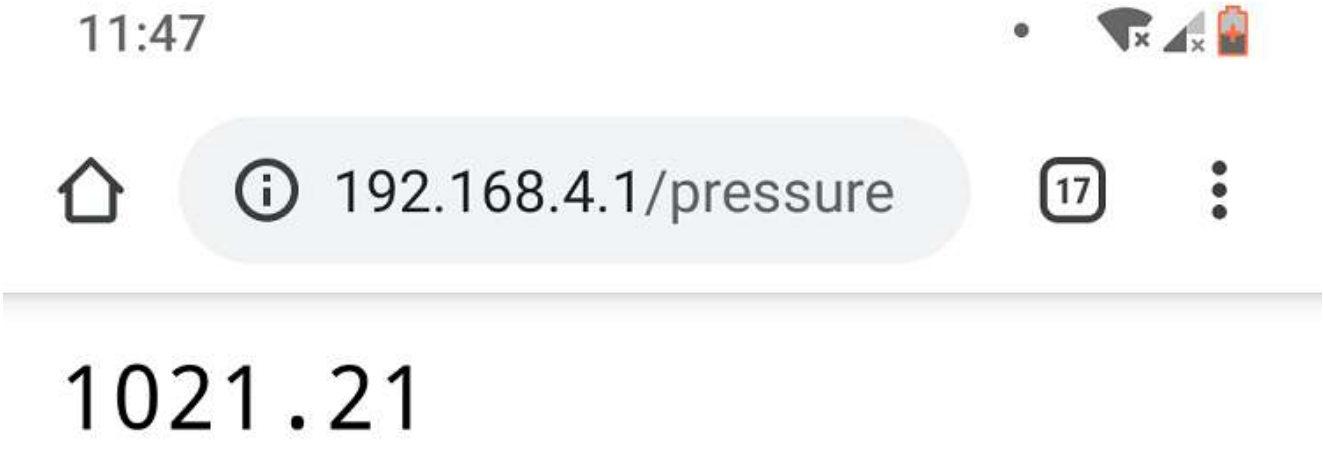
You should get the temperature value in your browser:



Try this URL path for the humidity `192.168.4.1/humidity` :

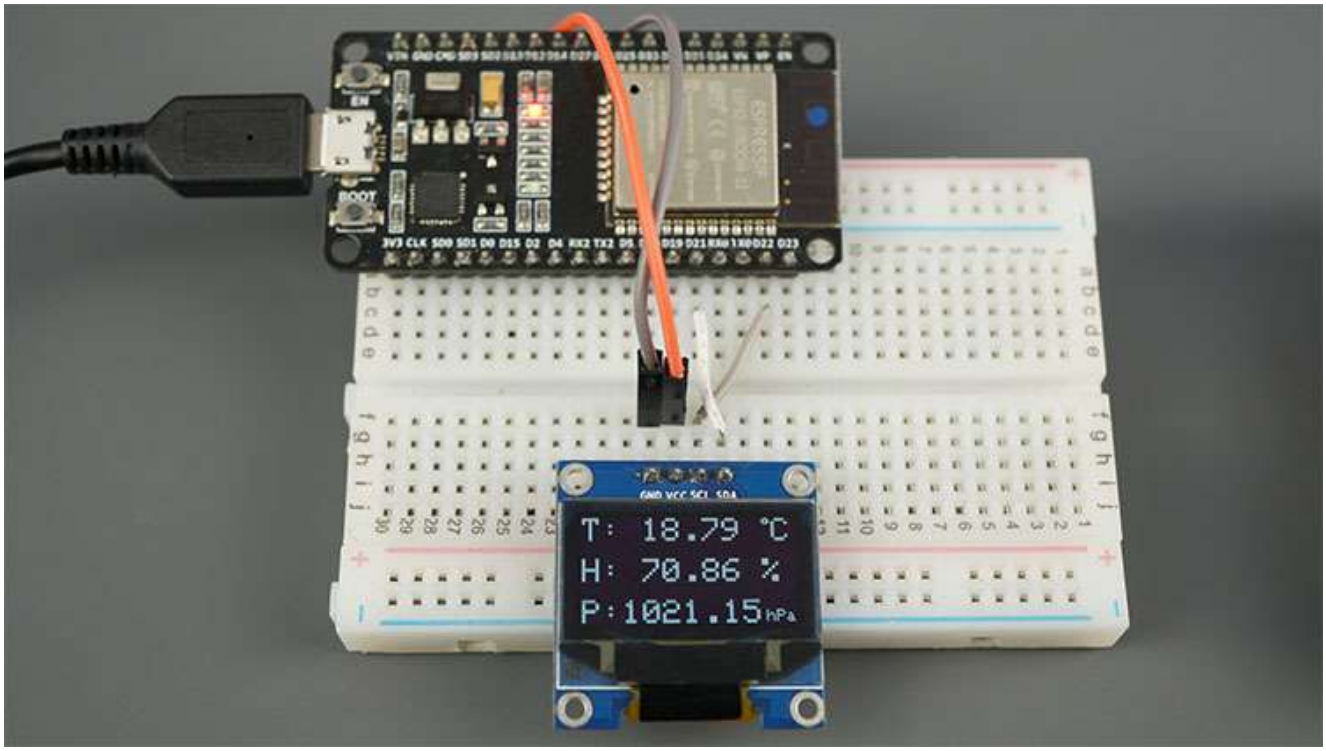


Finally, go to `192.168.4.1/pressure` URL:



If you're getting valid readings, it means that everything is working properly. Now, you need to prepare the other ESP32 board (client) to make those requests for you and display them on the OLED display.

