

3rd Edition

LEARN ESP32 with Arduino IDE



The complete guide to program the ESP32 with Arduino IDE, including projects, tips, and tricks!

RUI SANTOS & SARA SANTOS
RANDOM NERD TUTORIALS

Learn ESP32 with Arduino IDE

3rd Edition (Version 3.0)

The complete guide to programming the ESP32 with Arduino IDE, including projects, tips, and tricks.

Rui Santos & Sara Santos

Random Nerd Tutorials

Security Notice

Sorry for writing this notice, but the evidence is clear: piracy for digital products is over the entire internet.

For that reason, we've taken certain steps to protect the intellectual property contained in this eBook.

This eBook contains hidden random strings of text that only apply to your specific eBook version that is unique to your email address. You won't see anything different since those strings are hidden in this PDF. We apologize for having to do that. It means if someone were to share this eBook, we know who did it, and we can take further legal consequences.

You cannot redistribute this eBook. This eBook is for personal use and is only available for purchase at:

- <https://randomnerdtutorials.com/courses>
- <https://rntlab.com/shop>

Please send an email to the author (Rui Santos - hello@ruisantos.me) if you find this eBook anywhere else.

We want to thank you for purchasing this eBook, and we hope you learn a lot and have fun with it!

Disclaimer

This eBook was written for information purposes only. Every effort was made to make this eBook as complete and accurate as possible. The purpose of this eBook is to educate. The authors (Rui Santos and Sara Santos) do not warrant that the information contained in this eBook is fully complete and shall not be responsible for any errors or omissions.

The authors (Rui Santos and Sara Santos) shall have neither liability nor responsibility to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by this eBook.

Throughout this eBook, you will find some links, and some of them are affiliate links. This means the authors (Rui Santos and Sara Santos) earn a small commission from each purchase with that link. Please understand that the authors have experience with all those products and recommend them because they are useful, not because of the small commissions. Please do not spend any money on products unless you feel you need them.

Other Helpful Links:

- [Ask questions in our Forum](#)
- [Join Private Facebook Group](#)
- [Terms and Conditions](#)

Join the Private Facebook Group

This eBook comes with the opportunity to join a private community of like-minded people. If you purchased this eBook, you can join our private Facebook Group today!

Inside the group, you can ask questions and create discussions about everything related to ESP32, ESP32-CAM, ESP8266, Raspberry Pi, Raspberry Pi Pico, Arduino, and much more...

See it for yourself!

1. Go to -> <https://randomnerdtutorials.com/fb>
2. Click the "Join Group" button
3. We'll approve your request within less than 24 hours



Random Nerd Tutorials Community



About the Authors

This eBook was developed and written by Rui Santos and Sara Santos. We both live in Porto, Portugal, and we have known each other since 2009. If you want to learn more about us, read our [about page](#).



Hi! I'm Rui Santos, the founder of the [Random Nerd Tutorials blog](#). I have a master's degree in Electrical and Computer Engineering from FEUP and I've been running the RNT blog for more than 10 years. I've written hundreds of tutorials covering the usage of different microcontrollers (ESP32, ESP8266, Raspberry Pi, Arduino, and more) on the Internet of Things and Home Automation fields. We also self-published about a [dozen eBooks](#) on these subjects, helping thousands of students, engineers, and hobbyists passionate about electronics all over the world.



Hi! I'm Sara Santos, and I work with Rui at Random Nerd Tutorials since 2015. I have a master's degree in Bioengineering from FEUP. I create, write and edit the tutorials and articles for the [RNT](#) and [Maker Advisor](#) blogs, and I've written several of the eBooks available on the RNT blog. I also help you by answering your questions on our private forum and on our blog's comments section. I love books, writing, cats, and a hot cup of tea. I also love travel and writing about our travel adventures on our [travel blog](#).

Table of Contents

MODULE 0

Introduction	11
---------------------------	-----------

Welcome to Learn ESP32 with Arduino IDE	12
-----------------------------------------------	----

MODULE 1

Getting Started with ESP32	18
-----------------------------------------	-----------

1.1 - Introducing the ESP32.....	19
----------------------------------	----

1.2 - Installing ESP32 in Arduino IDE.....	33
--------------------------------------------	----

1.3 - How to Use Your ESP32 Board with this Course.....	43
---------------------------------------------------------	----

MODULE 2

Exploring the ESP32 GPIOs	52
----------------------------------------	-----------

2.1 - Pinout Reference: ESP32 GPIOs Explained	53
-----------------------------------------------------	----

2.2 - ESP32-S3 Pinout	64
-----------------------------	----

2.3 - ESP32 Digital Inputs and Outputs.....	72
---------------------------------------------	----

2.4 - Capacitive Touch Pins	78
-----------------------------------	----

2.5 - ESP32 Pulse-Width Modulation (PWM)	86
------------------------------------------------	----

2.6 - Reading Analog Inputs.....	95
----------------------------------	----

2.7 - ESP32 with PIR Motion Sensor: Interrupts and Timers	101
-----------------------------------------------------------------	-----

2.8 - ESP32 Dual Core: Create Tasks	116
-------------------------------------------	-----

MODULE 3

Saving Data and Handling Files	125
---------------------------------------------	------------

3.1 - Save Data Permanently using the Preferences Library	126
-----------------------------------------------------------------	-----

3.2 - ESP32 LittleFS Filesystem.....	139
--------------------------------------	-----

3.3 - LittleFS: Save Variables' Values in a File	159
--------------------------------------------------------	-----

3.4 - Saving Data to a microSD Card	166
-------------------------------------------	-----

MODULE 4

ESP32 Deep Sleep	181
4.1 - ESP32 Deep Sleep Mode.....	182
4.2 - Deep Sleep – Timer Wake Up.....	187
4.3 - Deep Sleep with Touch Wake Up.....	193
4.4 - Deep Sleep External Wake Up.....	200

MODULE 5

Introducing Wi-Fi.....	215
5.1 - Introducing Wi-Fi	216
5.2 - Getting Started with HTTP Requests	229
5.3 - Making HTTP Requests (WorldTime API and ThingSpeak)	236
5.4 - ESP32 Client-Server Wi-Fi Communication Between Two Boards.....	250

MODULE 6

ESP32 Web Servers	269
6.1 - Web Server Introduction	270
6.2 - ESP32 Web Server: Control Outputs	279
6.3 - ESP32 Web Server: HTML and CSS Basics.....	290
6.4 - ESP32 Web Server: Authentication	305
6.5 - Accessing the ESP32 Web Server from Anywhere	311
6.6 - ESP32 Web Server – Display Sensor Readings	320
6.7 - Asynchronous Web Server: Temperature and Humidity Readings.....	334
6.8 - Asynchronous Web Server: Control Outputs	351
6.9 - Build an ESP32 Web Server using Files from Filesystem (LittleFS)	366

MODULE 7

ESP32 Bluetooth	378
7.1 - ESP32 Bluetooth Low Energy (BLE): Introduction	379

7.2 - Bluetooth Low Energy: BLE Server, Scanner, and Notify.....	387
7.3 - ESP32 BLE Server and Client (Part 1/2)	405
7.4 - ESP32 BLE Server and Client (Part 2/2)	419
7.5 - Bluetooth Classic.....	432
MODULE 8	
LoRa Technology with ESP32.....	447
8.1 - ESP32 with LoRa: Introduction.....	448
8.2 - ESP32 LoRa Sender and Receiver	457
8.3 - Further Reading about LoRa Gateways	478
8.4 - LoRa: Where to Go Next?.....	481
MODULE 9	
ESP32 with MQTT	482
9.1 - Introducing MQTT	483
9.2 - Installing Mosquitto MQTT Broker on a Raspberry Pi	488
9.3 - MQTT Project: MQTT Client ESP32#1	498
9.4 - MQTT Project : MQTT Client ESP32 #2	514
9.5 - Installing Node-RED and Node-RED Dashboard on a RPi	526
9.6 - Connect the ESP32 to Node-RED using MQTT	541
MODULE 10	
ESP-NOW Communication Protocol	560
10.1 – ESP-NOW: Getting Started	561
10.2 - ESP-NOW Two-Way Communication Between ESP32	579
10.3 - ESP-NOW Send Data to Multiple Boards (one-to-many).....	593
10.4 - ESP-NOW Receive Data from Multiple Boards (many-to-one).....	607
10.5 - ESP-NOW Web Server Sensor Dashboard (ESP-NOW + Wi-Fi).....	619

PROJECT 1

ESP32 Wi-Fi Multisensor: Temperature, Humidity, Motion, Luminosity, and Relay Control.....645

Unit 1 - ESP32 Wi-Fi Multisensor	646
Unit 2 - ESP32 Wi-Fi Multisensor: How the Code Works?	674

PROJECT 2

Remote Controlled Wi-Fi Car Robot686

Unit 1 - Remote Controlled Wi-Fi Car Robot (Part 1/2).....	687
Unit 2 - Remote Controlled Wi-Fi Car Robot (Part 2/2).....	698
Unit 3 - Assembling the Smart Robot Car Chassis Kit	711
Unit 4 - Access Point (AP) For Wi-Fi Car Robot	717

PROJECT 3

ESP32 BLE Android Application.....724

Unit 1 - ESP32 BLE Android Application: Control Outputs and Display Sensor Readings	725
Unit 2 - Bluetooth Low Energy (BLE) Android Application with MIT App Inventor 2: How the App Works?	739

PROJECT 4

LoRa Long Range Sensor Monitoring and Data Logging.....754

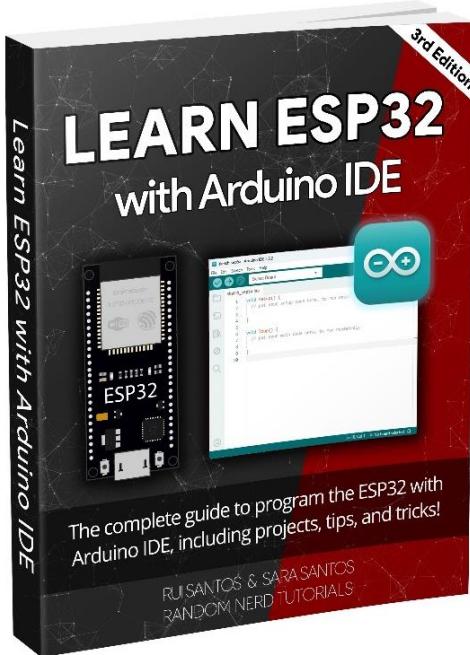
Unit 1 - LoRa Long Range Sensor Monitoring and Data Logging.....	755
Unit 2 - ESP32 LoRa Sender	763
Unit 3 - ESP32 LoRa Receiver	779
Unit 4 - LoRa Sender Solar Powered.....	799
Unit 5 - Final Tests, Demonstration, and Data Analysis	811
List of Parts Required.....	824

MODULE 0

Introduction

Welcome to Learn ESP32 with Arduino IDE

Welcome to the **Learn ESP32 with Arduino IDE** eBook! This is a practical course where you'll learn to take the most out of the ESP32 using the Arduino IDE.



How to Follow this eBook?

The Modules in this course are independent, which means there isn't a specific order to follow along, and you can pick any module you feel like reading at any time. However, **you MUST follow Module 1 first** to set up the ESP32 in your Arduino IDE correctly. Otherwise, the examples in the other Modules won't work.

Download Source Code and Resources



Each Unit contains the source code, schematics, and resources you need to follow the projects. You can download each resource at the Unit page or download the [Learn ESP32 with Arduino IDE GitHub repository](#) and instantly download all the resources for this course.

Important: don't copy the code from the PDF because the formatting might get messed up. Additionally, due to library updates or updates of the ESP32 core, we often need to modify the original codes. The codes from the GitHub links are always the most up-to-date and might not correspond exactly to what is in the PDF.

eBook Overview

This course contains 10 Modules and 4 Projects. Scroll down to take a look at the Modules and Projects covered in the course.

Module 1: Getting Started with ESP32

This first Module is an introduction to the ESP32 board. We'll explore its features and show you how to use your board with this course. You'll also prepare your Arduino IDE to upload code to the ESP32. You must follow this Module first before proceeding to any other project/example in the eBook.

Module 2: Exploring the ESP32 GPIO Pins

In this second Module, we'll explore the ESP32 GPIO functions. We'll show you how to control digital outputs, output PWM signals, and read digital and analog inputs. We'll also take a look at the ESP32 touch-capacitive pins and the ESP32 dual-core feature.

Module 3: Saving Data and Handling Files

Learn how to handle files on the ESP32 and how to save data permanently. We'll cover saving data on Preferences (NVS memory), on the ESP32 filesystem (LittleFS), and on a microSD card.

Module 4: ESP32 Deep Sleep Mode

Using deep sleep with your ESP32 is a great way to save power in battery-powered applications. This module will show you how to put your ESP32 into deep sleep mode and the different ways to wake it up.

Module 5: Introducing Wi-Fi

Get started with Wi-Fi on the ESP32. We'll cover basic concepts about Wi-Fi networks, you'll learn how to set the ESP32 as an access point and as a wi-fi station and how to make HTTP requests. We'll also cover how to set an ESP32 board as a Wi-Fi client and another one as a Wi-Fi server to exchange data between them.

Module 6: ESP32 Web Servers

This Module explains how to build several web servers with the ESP32. After explaining some theoretical concepts, you'll learn how to make a web server to display sensor readings, control outputs, and much more. You'll also learn how you can edit your web server interface using HTML and CSS.

Module 7: ESP32 Bluetooth Low Energy and Bluetooth Classic

The ESP32 comes not only with Wi-Fi but also Bluetooth and Bluetooth Low Energy built-in. Learn how to use the ESP32 Bluetooth functionalities to scan nearby devices and exchange information (BLE client and server).

Module 8: LoRa Technology with ESP32

LoRa is a long-range wireless technology. In this Module, you'll explore LoRa and how you can use it with the ESP32 to extend the communication range between IoT devices.

Module 9: ESP32 with MQTT

MQTT stands for Message Queuing Telemetry Transport. It is a lightweight publish and subscribe system perfect for Internet of Things applications. In this Module, you'll learn how to use MQTT to establish a communication between two ESP32 boards and how you can control the ESP32 using Node-RED (an IoT platform).

Module 10: ESP-NOW Communication Protocol

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 easily.

Project 1: ESP32 Wi-Fi Multisensor – Temperature, Humidity, Motion, Luminosity, and Relay Control

In this project, you'll build an ESP32 Wi-Fi Multisensor. This device consists of a PIR motion sensor, a light-dependent resistor (LDR), a DHT22 temperature and humidity sensor, a relay, and a status RGB LED. You'll also build a web server that allows you to control the ESP32 multisensor using different modes.

Project 2: Remote-Controlled Wi-Fi Car Robot

In this project, we'll show you how to create an ESP32 Wi-Fi remote-controlled car robot step by step.

Project 3: Bluetooth Low Energy (BLE) Android Application with MIT App Inventor – Control Outputs and Display Sensor Readings

In this project, you will create an Android application to interact with the ESP32 using Bluetooth Low Energy (BLE).

Project 4: LoRa Long Range Sensor Monitoring – Reporting Sensor Readings from Outside: Soil Moisture and Temperature

This project will build an off-the-grid monitoring system that sends soil moisture and temperature readings to an indoor receiver. To establish a communication between the sender and the receiver, we'll be using the LoRa communication protocol.

Getting Parts for the Course

To properly follow this course, you need some electronics components. In each Module, we provide a complete list of the parts required and links to [Maker Advisor](#), so that you can find the part you're looking for in your favorite store at the best price.



If you buy your parts through Maker Advisor links, we'll earn a small affiliate commission (you won't pay more for it). By getting your parts through our affiliate links, you are supporting our work. If there's a component or tool you're looking for, we advise you to look at [our favorite tools and parts here](#).

Note: for the complete parts list, consult the [Appendix at the end of this eBook](#).

Problems and Difficulties Throughout the Course

As you go through the course, you'll likely encounter some sort of difficulty or technical problem. I highly encourage you to spend a bit of time trying to fix technical problems by yourself. Fixing technical problems yourself is a very good way to learn a new subject.

If you have done your best but you couldn't find the solution for your issue, you can always rely on the community to help you out. You can use the [Private Forum](#) (recommended), [Facebook group](#), or our [Support form](#).

Check the Errata

Before starting any project, please visit the eBook page at the following link to review the errata and identify any necessary corrections.

- <https://rntlab.com/module-1/learn-esp32-welcome/>

If there isn't an errata on that page, it means there's nothing wrong you need to worry about.

Leave Feedback

Your feedback is very important so that we can improve the eBook and our learning materials. Suggestions, rectifications, and your opinion are essential.

You can use the following channels to leave feedback:

- [Support form](#)
- [RNT Lab Forum](#)
- [Facebook group](#)

MODULE 1

Getting Started with ESP32

1.1 – Introducing the ESP32

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 a dual-core processor. If you're familiar with the ESP8266, the ESP32 is its successor, loaded with lots of new features.

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 range 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 who are already familiar with programming the Arduino board, will 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 (not covered in this eBook).

ESP32 Specifications

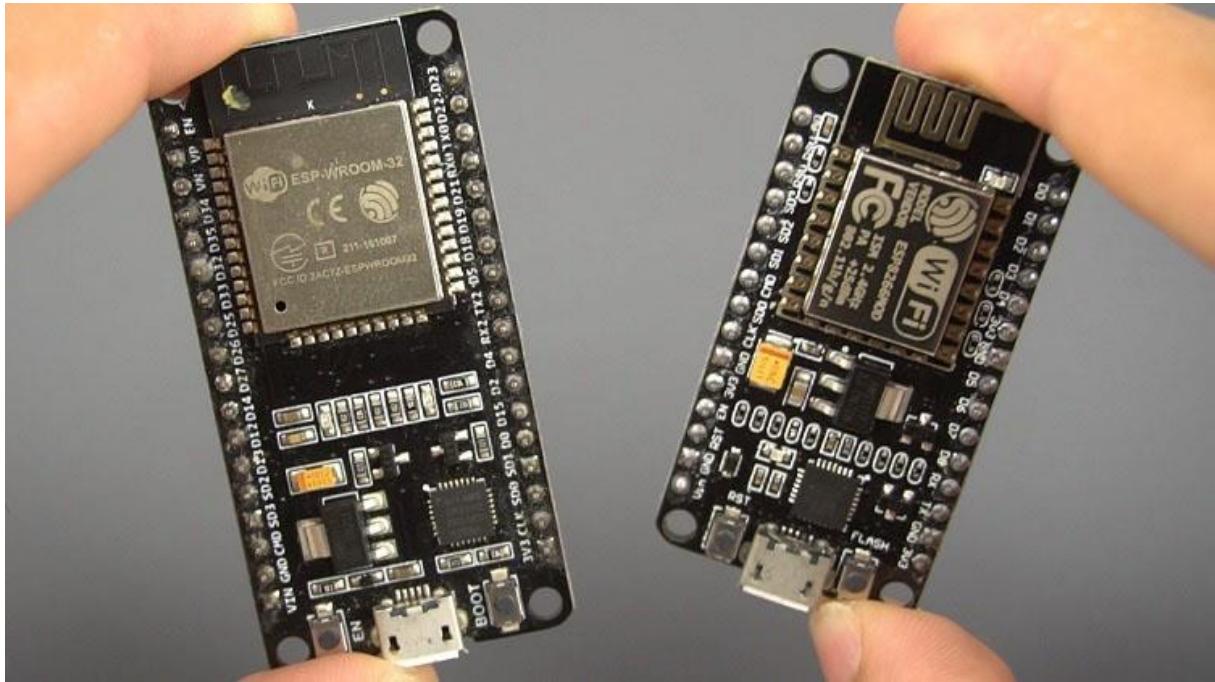
If you want to get a bit more technical and specific, you can take a look at the following detailed specifications of the ESP-WROOM-32 chip (source: <http://esp32.net/>)—for more details, check the datasheet):



- **Wireless connectivity Wi-Fi:** 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.

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 depends on where you get them and what model you're buying).

So, in summary:

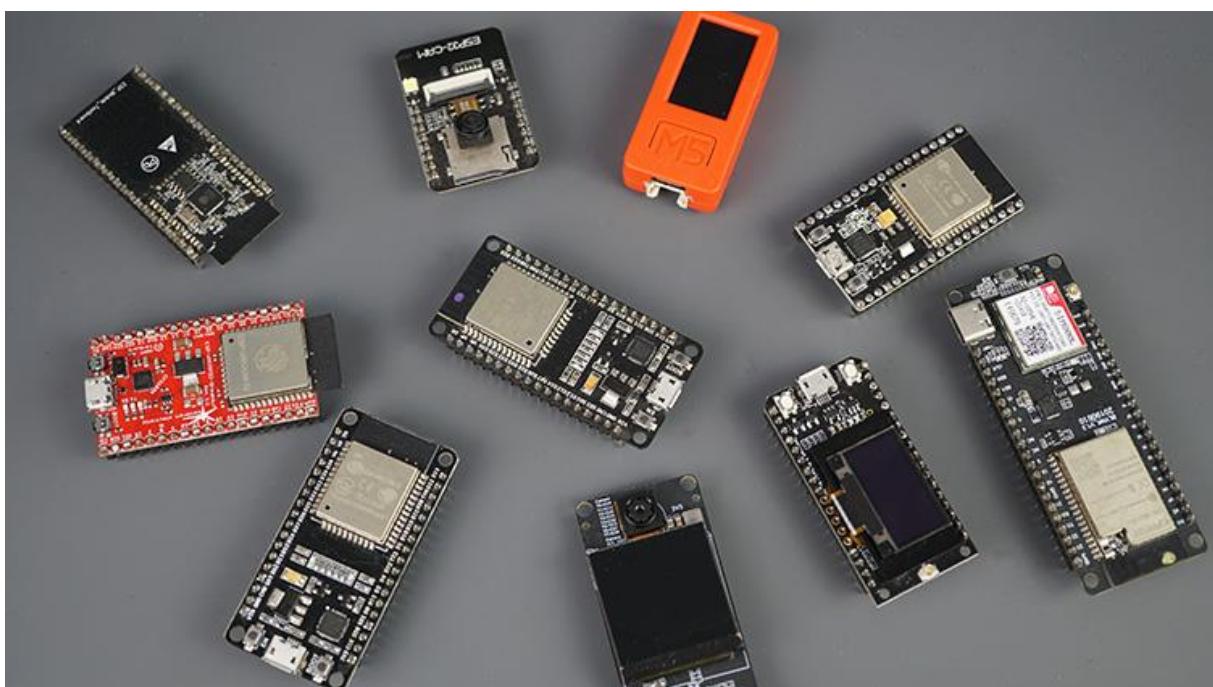
- The ESP32 is faster than the ESP8266;
- The ESP32 comes with more GPIOs with multiple functions;
- The ESP32 supports analog measurements on 18 channels (analog-enabled pins) versus just one 10-bit ADC pin on the ESP8266;
- The ESP32 supports Bluetooth while the ESP8266 doesn't;
- The ESP32 is dual-core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

For a more detailed explanation of the differences between the ESP32 and ESP8266, you can read 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 (like the [ESP32 CYD board](#)), 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 similarly, some boards may

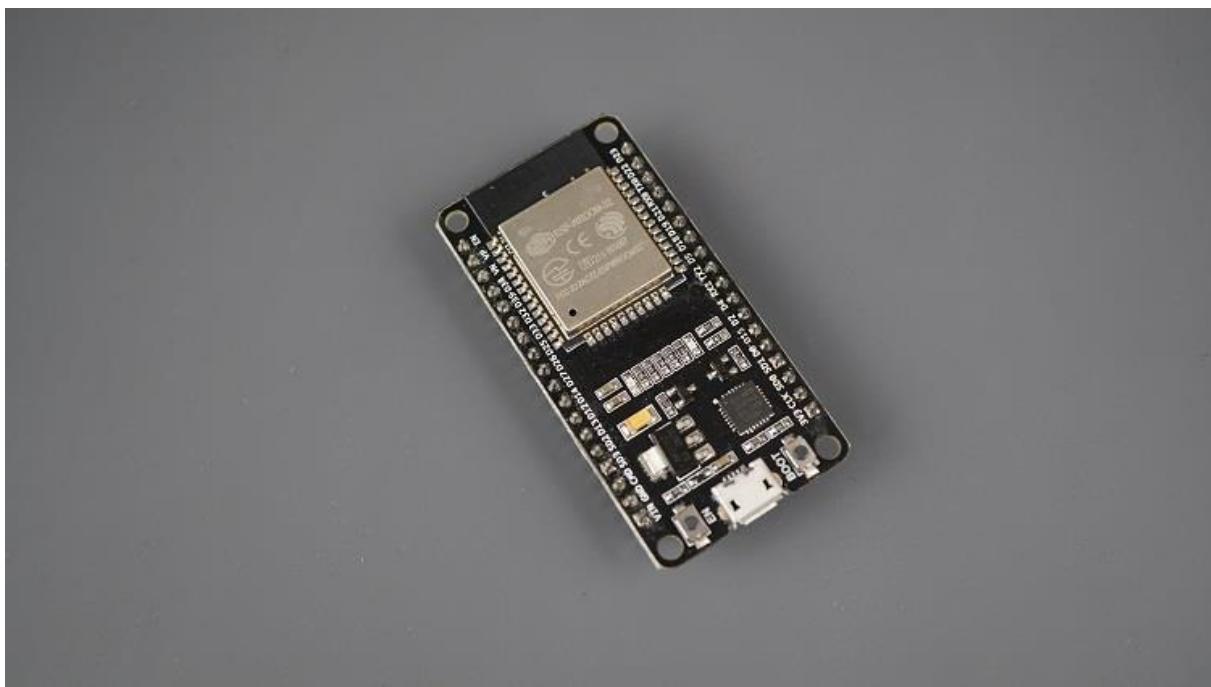
be more suitable for some projects than others. When looking for an ESP32 development board there are several aspects you need to take into account:

- **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.
- **GPIO 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. 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, a TFT Touchscreen display, 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 similarly.

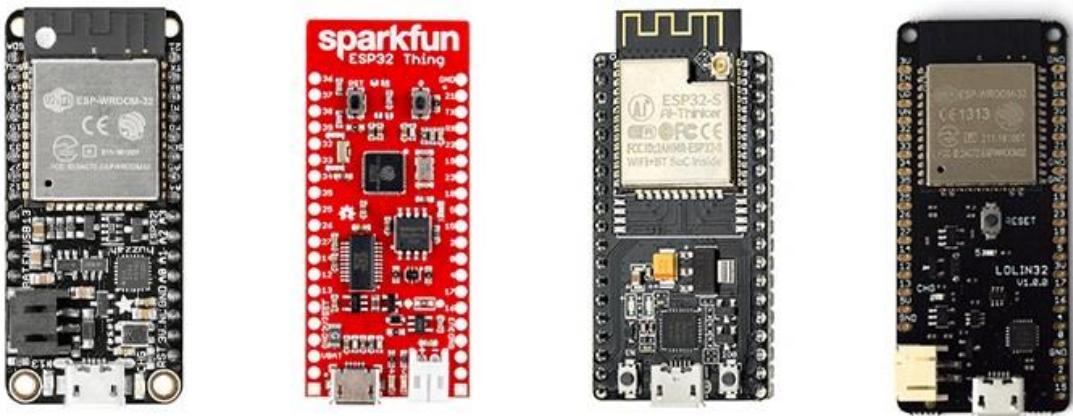


Where to Buy?

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

Throughout this course, we'll be using the ESP32 DEVKIT DOIT board as a reference. If you have a different board, don't worry. The information in this eBook is compatible with most ESP32 development boards.

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



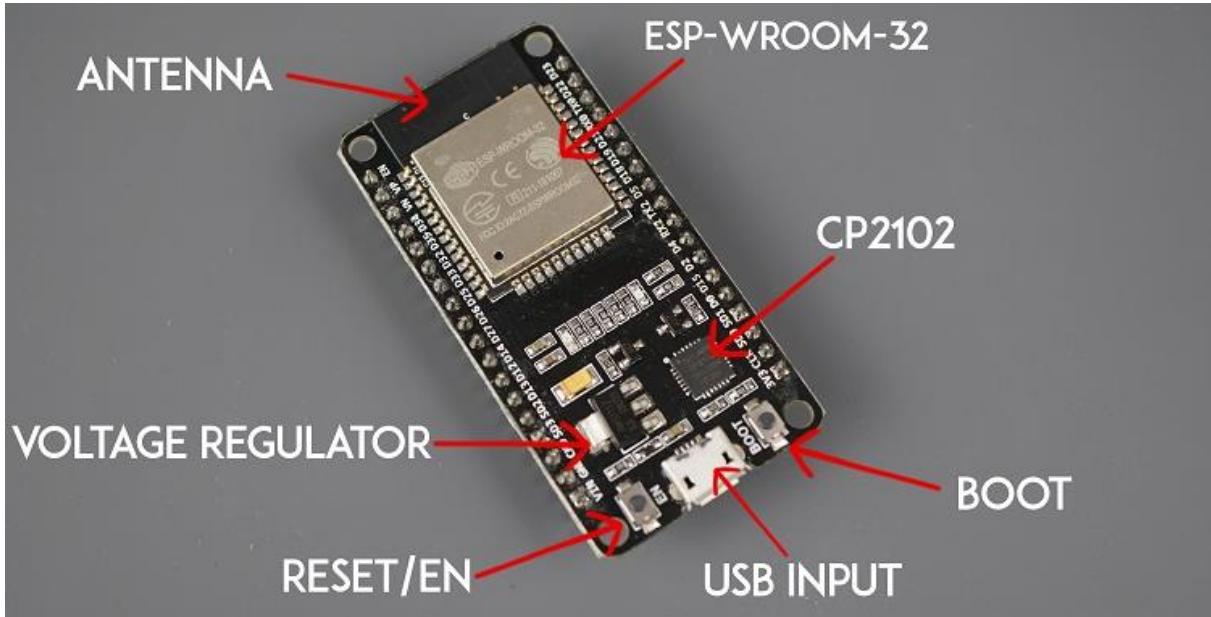
Specifications – ESP32 DEVKIT V1 DOIT

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

Specifications – ESP32 DEVKIT V1 DOIT	
Number of cores	2 (dual-core)
Wi-Fi	2.4 GHz up to 150 Mbit/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 board model
Peripherals	Capacitive touch, ADC (analog-to-digital converter), DAC (digital-to-analog converter), I ² C (inter-integrated circuit), UART (universal asynchronous receiver/transmitter), CAN 2.0 (controller area network), SPI (serial peripheral interface), I ² S (integrated inter-IC sound), RMII (reduced media-independent interface), PWM (pulse width modulation).

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 in Unit 2.1](#).