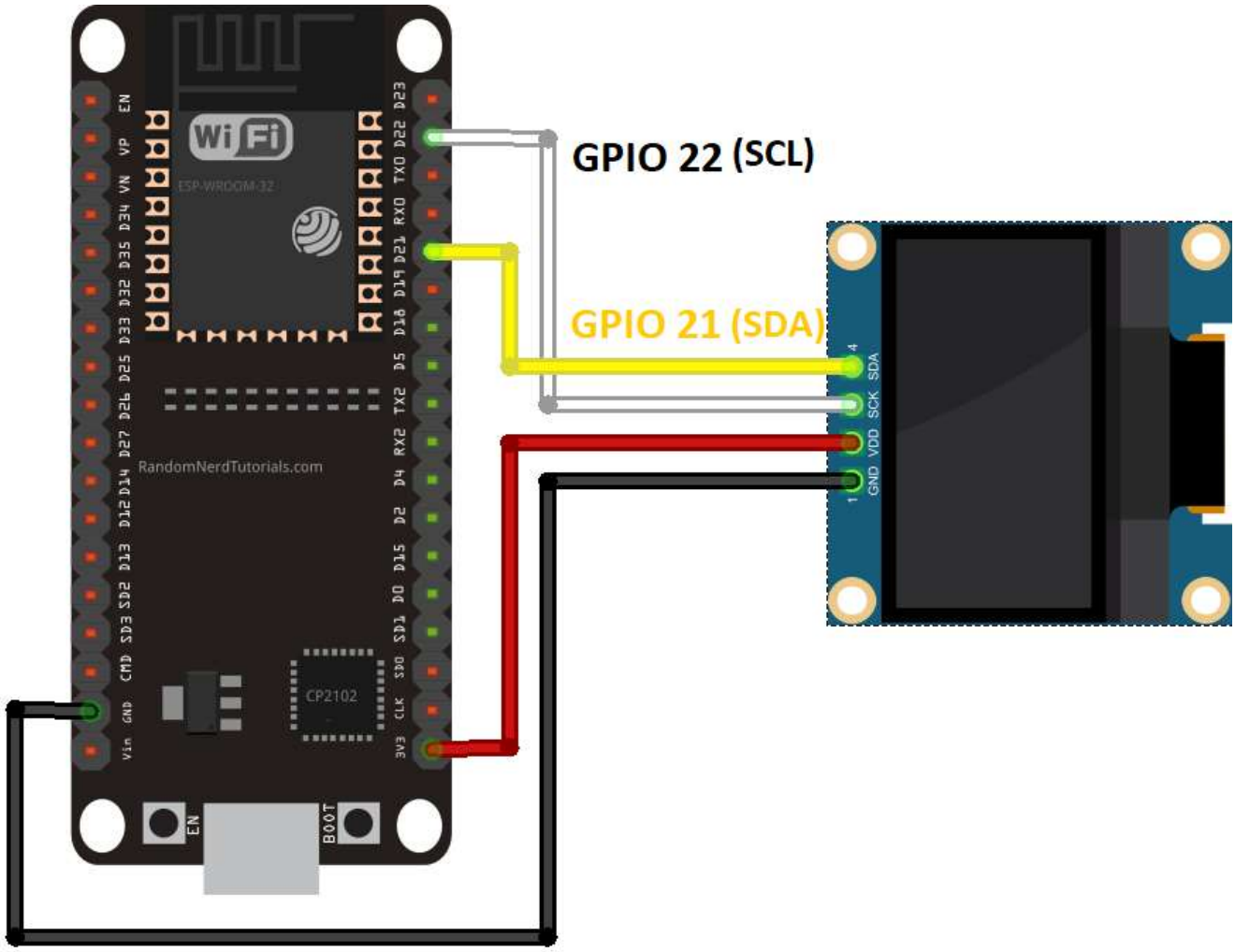
#2 ESP32 Client (Station)



The ESP32 Client is a Wi-Fi station that is connected to the ESP32 Server. The client requests the temperature, humidity and pressure from the server by making HTTP GET requests on the `/temperature`, `/humidity`, and `/pressure` URL routes. Then, it displays the readings on an OLED display.

Schematic Diagram

Wire the ESP32 to the OLED display as shown in the following schematic diagram.



OLED	ESP32
VIN/VCC	VIN
GND	GND
SCL	GPIO 22
SDA	GPIO 21

Arduino Sketch for #2 ESP32 Client

Upload the following code to the other ESP32:

```
/*
  Rui Santos
  Complete project details at https://RandomNerdTutorials.com/e
```

```
  Permission is hereby granted, free of charge, to any person o
  of this software and associated documentation files.

  The above copyright notice and this permission notice shall b
  copies or substantial portions of the Software.
*/

#include <WiFi.h>
#include <HTTPClient.h>

const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";

//Your IP address or domain name with URL path
const char* serverNameTemp = "http://192.168.4.1/temperature";
const char* serverNameHumi = "http://192.168.4.1/humidity";
const char* serverNamePres = "http://192.168.4.1/pressure";

#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define SCREEN_WIDTH 128 // OLED display width, in pixels
```

View raw code

How the code works

Include the necessary libraries for the Wi-Fi connection and for making HTTP requests:

```
#include <WiFi.h>
#include <HTTPClient.h>
```

Insert the ESP32 server network credentials. If you've changed the default network credentials in the ESP32 server, you should change them here to match.

```
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";
```

Then, save the URLs where the client will be making HTTP requests. The ESP32 server has the 192.168.4.1 IP address, and we'll be making requests on the `/temperature`, `/humidity` and `/pressure` URLs.

```
const char* serverNameTemp = "http://192.168.4.1/temperature";
const char* serverNameHumi = "http://192.168.4.1/humidity";
const char* serverNamePres = "http://192.168.4.1/pressure";
```

Include the libraries to interface with the OLED display:

```
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
```

Set the OLED display size:

```
#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels
```

Create a `display` object with the size you've defined earlier and with I2C communication protocol.

```
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED
```

Initialize string variables that will hold the temperature, humidity and pressure readings retrieved by the server.

```
String temperature;
String humidity;
String pressure;
```

Set the time interval between each request. By default, it's set to 5 seconds, but you can change it to any other interval.

```
const long interval = 5000;
```

In the `setup()`, initialize the OLED display:

```
// Address 0x3C for 128x64, you might need to change this value (
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
  Serial.println(F("SSD1306 allocation failed"));
  for(;;); // Don't proceed, loop forever
}
display.clearDisplay();
display.setTextColor(WHITE);
```

Note: if your OLED display is not working, check its I2C address using an I2C scanner sketch and change the code accordingly.