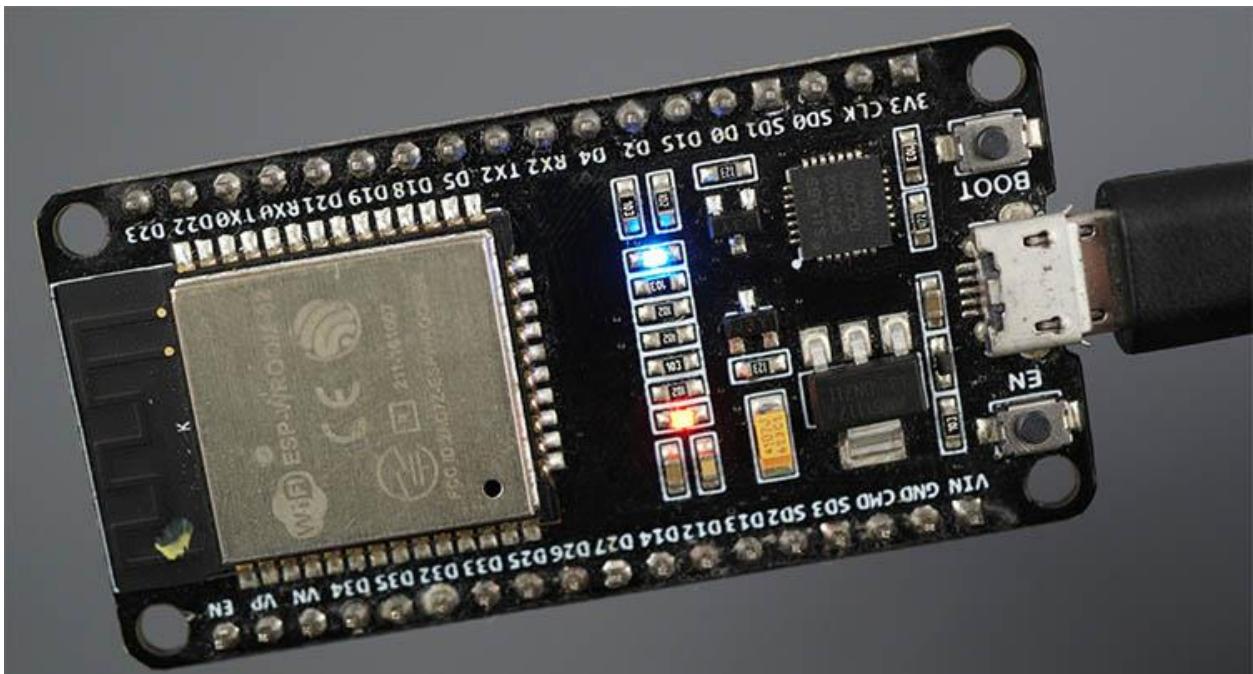


This ESP32 board comes with a microUSB interface that you can use to connect the board to your computer to upload code or apply power.

The device utilizes the *CP2102* chip (USB-to-UART) to establish communication with your computer through a COM port via a serial interface. Another widely used chip is the *CH340*. It's essential to identify the USB-to-UART chip on your board, as you'll need to install the appropriate drivers to enable your computer to communicate with the board. More detailed information about this process will be provided later.

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 (other boards may have the built-in LED connected to a different GPIO). 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. You'll find that different models will vary in the number of exposed GPIOs.

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 (you'll learn more about this later). This is possible due to the ESP32 chip's multiplexing feature that allows the assignment of 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 pinout diagram (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.

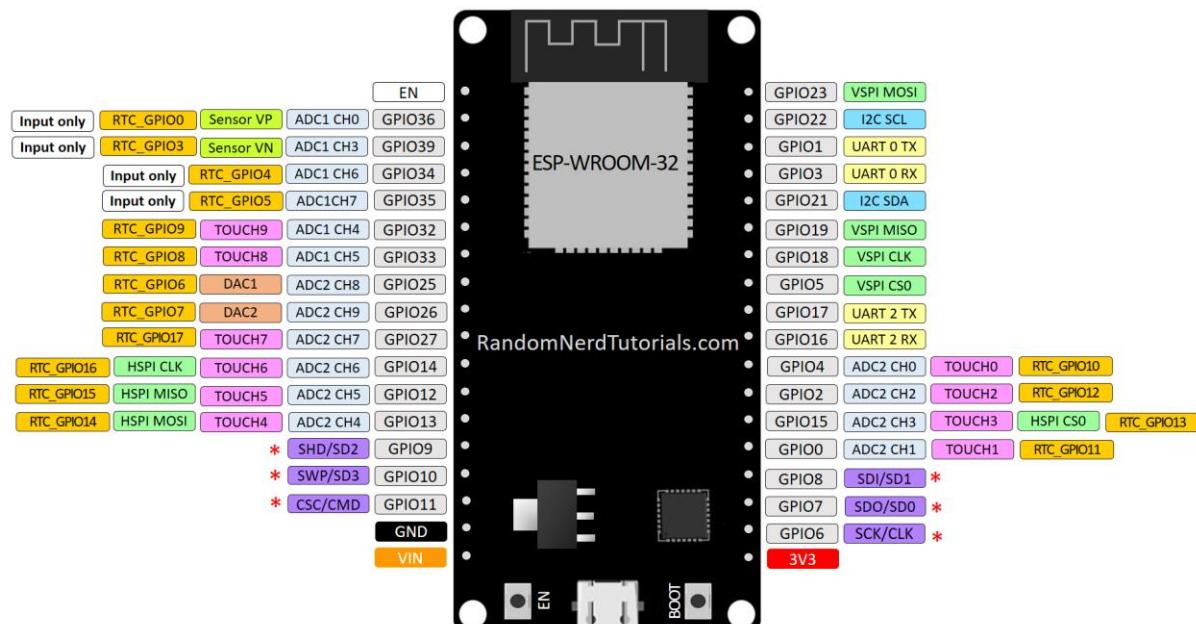
ESP32 Pinout

Note: To learn more about the ESP32 GPIOs, we recommend taking a look at [Unit 2.1](#). You can check the appropriate pins for your projects and which pins you should avoid using.

The following figure describes the board GPIOs and their functionalities. We recommend printing this pinout for future reference. You can download the pinout in *.pdf* or *.png* files:

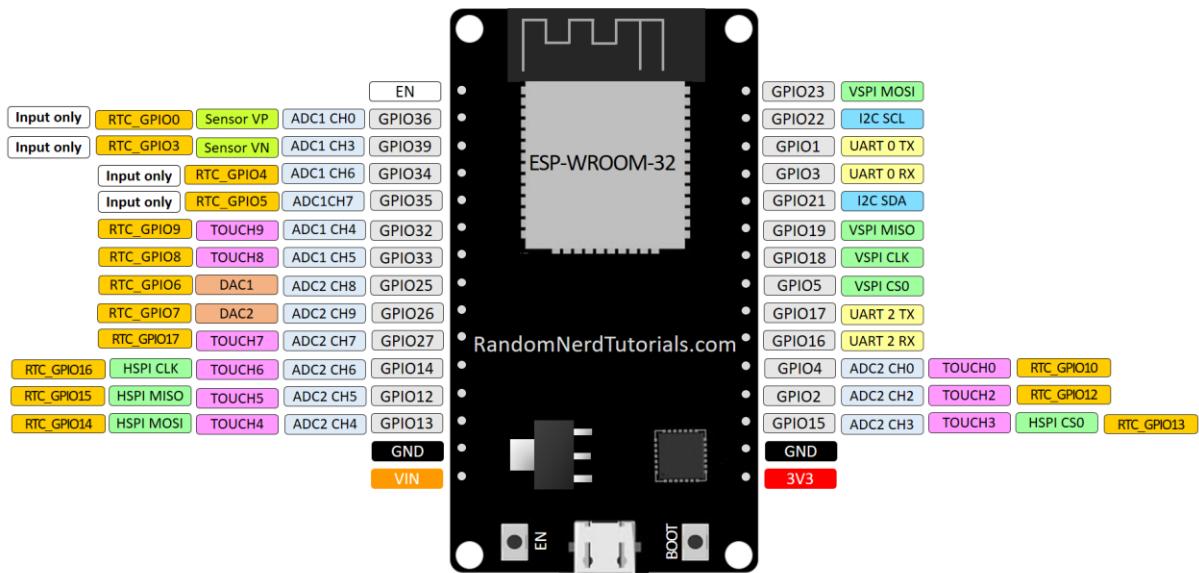
- [Printable version](#)
- [Image version 30 pins](#)
- [Image version 36 pins](#)

ESP32 DEVKIT V1 – DOIT
version with 36 GPIOs



ESP32 DEVKIT V1 – DOIT

version with 30 GPIOs



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, GPIO 5 is always the VSPI CS0 pin, GPIO 23 always corresponds to VSPI MOSI for SPI communication (this is true for boards with the ESP-WROOM-32 chip).

If you have a completely different ESP32 board, search online for its pinout diagram.

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
- And more...

This eBook focuses exclusively on programming the ESP32 with C/C++ “Arduino programming language”.

If you want to learn how to program the ESP32 board using MicroPython firmware, we have an eBook dedicated to that subject:

- [MicroPython Programming with ESP32 and ESP8266 eBook](#)

Next

Go to the next section to learn how to set up the ESP32 on the Arduino IDE.

1.2 - Installing ESP32 in Arduino IDE

To program your boards, you need an IDE to write your code. For beginners, we recommend using Arduino IDE. In this section, we'll show you how to install the ESP32 boards in Arduino IDE.

Requirements

You need JAVA installed on your computer. If you don't, go to the following website to download and install the latest version: <https://www.java.com/en/download/>

Downloading Arduino IDE

To download the Arduino IDE, visit the following URL:

- <https://www.arduino.cc/en/software>

The screenshot shows the Arduino IDE 2.3.2 download page. On the left, there's a teal rounded square icon with a white infinity symbol and a plus sign. To its right, the text "Arduino IDE 2.3.2" is displayed. Below this, a paragraph of text describes the new features of the release. Further down, there's a "SOURCE CODE" link and a note about GitHub. On the right side, a teal sidebar titled "DOWNLOAD OPTIONS" lists download links for Windows, Linux, and macOS, along with their respective file types (MSI installer, ZIP file, AppImage, ZIP file, Intel, and Apple Silicon). At the bottom of the sidebar, there's a "Release Notes" link.

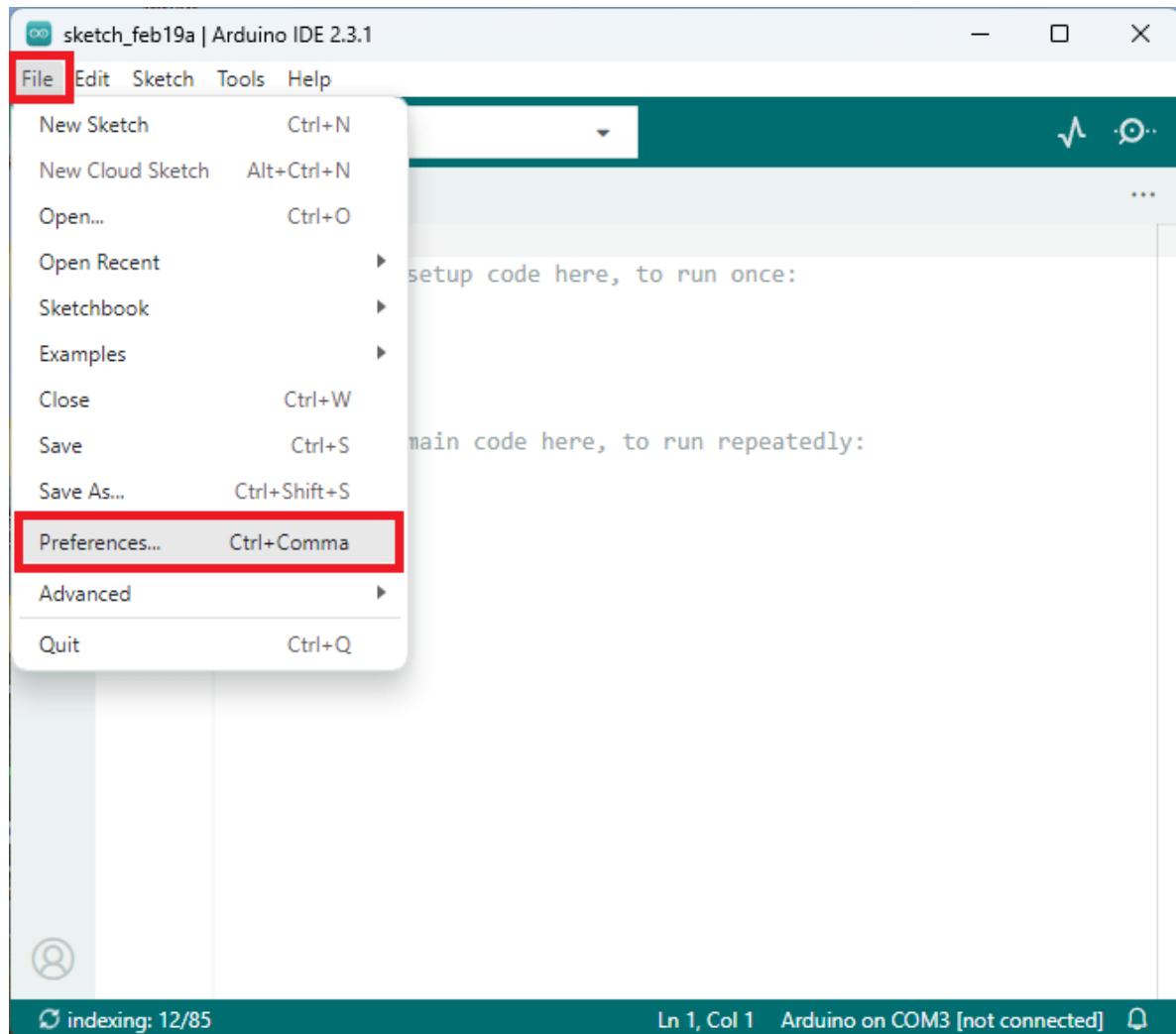
- **Windows:** extract the downloaded folder, run the executable file, and follow the instructions in the installation guide.
- **Mac OS X:** copy the downloaded file into your *application* folder.
- **Linux:** extract the downloaded file, and open the *arduino-ide* file that will launch the IDE.

If you have any doubts, you can go to the [Arduino Installation Guide](#).

Installing the ESP32 Boards in Arduino IDE

To program the ESP32 boards using Arduino IDE, you need to install the ESP32 boards in the IDE. Follow the next steps to install them.

Open the **Preferences** window in the Arduino IDE. Go to **File > Preferences**:



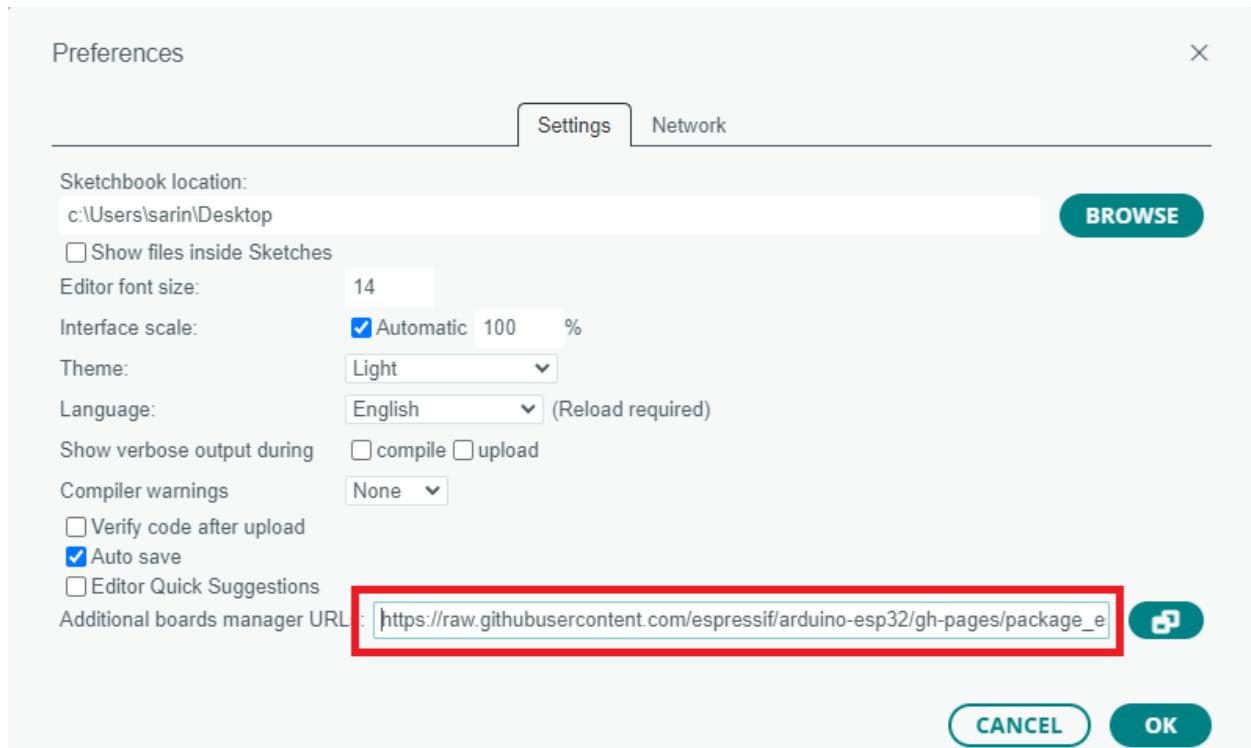
Copy and paste the following into the "Additional Board Manager URLs" field.