Connect the ESP32 client to the ESP32 server network.

```
WiFi.begin(ssid, password);
Serial.println("Connecting");
while(WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
}
Serial.println("");
Serial.print("Connected to WiFi network with IP Address: ");
```

In the `loop()` is where we make the HTTP GET requests. We've created a function called `httpGETRequest()` that accepts as argument the URL path where we want to make the request and returns the response as a `String`.

You can use the next function in your projects to simplify your code:

```
String httpGETRequest(const char* serverName) {
  HTTPClient http;

  // Your IP address with path or Domain name with URL path
  http.begin(serverName);

  // Send HTTP POST request
  int httpResponseCode = http.GET();

  String payload = "--";

  if (httpResponseCode>0) {
    Serial.print("HTTP Response code: ");
    Serial.println(httpResponseCode);
    payload = http.getString();
  }
  else {
    Serial.print("Error code: ");
    Serial.println(httpResponseCode);
  }
  // Free resources
  http.end();

  return payload;
}
```

We use that function to get the temperature, humidity and pressure readings from the server.

```
temperature = httpGETRequest(serverNameTemp);
humidity = httpGETRequest(serverNameHumi);
```

```
  pressure = httpGETRequest(serverNamePres);
```

Print those readings in the Serial Monitor for debugging.

```
  Serial.println("Temperature: " + temperature + " *C - Humidity: "
```

Then, display the temperature in the OLED display:

```
  display.setTextSize(2);
  display.setTextColor(WHITE);
  display.setCursor(0,0);
  display.print("T: ");
  display.print(temperature);
  display.print(" ");
  display.setTextSize(1);
  display.cp437(true);
  display.write(248);
  display.setTextSize(2);
  display.print("C");
```

The humidity:

```
  display.setTextSize(2);
  display.setCursor(0, 25);
  display.print("H: ");
  display.print(humidity);
  display.print(" %");
```

Finally, the pressure reading:

```
  display.setTextSize(2);
  display.setCursor(0, 50);
  display.print("P:");
```

```
  display.print(pressure);
  display.setTextSize(1);
  display.setCursor(110, 56);
  display.print("hPa");

  display.display();
```

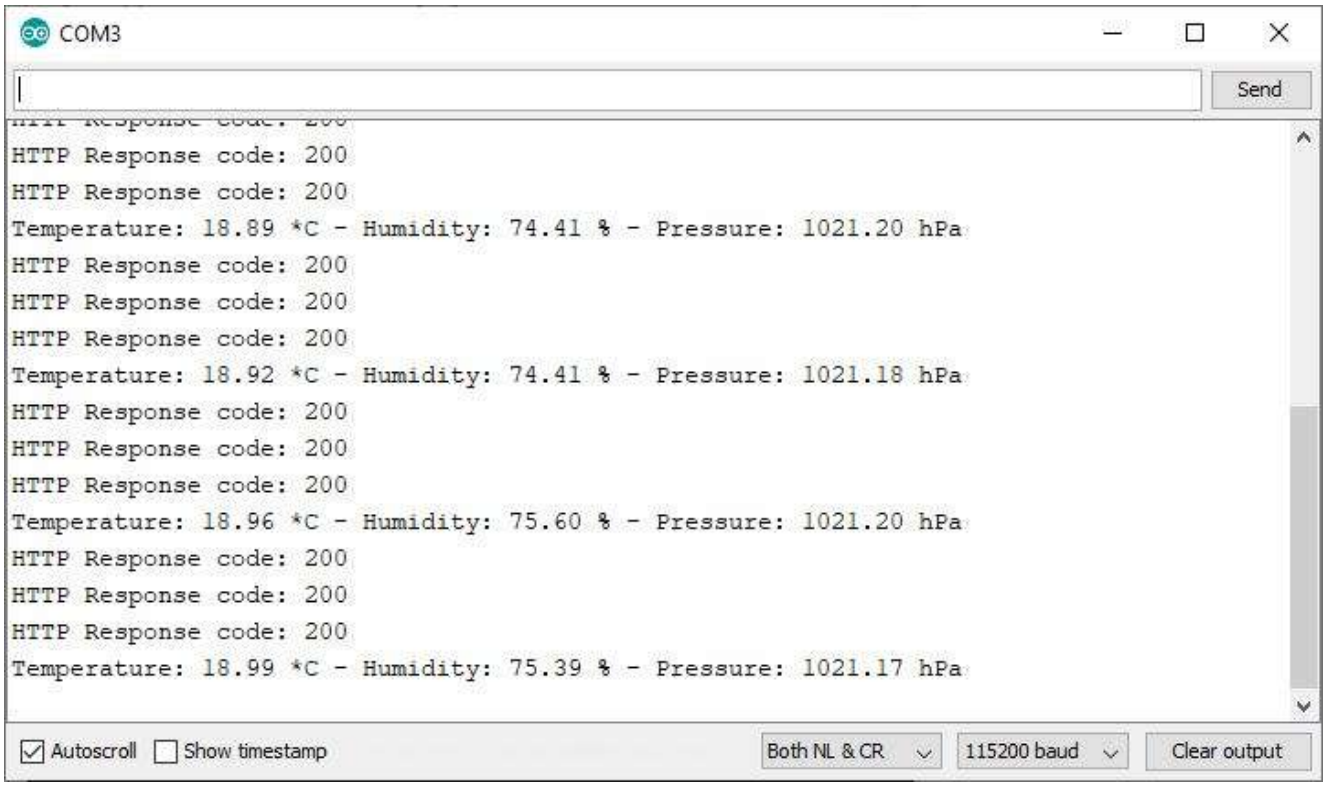
We use timers instead of delays to make a request every x number of seconds. That's why we have the `previousMillis`, `currentMillis` variables and use the `millis()` function. We have an article that shows the difference between timers and delays that you might find useful (or read ESP32 Timers).

Upload the sketch to #2 ESP32 (client) to test if everything is working properly.

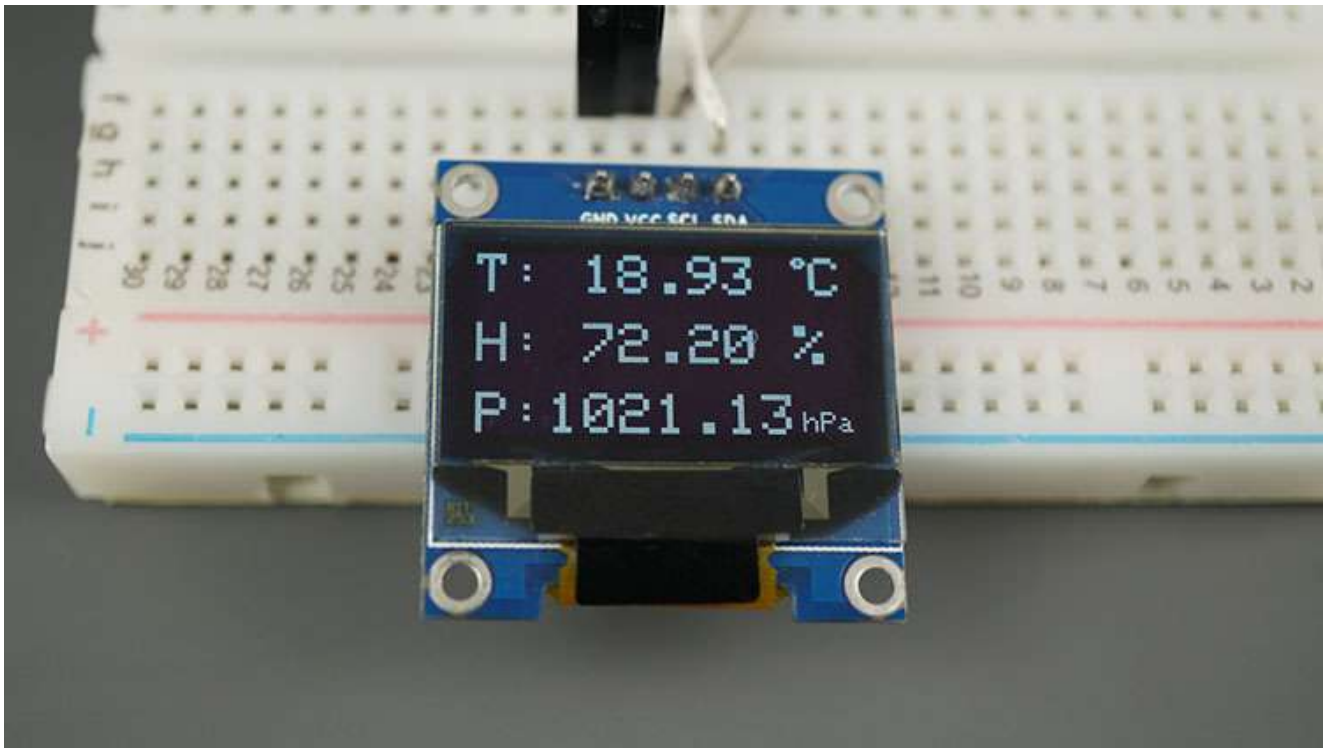
Testing the ESP32 Client

Having both boards fairly close and powered, you'll see that ESP #2 is receiving new temperature, humidity and pressure readings every 5 seconds from ESP #1.