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

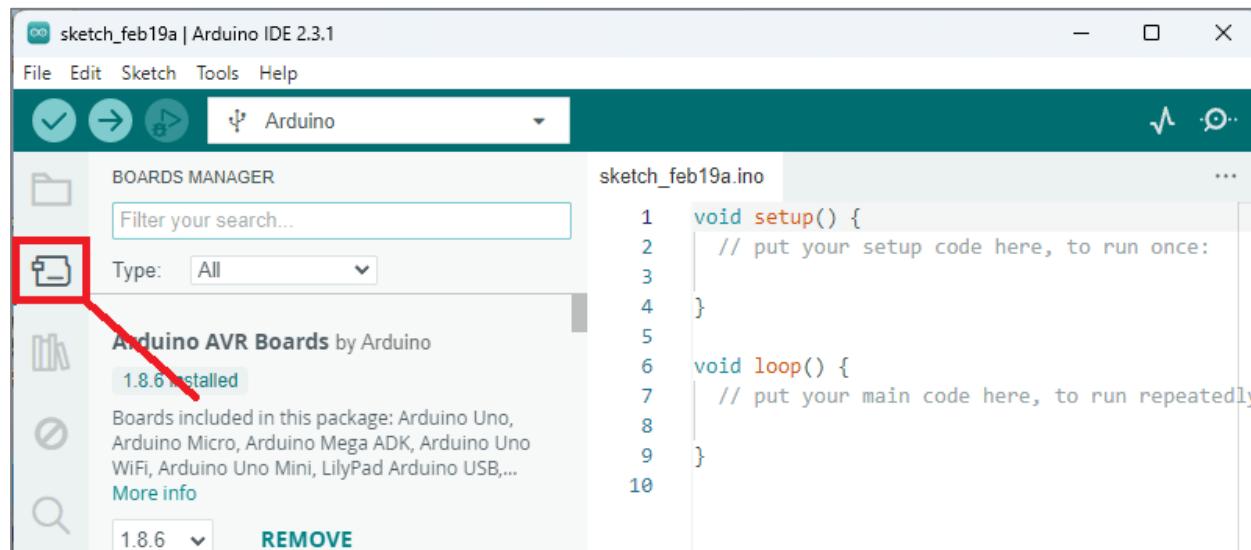
If you already have support for the ESP8266 boards, you can use two URLs as shown below:

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

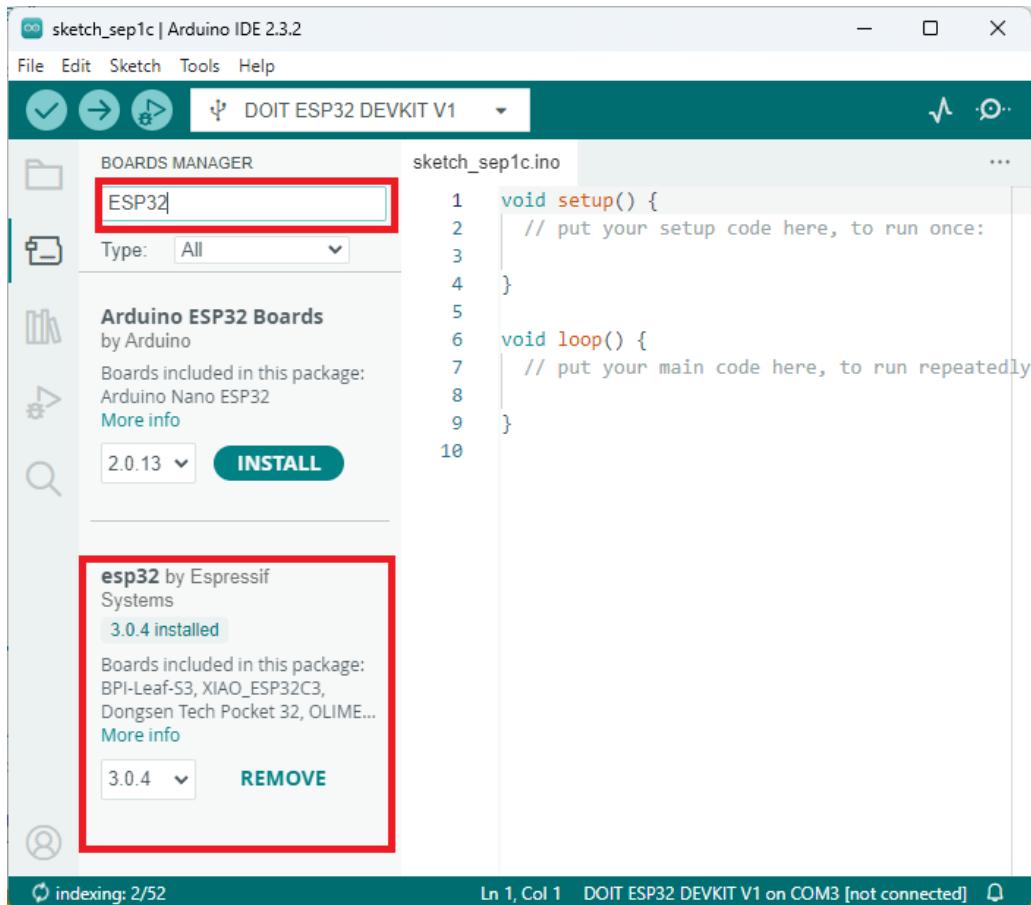
See the figure below. Then, click the "OK" button.



Open the **Boards Manager**. Go to **Tools > Board > Boards Manager...** or you can simply click the Boards Manager icon in the left-side corner.



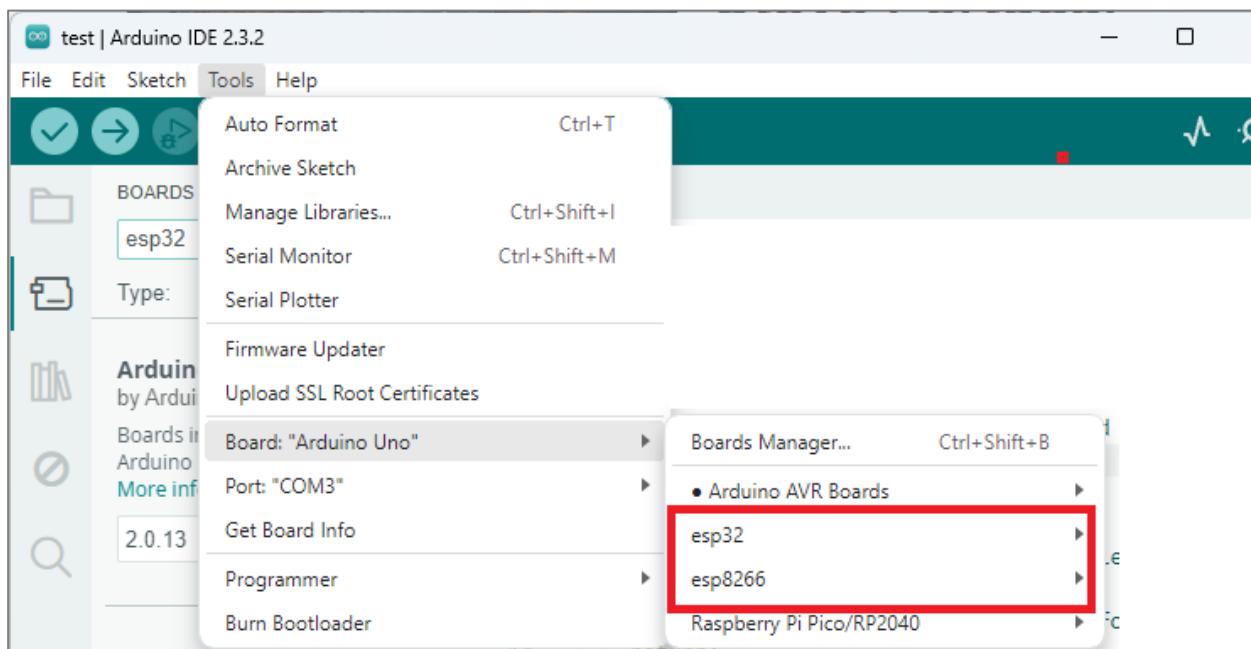
Search for ESP32 and install the "**ESP32 by Espressif Systems**":



You need to install at least version 3.0.x.

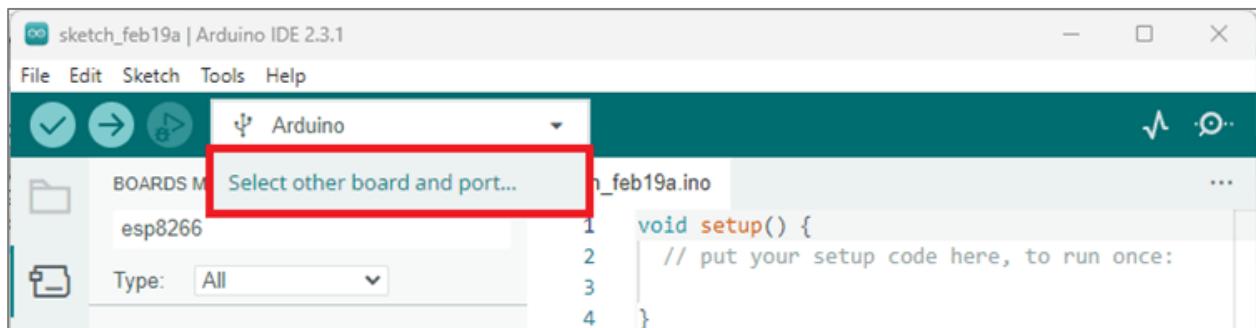
That's it. It will be installed after a few seconds. After this, restart your Arduino IDE.

Then, go to **Tools > Board** and check that you have ESP32 and ESP8266 boards available.

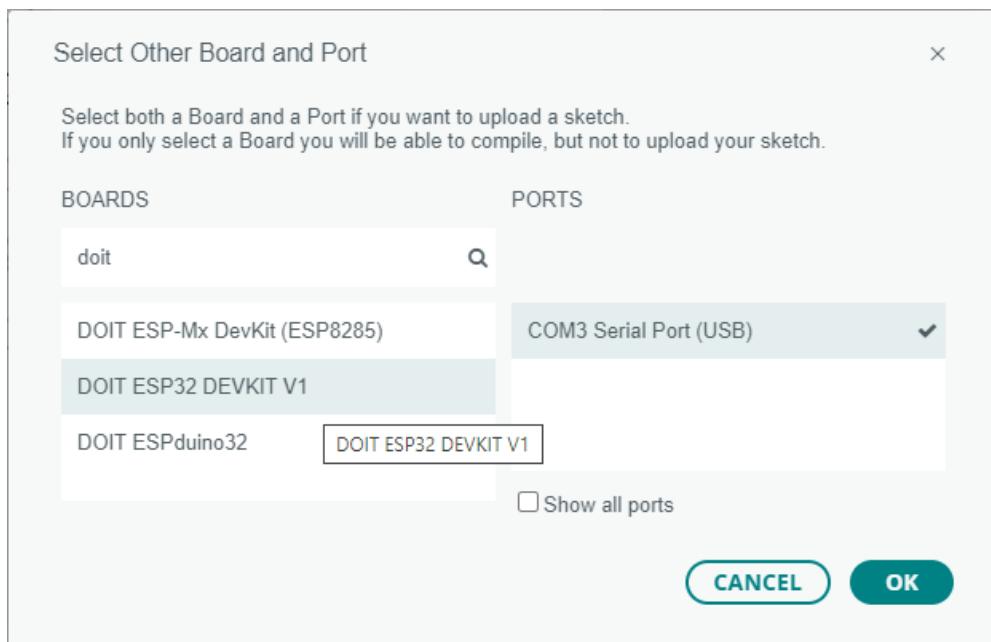


Testing the Installation

- 1) Select your Board in **Tools > Board** menu or on the top drop-down menu, click on “**Select other board and port...**”



A new window, as shown below, will open. Search for your ESP32 board model.



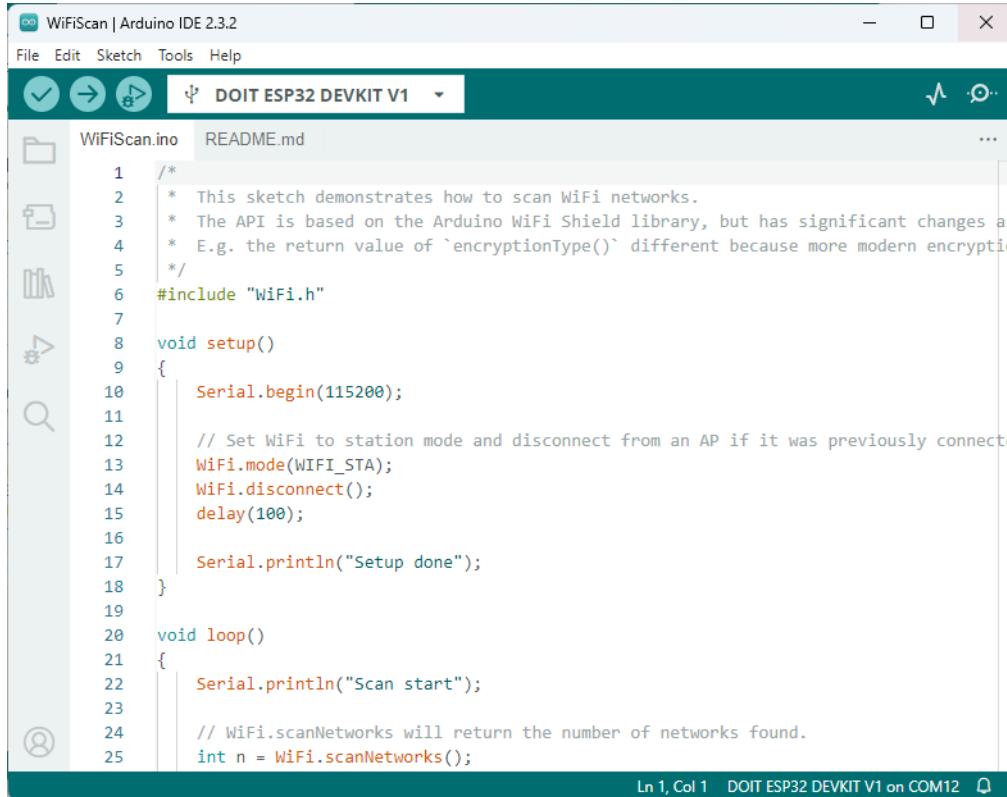
Select the board model you're using, and the COM port. In our example, we're using the DOIT ESP32 DEVKIT V1. Click **OK** when you're done.

If you don't see the COM Port in your Arduino IDE, you need to install the [CP210x USB to UART Bridge VCP Drivers](#) or other drivers depending on the board model you're using.

We recommend taking a look at this tutorial: [Install ESP32/ESP8266 USB Drivers – CP210x USB to UART Bridge](#)

- 2) Open the WiFiScan example— it searches for wi-fi networks within the range of your board.
 - o **File > Examples > WiFi > WiFiScan**

- 3) A new sketch opens in your Arduino IDE:



The screenshot shows the Arduino IDE 2.3.2 interface with the WiFiScan.ino sketch open. The code is as follows:

```
1  /*
2   * This sketch demonstrates how to scan WiFi networks.
3   * The API is based on the Arduino WiFi Shield library, but has significant changes as
4   * E.g. the return value of `encryptionType()` different because more modern encryption
5   */
6  #include "WiFi.h"
7
8  void setup()
9  {
10    Serial.begin(115200);
11
12    // Set WiFi to station mode and disconnect from an AP if it was previously connected
13    WiFi.mode(WIFI_STA);
14    WiFi.disconnect();
15    delay(100);
16
17    Serial.println("Setup done");
18 }
19
20 void loop()
21 {
22    Serial.println("Scan start");
23
24    // WiFi.scanNetworks will return the number of networks found.
25    int n = WiFi.scanNetworks();
```

Ln 1, Col 1 DOIT ESP32 DEVKIT V1 on COM12

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



Note: if you see a lot of dots on the debugging window, followed by an upload error, that means your board doesn't go into flashing mode automatically. Click the Arduino IDE Upload button again, and when you start seeing the dots on the debugging window, press the **onboard BOOT** button for a couple of seconds.

- 5) If everything goes as expected, it will upload successfully after a few seconds. You'll get a similar message:

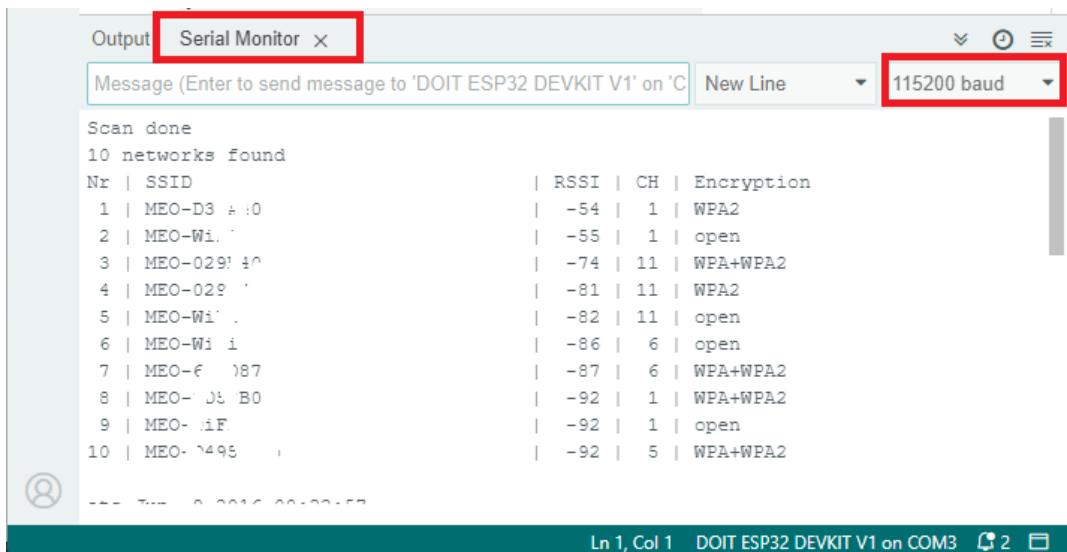
```
Output
Writing at 0x000b7163... (96 %)
Writing at 0x000bc7e1... (100 %)
Wrote 721168 bytes (468882 compressed) at 0x00010000 in 7.7 seconds (effective 747.8 kbi)
Hash of data verified.

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

- 6) Open the Arduino IDE Serial Monitor at a baud rate of 115200:



- 7) Press the ESP32 on-board Enable/RESET button and you should see the networks available near your board.



The screenshot shows the Arduino IDE's Serial Monitor window. The title bar has tabs for "Output" and "Serial Monitor" (which is currently active). Below the tabs is a message input field with placeholder text "Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'C')". To the right of the input field is a dropdown menu set to "115200 baud". The main area displays the output of a network scan. The text reads:

```
Scan done
10 networks found
Nr | SSID | RSSI | CH | Encryption
1 | MEO-D3 A :0 | -54 | 1 | WPA2
2 | MEO-Wi. | -55 | 1 | open
3 | MEO-029! 40 | -74 | 11 | WPA+WPA2
4 | MEO-029 | -81 | 11 | WPA2
5 | MEO-Wi' | -82 | 11 | open
6 | MEO-Wi i | -86 | 6 | open
7 | MEO-f 087 | -87 | 6 | WPA+WPA2
8 | MEO- Jt B0 | -92 | 1 | WPA+WPA2
9 | MEO- iF | -92 | 1 | open
10 | MEO- 2495 | -92 | 5 | WPA+WPA2
```

Everything seems to be working as expected, you can proceed to the next module.

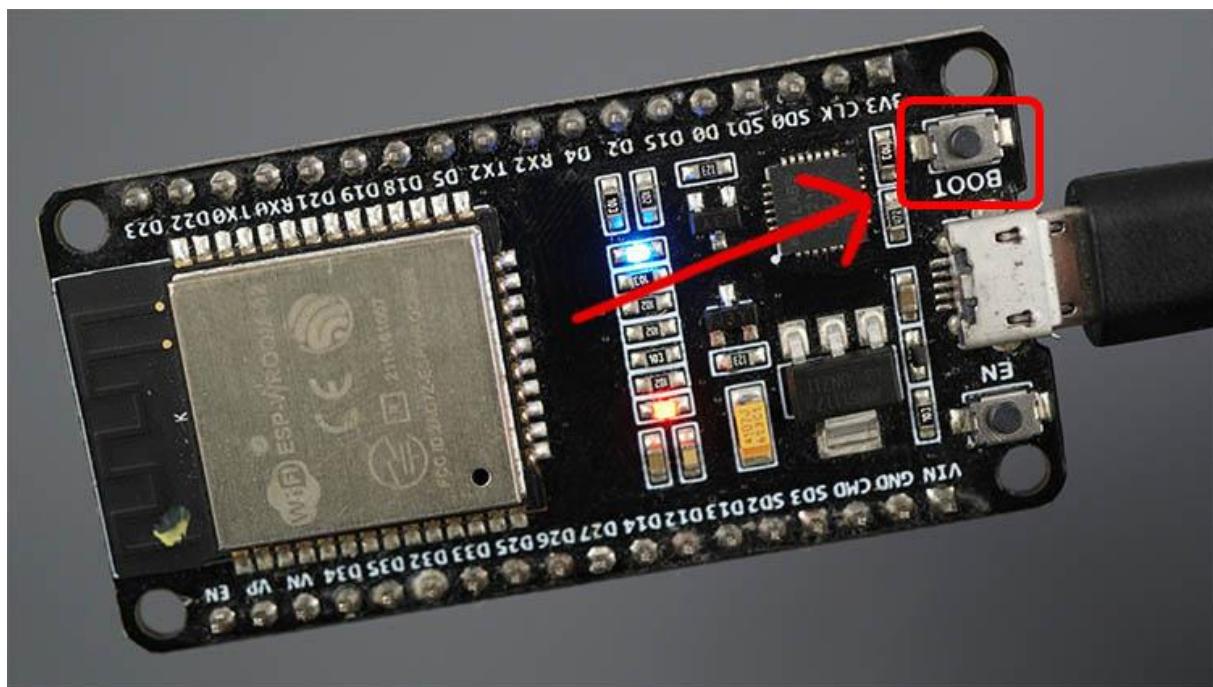
If you're having issues uploading code to your ESP32 board, we recommend taking a quick look at the troubleshooting tips on the following pages.

Troubleshooting Tip #1

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

When you try to upload a new sketch to your ESP32, and it fails to connect to your board, 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 on your ESP32 board.



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



After you see the **"Connecting...."** message in your Arduino IDE, release the **"BOOT"** button.

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

Troubleshooting Tip #2

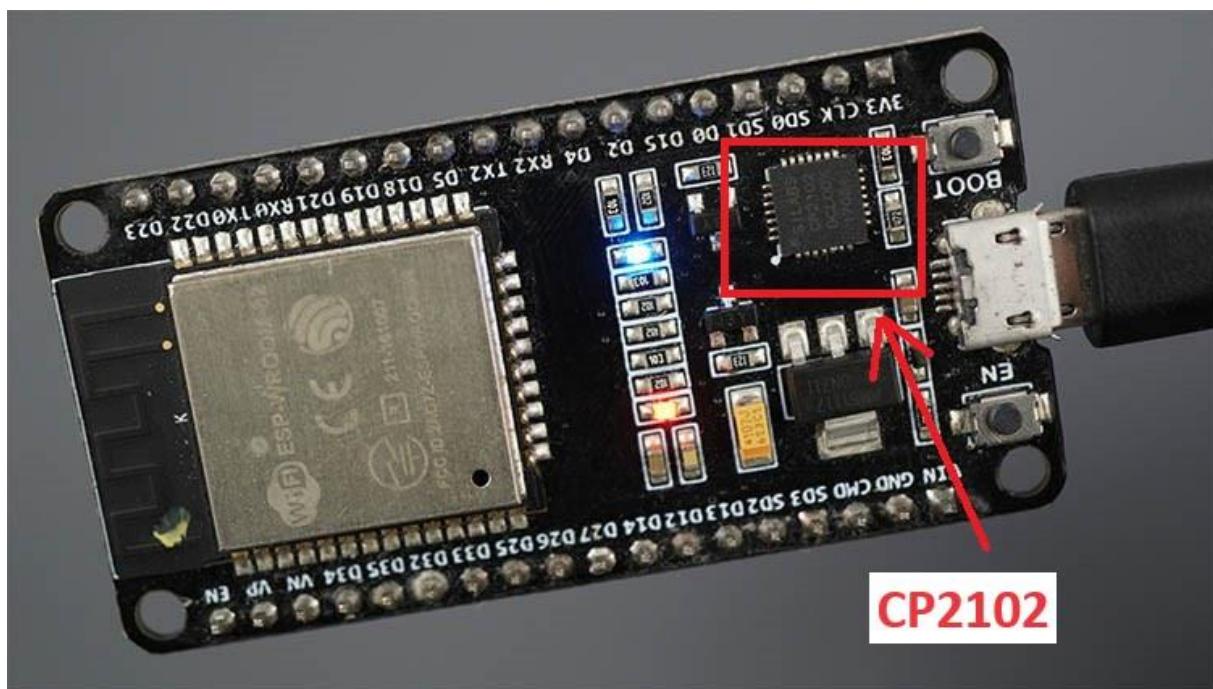
COM Port not found/not available

If you plug your ESP32 board into your computer, but you can't find the ESP32 COM port available in your Arduino IDE (it's grayed out):

It might be one of these two problems: **1. USB drivers are missing**, or **2. USB cable without data wires**.

1. If you don't see your ESP's COM port available, this often means you don't have the USB drivers installed. Take a closer look at the chip next to the voltage regulator on board and check its name.

The [ESP32 DEVKIT V1 DOIT](#) board uses the CP2102 chip.



You can follow the next tutorial to see how to install the CP210x drivers:

- [Install ESP32 USB Drivers: CP210x USB to UART Bridge \(Windows PC\)](#)
- [Install ESP32 USB Drivers: CP210x USB to UART Bridge \(Mac OS X\)](#)

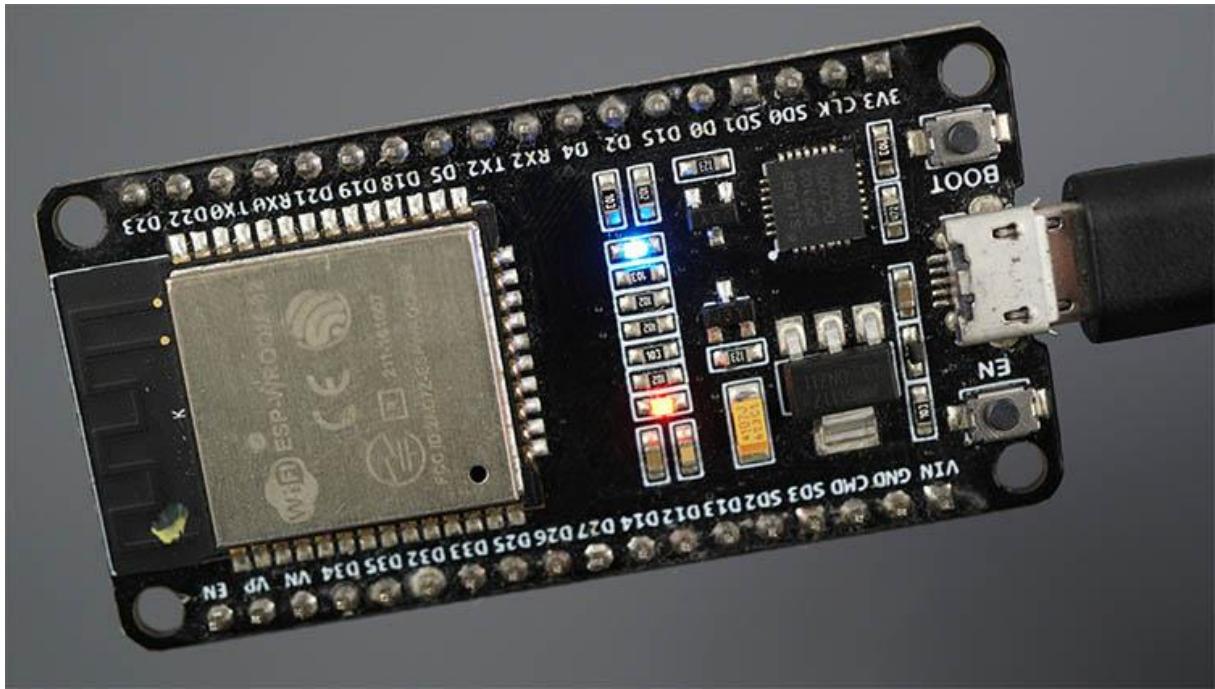
After installing, restart the Arduino IDE, and you should see the COM port in the Tools menu.

If your board uses different drivers, you should be able to find an installation guide with a quick Google search. Another common USB-to-UART chip used on ESP32 boards is the CH340.

2. If you have the drivers installed but you can't see your device, double-check that you're using a USB cable with data wires. USB cables from powerbanks often don't have data wires (they are charge only). So, your computer will never establish a serial communication with your ESP32. Using a proper USB cable should solve your problem.

1.3 - How to Use Your ESP32 Board with this Course

We'll be using the ESP32 DEVKIT V1 DOIT board for this course, but most ESP32 development boards should work just fine.



Here are just a few examples of boards that are compatible with this course.

DOIT DEVKIT V1



ESP32 DevKit



ESP-32S NodeMCU



ESP32 Thing



WEMOS LOLIN32



"WeMos" OLED



HUZZAH32



Others

(...)

There are many ESP32 development boards available, and if you have a different board than these ones, you can also follow the course. We recommend getting one with a similar ESP32 chip: the ESP-WROOM-32.

How to Use Your ESP32 Board with this Course

This section shows the possible changes you may need to make if you use a different ESP32 board.

You just need to keep in mind two main aspects:

- 1) How you wire a circuit to your specific ESP32
- 2) Select the correct board in the Arduino IDE

Finding Your Board Model

Here are a couple of tips to identify your ESP32 board model.

Visiting the Product Page

A good place to start is to open the product page where you purchased your ESP32. I've ordered mine from [Banggood, but you can find it at different stores.](#)



Please note that some vendors will add all sorts of keywords to their product name, so you might be thinking that you're ordering an ESP-32S NodeMCU board, and you bought an ESP32 DOT IT board.

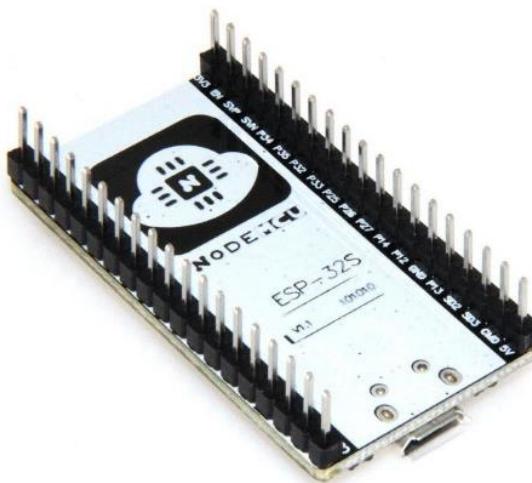
ESP32 Board Name

After reading your ESP32 product page, you should know the name of your board. But if you still have doubts, you can take a look at the back of the board.

The name is usually printed on the board. In my case, you can see this is the ESP32 DEVKIT V1 DOIT board.



If you have an ESP32 NodeMCU, the following figure shows what it looks like (it says NodeMCU ESP-32S).



ESP32 Pinout

Knowing the name of your board is very important so that you can search for its pinout. Search for your ESP32 board name and add the “pinout” keyword at the end.



ESP32 DEVKIT V1 DOIT pinout



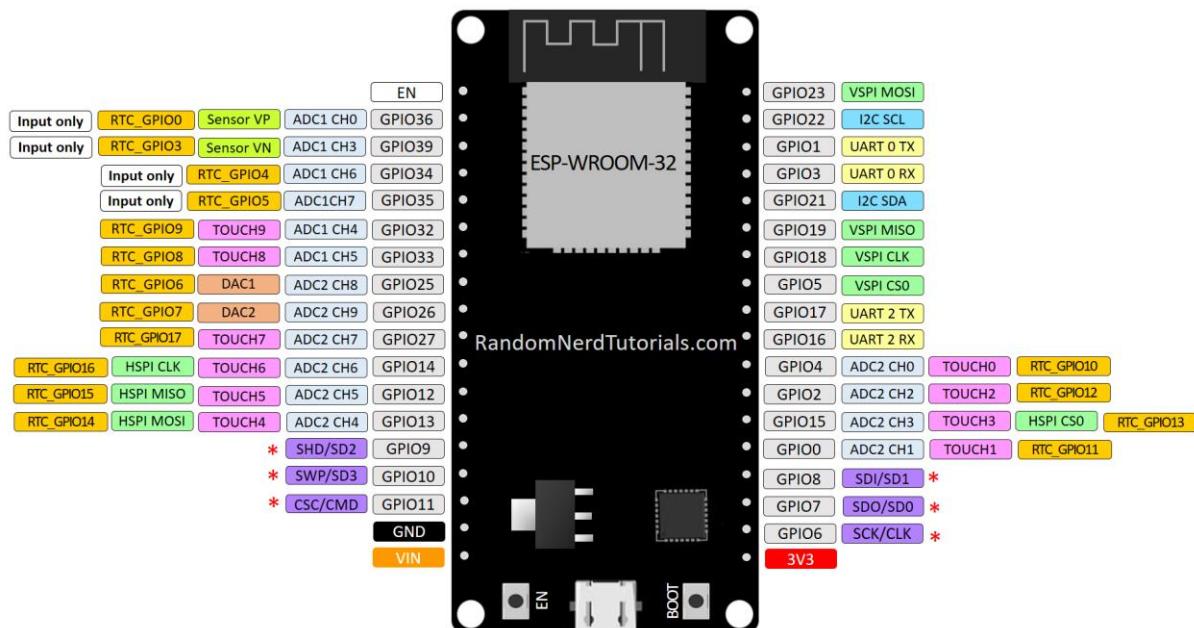
Google Search

I'm Feeling Lucky

Then, find one image that has the same pinout as your ESP32. Here's the ESP32 DEVKIT V1 DOIT board pinout:

- [Printable version](#)
- [Image version 30 pins](#)
- [Image version 36 pins](#)

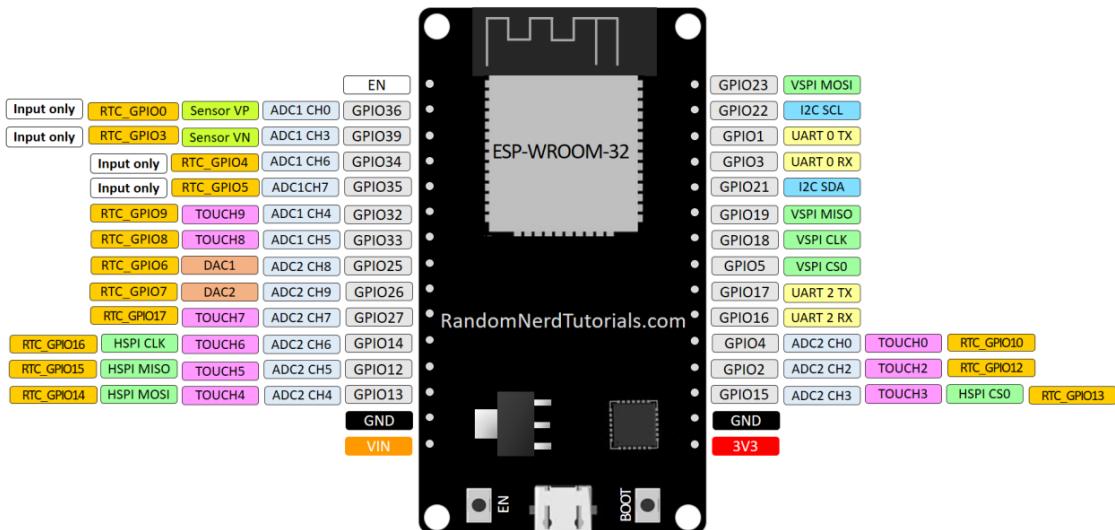
ESP32 DEVKIT V1 – DOIT version with 36 GPIOs



* Pins SCK/CLK, SDO/SD0, SDI/SD1, SHD/SD2, SWP/SD3 and SCS/CMD, namely, GPIO6 to GPIO11 are connected to the integrated SPI flash integrated on ESP-WROOM-32 and are not recommended for other uses.

ESP32 DEVKIT V1 – DOIT

version with 30 GPIOs



I also recommend visiting the ESP32.net website as it provides an extensive list with names and pictures for all known ESP32 development boards.

Blink – Example Sketch

Let's examine a sample sketch to highlight the key elements for building a simple circuit that blinks an LED using the ESP32. Copy the following code into the Arduino IDE:

- [Click here to download the code](#)