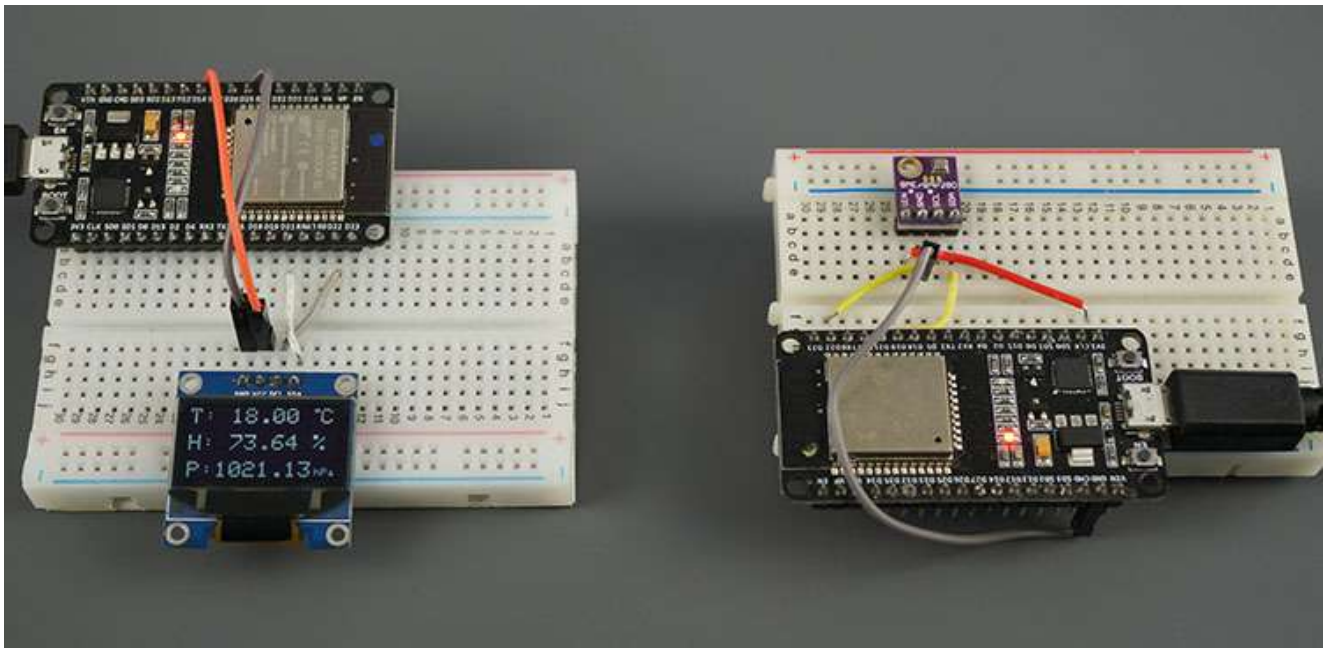
This is what you should see on the ESP32 Client Serial Monitor.



The sensor readings are also displayed in the OLED.



That's it! Your two boards are talking with each other.



Wrapping Up

In this tutorial you've learned how to send data from one ESP32 to another ESP32 board via Wi-Fi using HTTP requests without the need to connect to the internet. For demonstration purposes, we've shown how to send BME280 sensor readings, but you can use any other sensor or send any other data. Other recommended sensors:

- ESP32 DHT11 or DHT22 (Guide)

Getting Started with ESP-NOW (ESP32 with Arduino IDE)

Learn how to use ESP-NOW to exchange data between ESP32 boards programmed with Arduino IDE. ESP-NOW is a connectionless communication protocol developed by Espressif that features short packet transmission. This protocol enables multiple devices to talk to each other in an easy way.

We have other tutorials for ESP-NOW with the ESP32:

- ESP-NOW Two-Way Communication Between ESP32 Boards
- ESP-NOW with ESP32: Send Data to Multiple Boards (one-to-many)
- ESP-NOW with ESP32: Receive Data from Multiple Boards (many-to-one)
- ESP32: ESP-NOW Web Server Sensor Dashboard (ESP-NOW + Wi-Fi)

Arduino IDE

We'll program the ESP32 board using Arduino IDE, so before proceeding with this tutorial you should have the ESP32 add-on installed in your Arduino IDE. Follow

the next guide:

- Installing the ESP32 Board in Arduino IDE (Windows, Mac OS X, and Linux instructions)

Note: we have a similar guide for the ESP8266 NodeMCU Board: Getting Started with ESP-NOW (ESP8266 NodeMCU with Arduino IDE)

Introducing ESP-NOW

For a video introduction to ESP-NOW protocol, watch the following (try the project featured in this video):

Stating the Espressif website, ESP-NOW is a "*protocol developed by Espressif, which enables multiple devices to communicate with one another without using Wi-Fi. The protocol is similar to the low-power 2.4GHz wireless connectivity (…) . The pairing between devices is needed prior to their communication. After the pairing is done, the connection is safe and peer-to-peer, with no handshake being required*."

This means that after pairing a device with each other, the connection is persistent. In other words, if suddenly one of your boards loses power or resets, when it restarts, it will automatically connect to its peer to continue the communication.

ESP-NOW supports the following features:

- Encrypted and unencrypted unicast communication;
- Mixed encrypted and unencrypted peer devices;
- **Up to 250-byte** payload can be carried;
- Sending callback function that can be set to inform the application layer of transmission success or failure.

ESP-NOW technology also has the following limitations:

- Limited encrypted peers. 10 encrypted peers at the most are supported in Station mode; 6 at the most in SoftAP or SoftAP + Station mode;
- Multiple unencrypted peers are supported, however, their total number should be less than 20, including encrypted peers;
- **Payload is limited to 250 bytes**.

In simple words, ESP-NOW is a fast communication protocol that can be used to exchange small messages (up to 250 bytes) between ESP32 boards.

ESP-NOW is very versatile and you can have one-way or two-way communication in different setups.

ESP-NOW One-Way Communication

For example, in one-way communication, you can have scenarios like this:

- **One ESP32 board sending data to another ESP32 board**

This configuration is very easy to implement and it is great to send data from one board to the other like sensor readings or ON and OFF commands to control

GPIOs.

- **A “master” ESP32 sending data to multiple ESP32 “slaves”**

One ESP32 board sending the same or different commands to different ESP32 boards. This configuration is ideal to build something like a remote control. You can have several ESP32 boards around the house that are controlled by one main ESP32 board.

- **One ESP32 “slave” receiving data from multiple “masters”**

This configuration is ideal if you want to collect data from several sensors nodes into one ESP32 board. This can be configured as a web server to display data from all the other boards, for example.

Note: in the ESP-NOW documentation there isn’t such thing as “sender/master” and “receiver/slave”. Every board can be a sender or receiver. However, to keep things clear we’ll use the terms “sender” and “receiver” or “master” and “slave”.

ESP-NOW Two-Way Communication

With ESP-NOW, each board can be a sender and a receiver at the same time. So, you can establish two-way communication between boards.

For example, you can have two boards communicating with each other.

Learn how to: Exchange Sensor Readings with ESP-NOW Two-Way Communication.

You can add more boards to this configuration and have something that looks like a network (all ESP32 boards communicate with each other).

In summary, ESP-NOW is ideal to build a network in which you can have several ESP32 boards exchanging data with each other.

ESP32: Getting Board MAC Address

To communicate via ESP-NOW, you need to know the MAC Address of the ESP32 receiver. That's how you know to which device you'll send the data to.

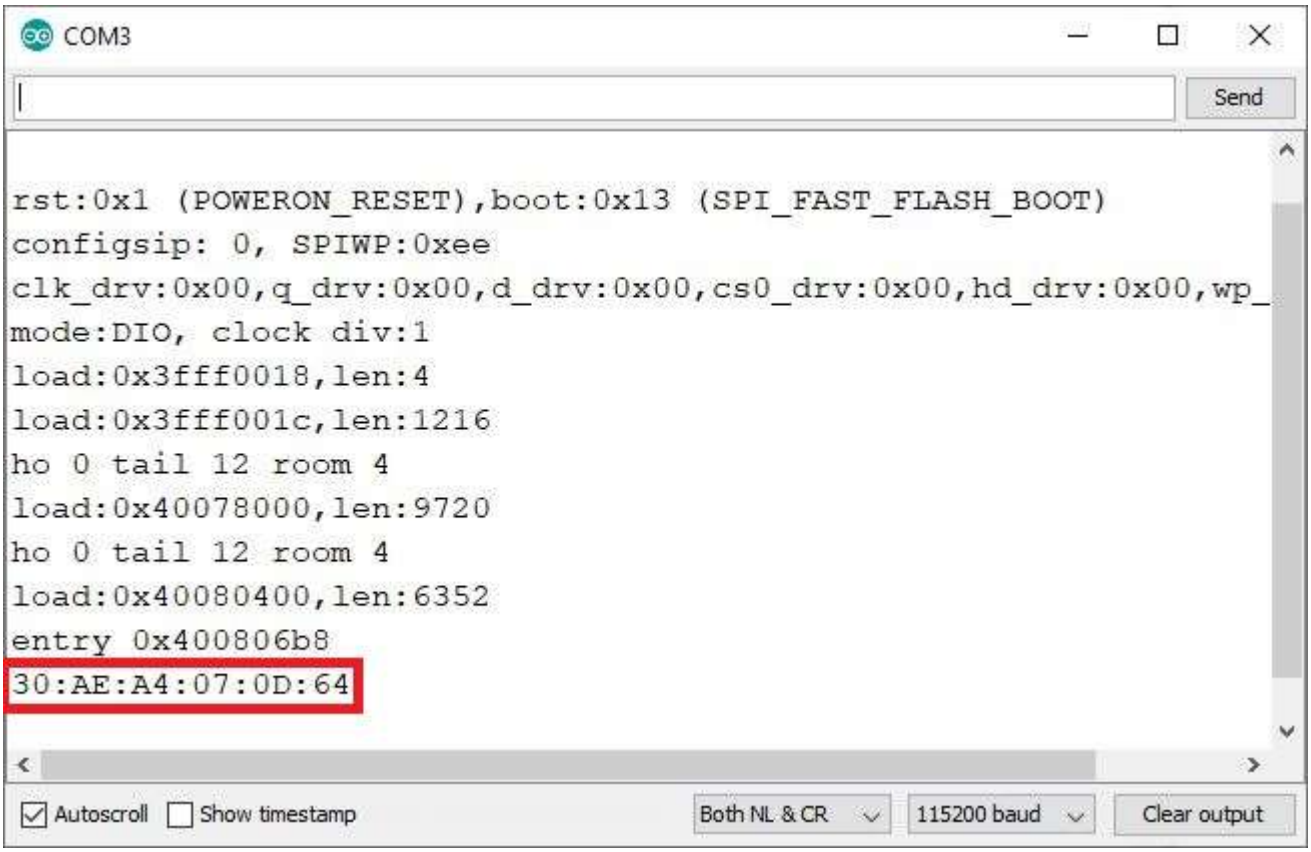
Each ESP32 has a unique MAC Address and that's how we identify each board to send data to it using ESP-NOW (learn how to Get and Change the ESP32 MAC Address).

To get your board's MAC Address, upload the following code.

```
// Complete Instructions to Get and Change ESP MAC Address: https

#include "WiFi.h"

void setup(){
  Serial.begin(115200);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println(WiFi.macAddress());
}

void loop(){

}
```

View raw code

After uploading the code, open the Serial Monitor at a baud rate of 115200 and press the ESP32 RST/EN button. The MAC address should be printed as follows:



Save your board MAC address because you'll need it to send data to the right board via ESP-NOW.

ESP-NOW One-way Point to Point Communication

To get you started with ESP-NOW wireless communication, we'll build a simple project that shows how to send a message from one ESP32 to another. One ESP32 will be the "sender" and the other ESP32 will be the "receiver".

We'll send a structure that contains a variable of type *char*, *int*, *float*, and *boolean*. Then, you can modify the structure to send whichever variable types are suitable for your project (like sensor readings, or boolean variables to turn something on or off).

For better understanding, we'll call "sender" to ESP32 #1 and "receiver" to ESP32 #2.

Here's what we should include in the sender sketch:

1. Initialize ESP-NOW;
2. Register a callback function upon sending data – the `OnDataSent` function will be executed when a message is sent. This can tell us if the message was successfully delivered or not;
3. Add a peer device (the receiver). For this, you need to know the receiver MAC address;
4. Send a message to the peer device.

On the receiver side, the sketch should include:

1. Initialize ESP-NOW;
2. Register for a receive callback function (`OnDataRecv`). This is a function that will be executed when a message is received.

3. Inside that callback function, save the message into a variable to execute any task with that information.

ESP-NOW works with callback functions that are called when a device receives a message or when a message is sent (you get if the message was successfully delivered or if it failed).