```
// ledPin refers to ESP32 GPIO 23
const int ledPin = 23;

// the setup function runs once when you press reset or power the board
void setup() {
    // initialize digital pin ledPin as an output.
    pinMode(ledPin, OUTPUT);
}

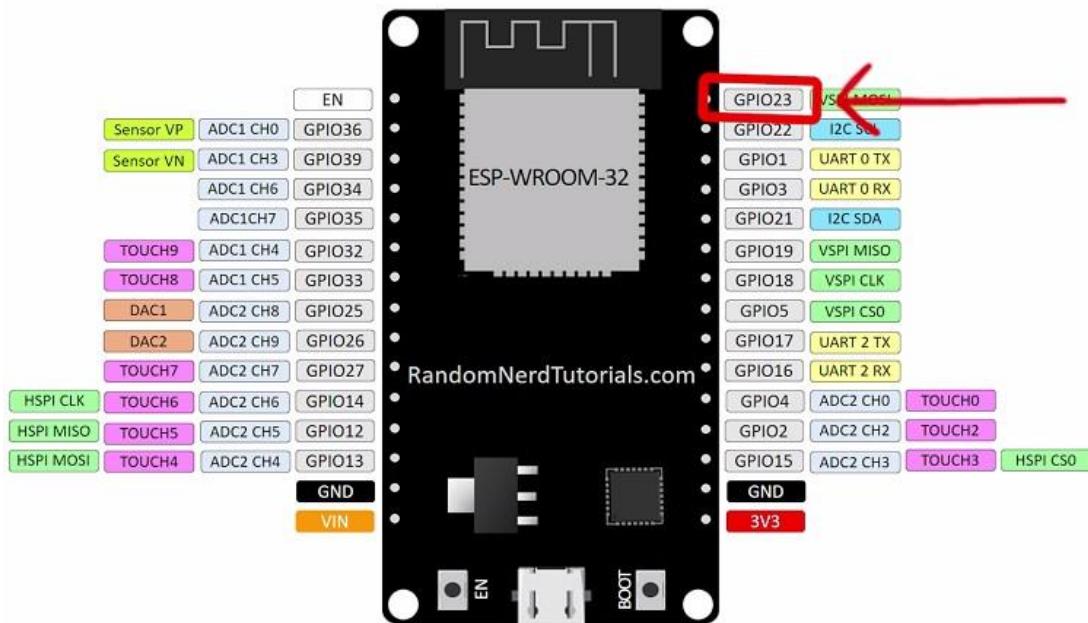
// the loop function runs over and over again forever
void loop() {
    digitalWrite(ledPin, HIGH);      // turn the LED on (HIGH is the voltage level)
    delay(1000);                  // wait for a second
    digitalWrite(ledPin, LOW);       // turn the LED off by making the voltage LOW
    delay(1000);                  // wait for a second
}
```

As you can see, you need to connect the LED to pin 23, which refers to GPIO 23:

```
const int ledPin = 23;
```

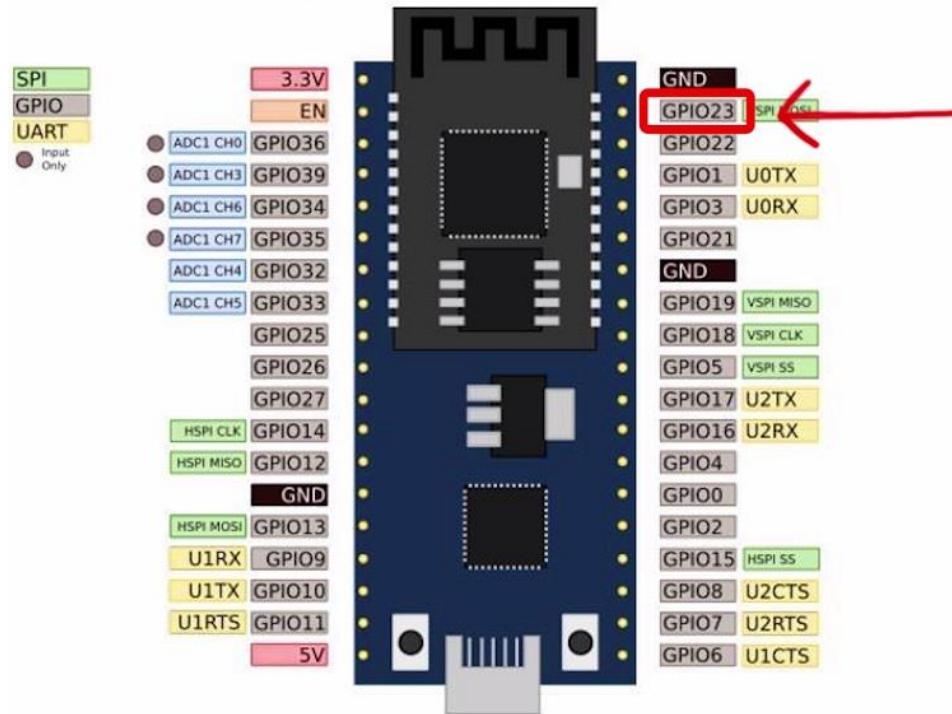
If you're using the ESP32 DEVKIT V1 DOIT board, you need to connect your LED to the first pin on the top right corner.

ESP32 DEVKIT V1 - DOIT



But if you're using the NodeMCU ESP-32S board, GPIO 23 is located in the 2nd pin at the top right corner—see the picture below.

NodeMCU ESP-32S



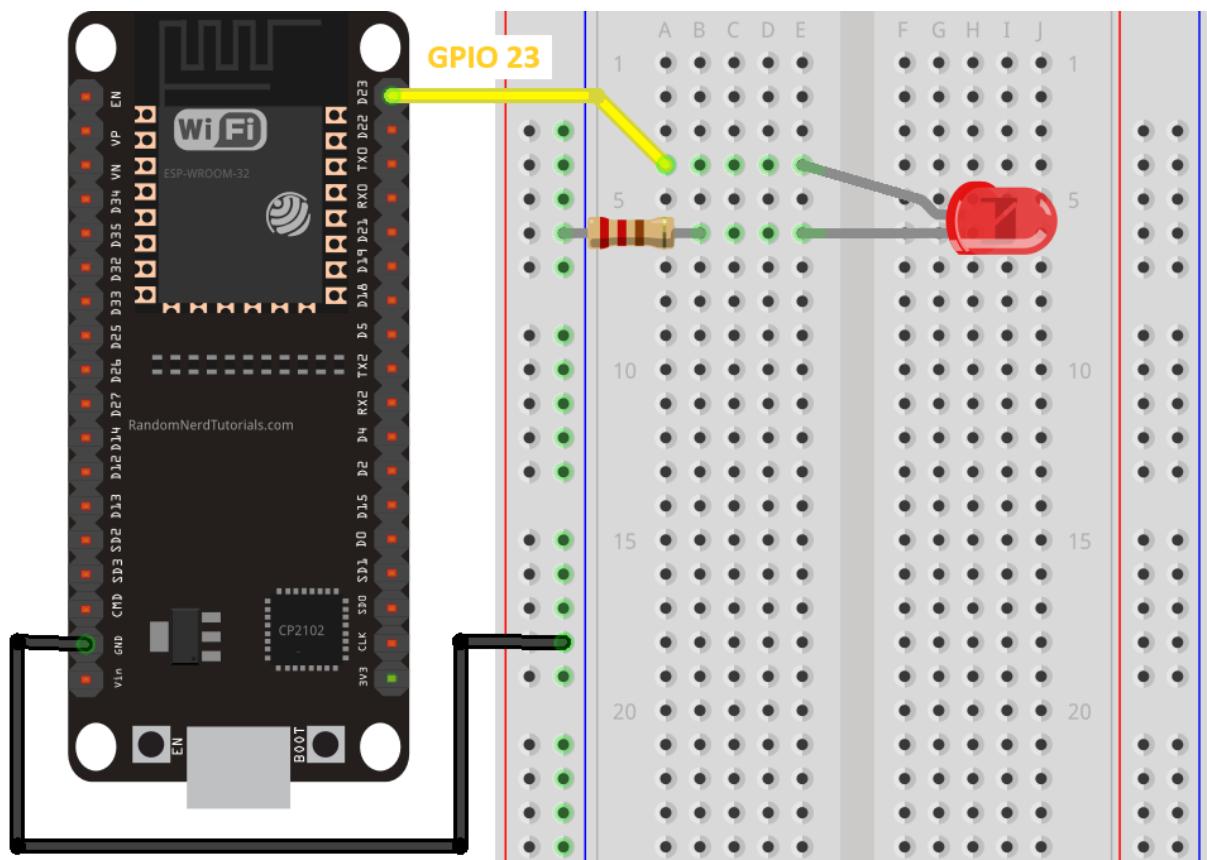
The location of GPIO 23 might differ depending on the board you're using. So make sure you always check the pinout for your specific board before building any circuit.

Wiring the Circuit

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [220 Ohm resistor \(or similar value\)](#)
- [Jumper wires](#)
- [Breadboard](#) (optional)

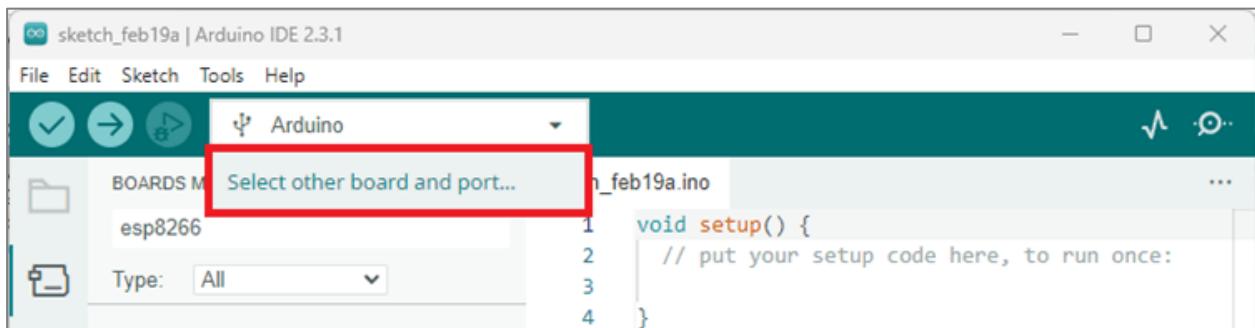
Follow the next diagram to wire the LED to the ESP32 DEVKIT DOIT board (also check where GND is located in your board).



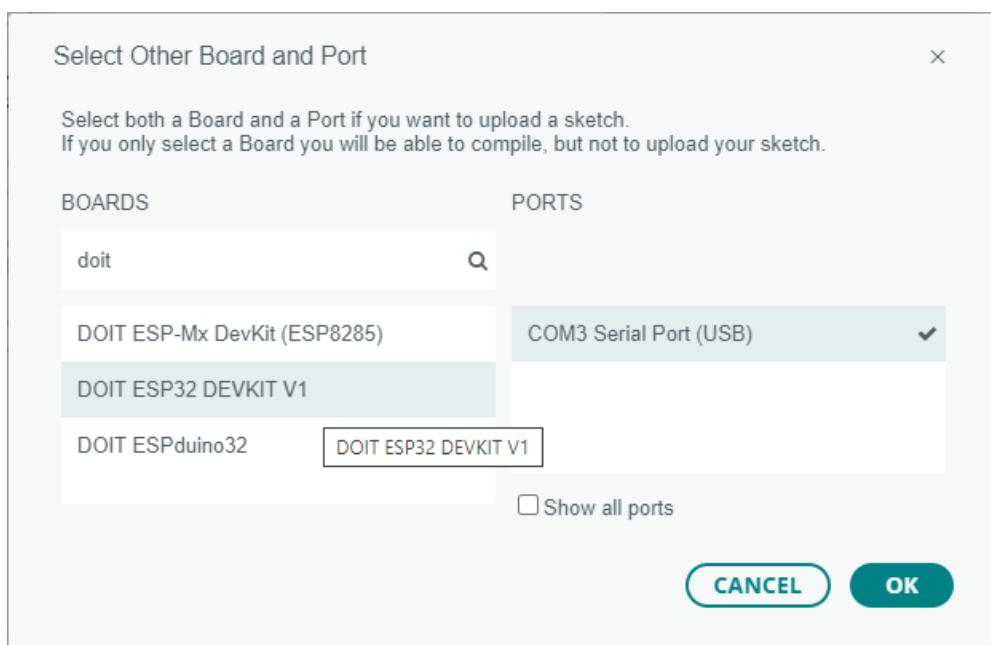
Preparing the Arduino IDE

After wiring the circuit and connecting the ESP32 board to your computer, follow the next steps.

- 1) Select your Board in **Tools > Board** menu or on the top drop-down menu, click on “**Select other board and port...**”



A new window, as shown below, will open. Search for your ESP32 board model.



Select the board model you're using, and the COM port. In our example, we're using the DOIT ESP32 DEVKIT V1. Click **OK** when you're done.

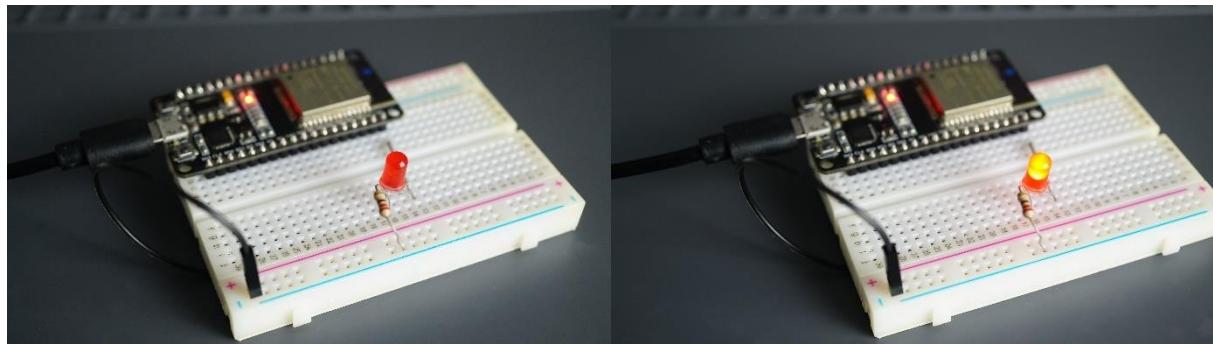
Note: if you don't find your board name on the list, the **ESP32 Dev Module** option works well for most models.

Uploading the Sketch

Click on the Arduino IDE upload button, and wait a few seconds while it compiles and uploads your sketch to the board.



The LED attached to GPIO 23 should be blinking every other second.



Wrapping Up

Here are the steps that you should follow to identify your board, wire the circuit, and upload the code:

- Go to the board product page to see if you find any relevant information that allows you to identify your board;
- Check the back of your board—it might have its name printed;
- Use the esp32.net website to find more information about your board;
- Use Google image search to find the pinout for your board and print it to use it as a reference;
- Before wiring any circuit, check your board pinout;
- To upload code, select the right board and COM port in the Arduino IDE.

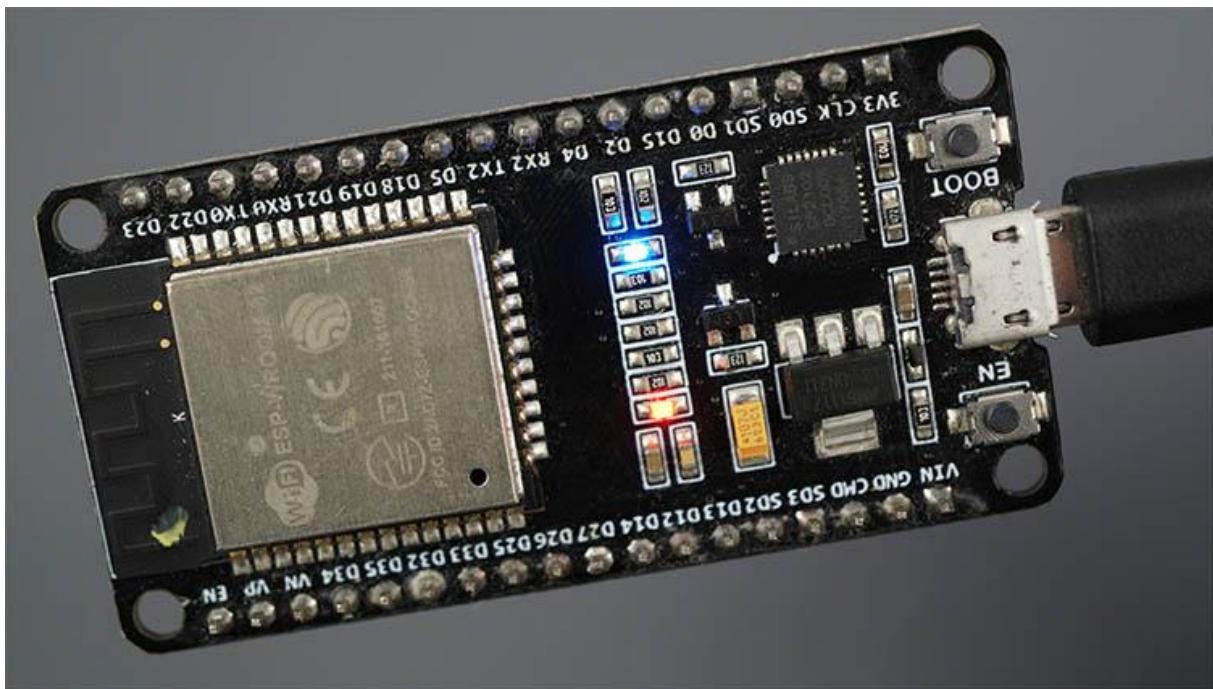
That's it. I hope this is helpful and makes it easier for you to use the resources in this eBook, regardless of your ESP32 development board.

MODULE 2

Exploring the ESP32 GPIOs

2.1 - Pinout Reference: ESP32 GPIOs Explained

The ESP32 chip (ESP-WROOM-32) comes with 48 pins with multiple functions. Not all pins are exposed in all ESP32 development boards, and some pins cannot be used for certain functions. There are many questions on how to use the ESP32 GPIOs. What pins should you use? What pins should you avoid using? This section is a simple easy-to-follow reference guide for the ESP32 GPIOs.



ESP32 Peripherals

The ESP32 peripherals include:

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

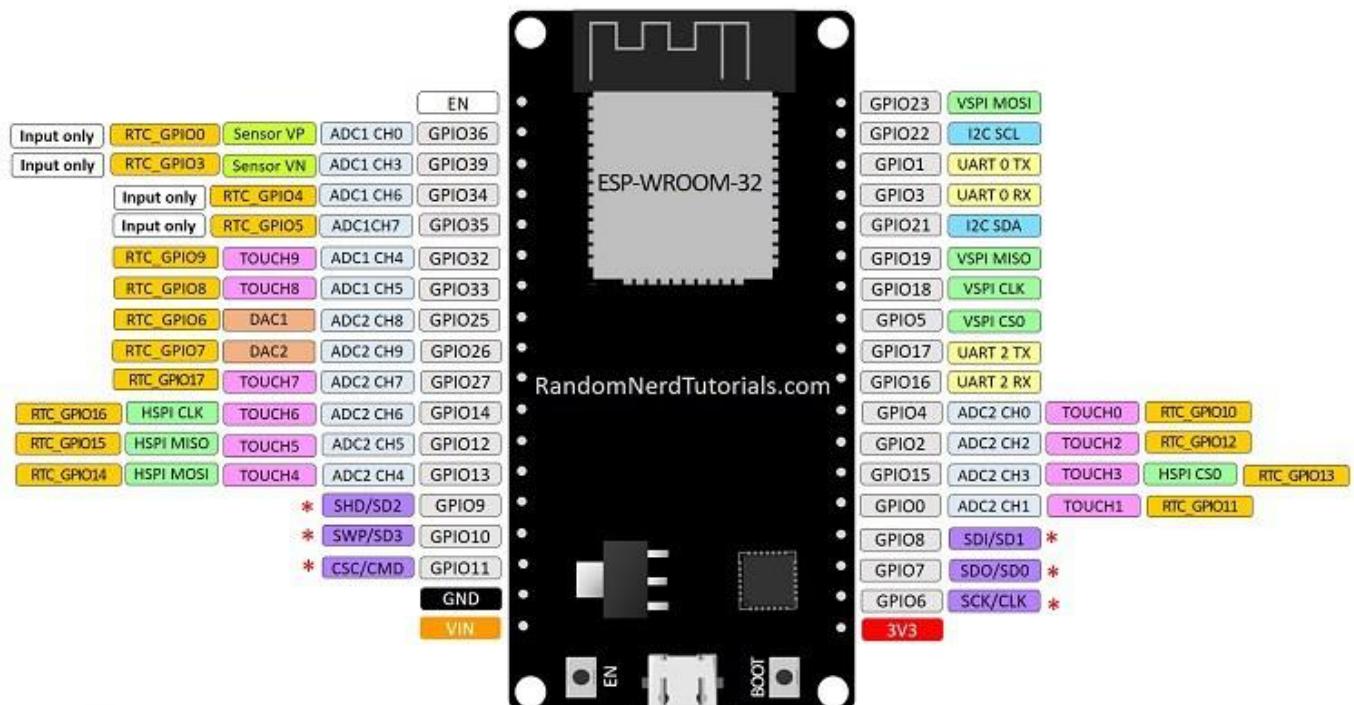
Note: not all GPIOs are accessible in all development boards, but each specific GPIO works in the same way regardless of the development board you're using (as long as it has the same chip ESP-WROOM-32).

The ADC (analog to digital converter) and DAC (digital to analog converter) features are assigned to specific static pins. In the case of UART, I2C, SPI, and PWM, 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**.

The ESP32 multiplexing feature allows us to reassign most of the GPIO functions to different pins. Although you can define the pins properties on the software, we'll discuss the pins assigned by default, as shown in the following figure (this is an example for the [ESP32 DEVKIT V1 DOIT board](#) with 36 pins. Note that the pin location can change depending on the manufacturer).

ESP32 DEVKIT V1 – DOIT

version with 36 GPIOs



* Pins SCK/CLK, SDO/SD0, SDI/SD1, SHD/SD2, SWP/SD3 and CSC/CMD, namely, GPIO6 to GPIO11 are connected to the integrated SPI flash integrated on ESP-WROOM-32 and are not recommended for other uses.

Additionally, there are pins with specific features that make them suitable or not for a particular project.

The next 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 (when the ESP32 starts/restarts). 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
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 fail 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

Input only pins

GPIOs 34 to 39 are GPIOs—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)

Analog to Digital Converter (ADC)

The ESP32 has 18x 12-bit ADC input. 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)

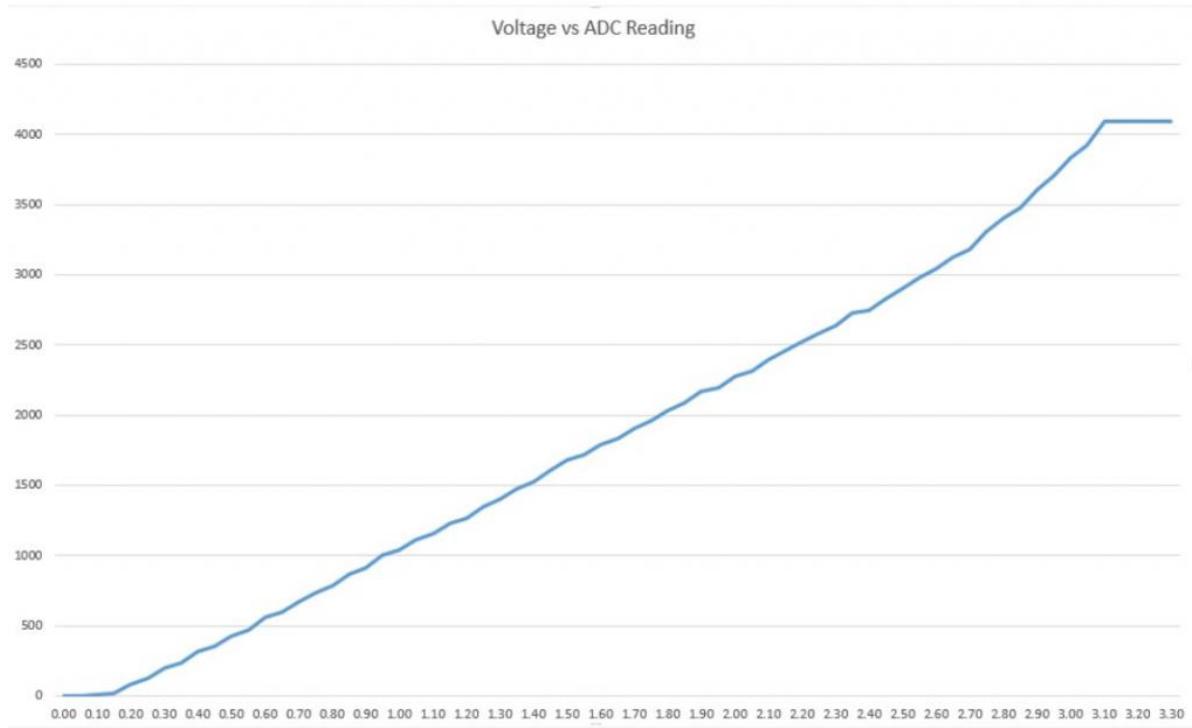
- ADC1_CH5 (GPIO 33)
- ADC1_CH6 (GPIO 34)
- ADC1_CH7 (GPIO 35)
- ADC2_CH0 (GPIO 4)
- ADC2_CH1 (GPIO 0)
- ADC2_CH2 (GPIO 2)
- ADC2_CH3 (GPIO 15)
- ADC2_CH4 (GPIO 13)
- ADC2_CH5 (GPIO 12)
- ADC2_CH6 (GPIO 14)
- ADC2_CH7 (GPIO 27)
- ADC2_CH8 (GPIO 25)
- ADC2_CH9 (GPIO 26)

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 0 V and 4095 to 3.3 V. You can also set the resolution of your channels on the code and the ADC range.

The ADC GPIOs are divided into two types: ADC1 and ADC2.

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 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 2x 8-bit 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) coprocessor 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)

PWM

The ESP32 has an LED PWM controller with 6 to 16 independent channels (depending on the ESP32 model) 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 (optional);
- GPIO where you want to output the signal.

I2C

When using the ESP32 with the Arduino IDE, these are the ESP32 I2C default pins for most ESP32 boards (supported by the `Wire` library):

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

Note: depending on the board model you're using, the default I2C pins might be different. You can check your board's I2C pins by running the following code:
- [Click here to download the code.](#)

SPI

The ESP32 integrates 4 SPI peripherals: SPI0, SPI1, SPI2 (commonly referred to as HSPI), and SPI3 (commonly referred to as VSPI).

SPI0 and SPI1 are used internally to communicate with the built-in flash memory, and you should not use them for other tasks.