ESP-NOW Useful Functions

Here's a summary of the most essential ESP-NOW functions:

Function Name and Description
`esp_now_init()` Initializes ESP-NOW. You must initialize Wi-Fi before initializing ESP-NOW.
`esp_now_add_peer()` Call this function to pair a device and pass as an argument the peer MAC address.
`esp_now_send()` Send data with ESP-NOW.
`esp_now_register_send_cb()` Register a callback function that is triggered upon sending data. When a message is sent, a function is called – this function returns whether the delivery was successful or not.
`esp_now_register_rcv_cb()` Register a callback function that is triggered upon receiving data. When data is received via ESP-NOW, a function is called.

For more information about these functions read the ESP-NOW documentation at Read the Docs.

ESP32 Sender Sketch (ESP-NOW)

Here's the code for the ESP32 Sender board. Copy the code to your Arduino IDE, but don't upload it yet. You need to make a few modifications to make it work for you.

```
/*
  Rui Santos
  Complete project details at https://RandomNerdTutorials.com/e

  Permission is hereby granted, free of charge, to any person o
  of this software and associated documentation files.

  The above copyright notice and this permission notice shall b
  copies or substantial portions of the Software.
*/

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

// REPLACE WITH YOUR RECEIVER MAC Address
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF

// Structure example to send data
// Must match the receiver structure
typedef struct struct_message {
  char a[32];
  int b;
  float c;
  bool d;
} struct_message;

// Create a struct_message called myData
```

View raw code

How the code works

First, include the `esp_now.h` and `WiFi.h` libraries.

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

In the next line, you should insert the ESP32 receiver MAC address.

```
uint8_t broadcastAddress[] = {0x30, 0xAE, 0xA4, 0x07, 0x0D, 0x64}
```

In our case, the receiver MAC address is: `30:AE:A4:07:0D:64`, but you need to replace that variable with your own MAC address.

Then, create a structure that contains the type of data we want to send. We called this structure `struct_message` and it contains 4 different variable types. You can change this to send other variable types.

```
typedef struct struct_message {
  char a[32];
  int b;
  float c;
  bool d;
} struct_message;
```

Then, create a new variable of type `struct_message` that is called `myData` that will store the variables' values.

```
struct_message myData;
```

Create a variable of type `esp_now_peer_info_t` to store information about the peer.

```
esp_now_peer_info_t peerInfo;
```

Next, define the `OnDataSent()` function. This is a callback function that will be executed when a message is sent. In this case, this function simply prints if the message was successfully delivered or not.

```
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t st
  Serial.print("\r\nLast Packet Send Status:\t");
  Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Succe
}
```

In the `setup()`, initialize the serial monitor for debugging purposes:

```
Serial.begin(115200);
```

Set the device as a Wi-Fi station:

```
WiFi.mode(WIFI_STA);
```

Initialize ESP-NOW:

```
if (esp_now_init() != ESP_OK) {
  Serial.println("Error initializing ESP-NOW");
  return;
}
```

After successfully initializing ESP-NOW, register the callback function that will be called when a message is sent. In this case, we register for the `OnDataSent()` function created previously.

```
esp_now_register_send_cb(OnDataSent);
```

After that, we need to pair with another ESP-NOW device to send data. That's what we do in the next lines:

```
//Register peer
memcpy(peerInfo.peer_addr, broadcastAddress, 6);
```

```
  peerInfo.channel = 0;
  peerInfo.encrypt = false;

  //Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }
```

In the `loop()`, we'll send a message via ESP-NOW every 2 seconds (you can change this delay time).

First, we set the variables values as follows:

```
  strcpy(myData.a, "THIS IS A CHAR");
  myData.b = random(1,20);
  myData.c = 1.2;
  myData.d = false;
```

Remember that `myData` is a structure. Here we assign the values we want to send inside the structure. For example, the first line assigns a char, the second line assigns a random Int number, a Float, and a Boolean variable.

We create this kind of structure to show you how to send the most common variable types. You can change the structure to send other data types.

Finally, send the message as follows:

```
  esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &my
```

Check if the message was successfully sent:

```
  if (result == ESP_OK) {
    Serial.println("Sent with success");
```

```
  }
  else {
    Serial.println("Error sending the data");
  }
```

The `loop()` is executed every 2000 milliseconds (2 seconds).

```
  delay(2000);
```

ESP32 Receiver Sketch (ESP-NOW)

Upload the following code to your ESP32 receiver board.

```
/*
  Rui Santos
  Complete project details at https://RandomNerdTutorials.com/e

  Permission is hereby granted, free of charge, to any person o
  of this software and associated documentation files.

  The above copyright notice and this permission notice shall b
  copies or substantial portions of the Software.
*/

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

// Structure example to receive data
// Must match the sender structure
typedef struct struct_message {
    char a[32];
    int b;
    float c;
    bool d;
} struct_message;
```

```
// Create a struct_message called myData
struct_message myData;
```

View raw code

How the code works

Similarly to the sender, start by including the libraries:

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

Create a structure to receive the data. This structure should be the same defined in the sender sketch.

```
typedef struct struct_message {
  char a[32];
  int b;
  float c;
  bool d;
} struct_message;
```

Create a `struct_message` variable called `myData`.

```
struct_message myData;
```

Create a callback function that will be called when the ESP32 receives the data via ESP-NOW. The function is called `onDataRecv()` and should accept several parameters as follows:

```
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData,
```

We copy the content of the `incomingData` data variable into the `myData` variable.

```
memcpy(&myData, incomingData, sizeof(myData));
```

Now, the `myData` structure contains several variables inside with the values sent by the ESP32 sender. To access variable `a`, for example, we just need to call `myData.a`.

In this example, we simply print the received data, but in a practical application you can print the data on a display, for example.

```
Serial.print("Bytes received: ");
Serial.println(len);
Serial.print("Char: ");
Serial.println(myData.a);
Serial.print("Int: ");
Serial.println(myData.b);
Serial.print("Float: ");
Serial.println(myData.c);
Serial.print("Bool: ");
Serial.println(myData.d);
Serial.println();
```

In the `setup()`, intialize the Serial Monitor.

```
Serial.begin(115200);
```

Set the device as a Wi-Fi Station.

```
WiFi.mode(WIFI_STA);
```

Initialize ESP-NOW:

```
if (esp_now_init() != ESP_OK) {
  Serial.println("Error initializing ESP-NOW");
  return;
}
```

Register for a callback function that will be called when data is received. In this case, we register for the `OnDataRecv()` function that was created previously.

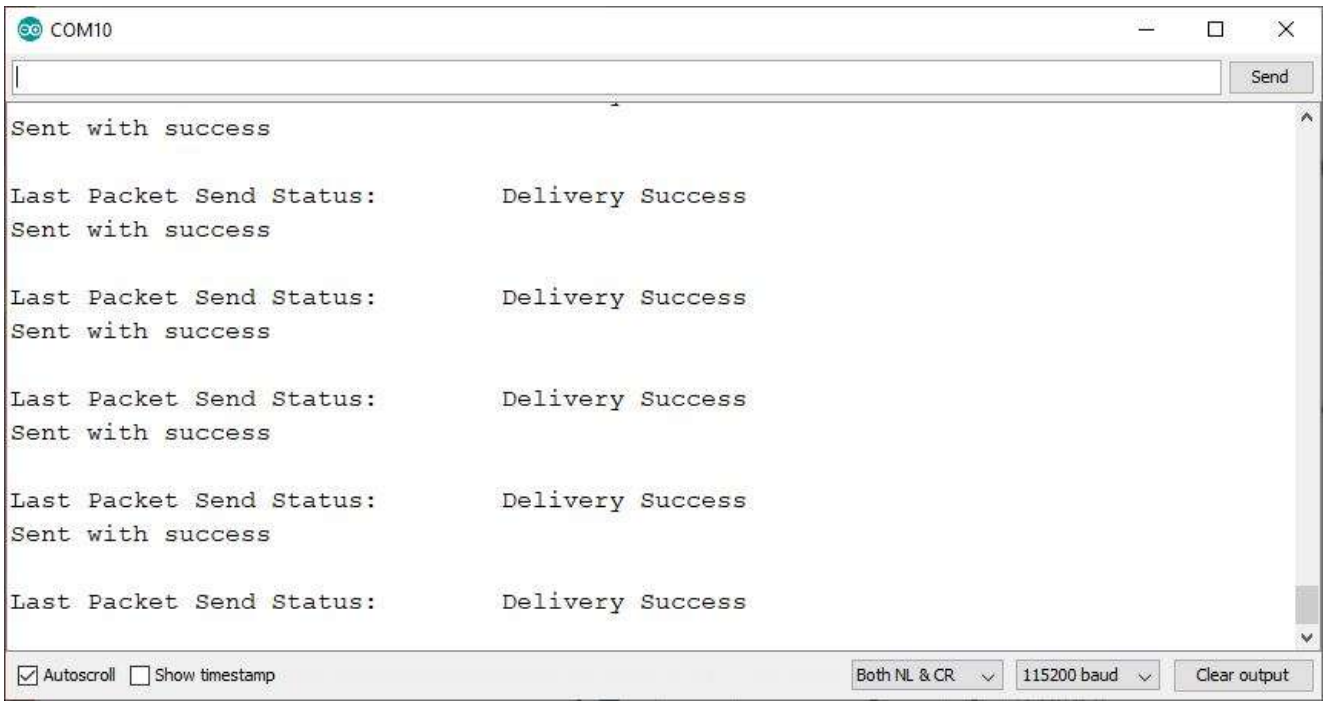
```
esp_now_register_recv_cb(OnDataRecv);
```

Testing ESP-NOW Communication

Upload the sender sketch to the sender ESP32 board and the receiver sketch to the receiver ESP32 board.

Now, open two Arduino IDE windows. One for the receiver, and another for the sender. Open the Serial Monitor for each board. It should be a different COM port for each board.

This is what you should get on the sender side.



And this is what you should get on the receiver side. Note that the Int variable changes between each reading received (because we set it to a random number on the sender side).

We tested the communication range between the two boards, and we are able to get a stable communication up to 220 meters (approximately 722 feet) in open field. In this experiment both ESP32 on-board antennas were pointing to each other.

Wrapping Up

We tried to keep our examples as simple as possible so that you better understand how everything works. There are more ESP-NOW-related functions that can be useful in your projects, like: managing peers, deleting peers, scanning for slave devices, etc… For a complete example, in your Arduino IDE, you can go to **File** > **Examples** > **ESP32** > **ESPNow** and choose one of the example sketches.

We hope you've found this introduction to ESP-NOW useful. As a simple getting started example, we've shown you how to send data as a structure from one ESP32 to another. The idea is to replace the structure values with sensor readings or GPIO states, for example.

Additionally, with ESP-NOW, each board can simultaneously be a sender and receiver. One board can send data to multiple boards and also receive data from multiple boards.

We also have a tutorial about ESP-NOW with the ESP8266: Getting Started with ESP-NOW (ESP8266 NodeMCU with Arduino IDE).

To learn more about the ESP32 board, make sure you take a look at our resources:

- ESP-NOW Two-Way Communication Between ESP32 Boards
- **Learn ESP32 with Arduino IDE (video course + eBook);**
- MicroPython Programming with ESP32 and ESP8266
- More ESP32 resources…

Thanks for reading.