You can use **HSPI** and **VSPI** to communicate with other devices. HSPI and VSPI have independent bus signals, and each bus can drive up to three SPI slaves.

Many ESP32 boards come with default SPI pins pre-assigned. The pin mapping for most boards is as follows:

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

Note: depending on the board you're using, the default SPI pins might be different. You can check your board's SPI pins by running the following code:
- [Click here to download the code.](#)

Interrupts

All GPIOs can be configured as interrupts.

UART Pins – Serial Communication

The ESP32 supports multiple UART (Universal Asynchronous Receiver-Transmitter) interfaces that allow serial communication with various devices. The ESP32

supports up to three UART interfaces: **UART0**, **UART1**, and **UART2**, depending on the ESP32 board model you're using.

Like I2C and SPI, these UARTs can be mapped to any GPIO pin, although they have default pin assignments on most board models.

The following table shows the default **UART0**, **UART1**, and **UART2** RX and TX pins for the ESP32:

UART Port	TX	RX
UART0	GPIO 1	GPIO 3
UART1	GPIO 10*	GPIO 9*
UART2	GPIO 17	GPIO 16

*** Note:** **UART1** uses pins connected to the SPI Flash memory. So, you'll need to reassign those pins if you want to use UART 1.

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 uploading 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 having 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.

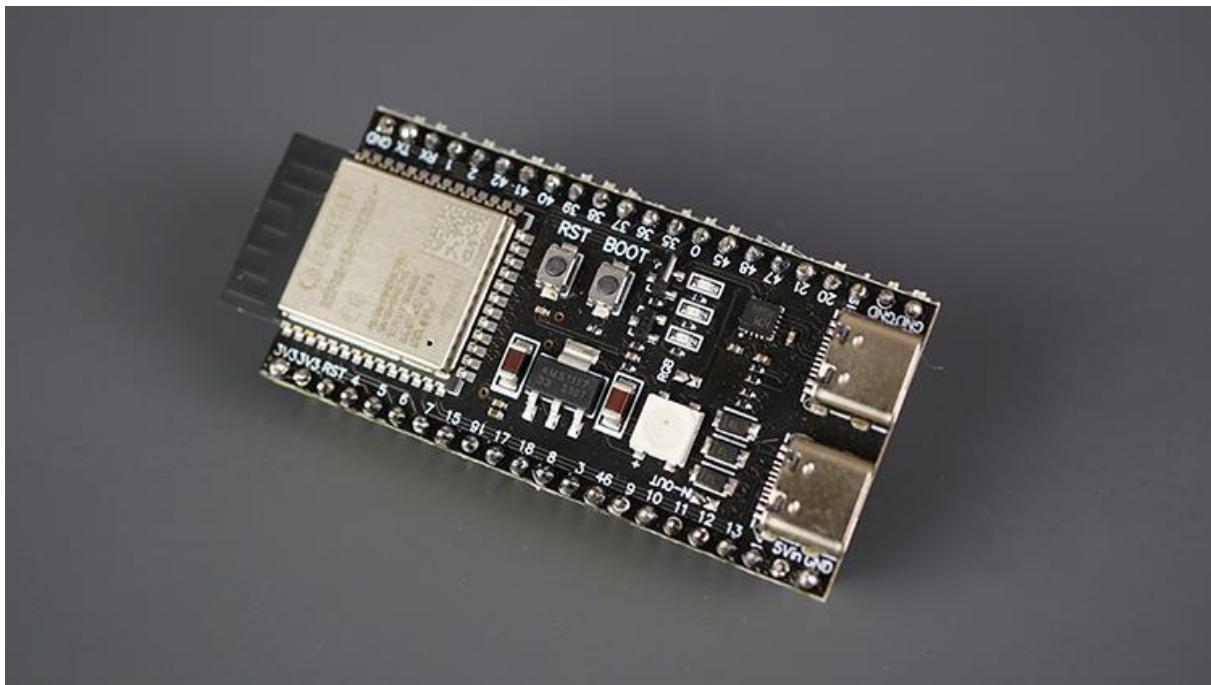
- GPIO 1
- GPIO 3
- GPIO 5
- GPIO 6 to GPIO 11 (connected to the ESP32 integrated SPI flash memory – not recommended to use).
- GPIO 14
- GPIO 15

Enable (EN)

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

2.2 - ESP32-S3 Pinout

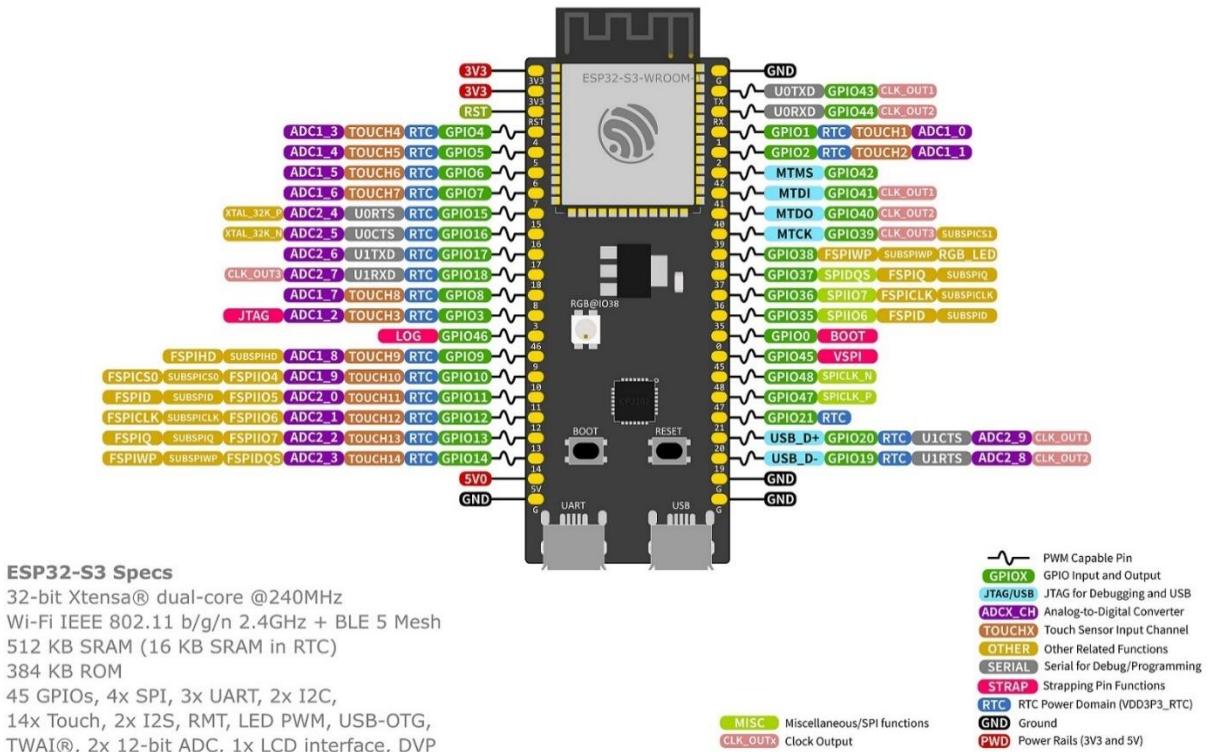
If you're using an ESP32 development board with the ESP32-S3 chip, it has a different pinout and the default pins for I2C, SPI, UART are different. You need to take that into account if you're using an ESP32 board with an S3 chip.



ESP32-S3 DevKitC-1 Pinout

The following picture shows the pinout of the ESP32-S3 DevKitC-1 board, one of the most popular development boards with the S3 chip. There are different versions of the same board with slightly different pinouts. Always double-check the pin location and the GPIO label before connecting any peripherals.

- [Click here to download the pinout \(.png\).](#)



For more information about this board, check the [official documentation here](#).

SPI Flash and PSRAM

GPIOs 26 to 32 are connected to the integrated SPI flash and PSRAM and are not recommended for other uses. They are not exposed in this particular board, but if they are exposed on your board, avoid using them:

- GPIO 26 (Flash/PSRAM SPICS1)
- GPIO 27 (Flash/PSRAM SPIHD)
- GPIO 28 (Flash/PSRAM SPIWP)
- GPIO 29 (Flash/PSRAM SPICS0)
- GPIO 30 (Flash/PSRAM SPICLK)
- GPIO 31 (Flash/PSRAM SPIQ)
- GPIO 32 (Flash/PSRAM SPID)

Capacitive touch GPIOs

The ESP32-S3 has 14 internal capacitive touch GPIOs. 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:

- T1 (GPIO 1)
- T2 (GPIO 2)
- T3 (GPIO 3)
- T4 (GPIO 4)
- T5 (GPIO 5)
- T6 (GPIO 6)
- T7 (GPIO 7)
- T8 (GPIO 8)
- T9 (GPIO 9)
- T10 (GPIO 10)
- T11 (GPIO 11)
- T12 (GPIO 12)
- T13 (GPIO 13)
- T14 (GPIO 14)

Analog to Digital Converter (ADC)

The ESP32 has 20x 12-bit ADC input channels. These are the GPIOs that can be used as ADC and respective channels:

- ADC1_CH0 (GPIO 1)
- ADC1_CH1 (GPIO 2)
- ADC1_CH2 (GPIO 3)
- ADC1_CH3 (GPIO 4)
- ADC1_CH4 (GPIO 5)
- ADC1_CH5 (GPIO 6)
- ADC1_CH6 (GPIO 7)
- ADC1_CH7 (GPIO 8)

- ADC1_CH8 (GPIO 9)
- ADC1_CH9 (GPIO 10)
- ADC2_CH0 (GPIO 11)
- ADC2_CH1 (GPIO 12)
- ADC2_CH2 (GPIO 13)
- ADC2_CH3 (GPIO 14)
- ADC2_CH4 (GPIO 15)
- ADC2_CH5 (GPIO 16)
- ADC2_CH6 (GPIO 17)
- ADC2_CH7 (GPIO 18)
- ADC2_CH8 (GPIO 19)
- ADC2_CH9 (GPIO 20)

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 0 V and 4095 to 3.3 V. You can also set the resolution of your channels on the code and the ADC range.

Digital to Analog Converter (DAC)

ESP32-S3 doesn't come with DAC pins.

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) coprocessor is running. The following GPIOs can be used as an external wake-up source.

- RTC_GPIO0 (GPIO0)
- RTC_GPIO1 (GPIO1)
- RTC_GPIO2 (GPIO2)
- RTC_GPIO3 (GPIO3)

- RTC_GPIO4 (GPIO4)
- RTC_GPIO5 (GPIO5)
- RTC_GPIO6 (GPIO6)
- RTC_GPIO7 (GPIO7)
- RTC_GPIO8 (GPIO8)
- RTC_GPIO9 (GPIO9)
- RTC_GPIO10 (GPIO10)
- RTC_GPIO11 (GPIO11)
- RTC_GPIO12 (GPIO12)
- RTC_GPIO13 (GPIO13)
- RTC_GPIO14 (GPIO14)
- RTC_GPIO15 (GPIO15)
- RTC_GPIO16 (GPIO16)
- RTC_GPIO17 (GPIO17)
- RTC_GPIO18 (GPIO18)
- RTC_GPIO19 (GPIO19)
- RTC_GPIO20 (GPIO20)
- RTC_GPIO21 (GPIO21)

PWM

The ESP32-S3 has an LED PWM controller with 8 PWM 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.

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

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

I2C

When using the ESP32-S3 with the Arduino IDE, these are the ESP32 I2C default pins:

- GPIO 8 (SDA)
- GPIO 9 (SCL)

Note: depending on the board model you're using, the default I2C pins might be different. You can check your board's I2C pins by running the following code:
- [Click here to download the code.](#)

SPI

The ESP32 integrates 4 SPI peripherals: SPI0, SPI1, SPI2 (commonly referred to as HSPI), and SPI3 (commonly referred to as VSPI).

SPI0 and SPI1 are used internally to communicate with the built-in flash memory, and you should not use them for other tasks.

You can use **HSPI** and **VSPI** to communicate with other devices. HSPI and VSPi have independent bus signals.

SPI	MOSI	MISO	CLK	CS
HSPI (SPI 2)	GPIO 35	GPIO 13	GPIO 12	GPIO 10
VSPi (SPI 3)	GPIO 35	GPIO 37	GPIO 36	GPIO 39

Note: depending on the board you're using, the default SPI pins might be different. You can check your board's SPI pins by running the following code:
- [Click here to download the code.](#)

Interrupts

All GPIOs can be configured as interrupts.

UART Pins – Serial Communication

The ESP32-S3 supports multiple UART (Universal Asynchronous Receiver-Transmitter) interfaces that allow serial communication with various devices. The ESP32 supports up to three UART interfaces: **UART0**, **UART1**, and **UART2**, depending on the ESP32 board model you're using.

Like I2C and SPI, these UARTs can be mapped to any GPIO pin, although they have default pin assignments on most board models.

The following table shows the default UART0, UART1, and UART2 RX and TX pins for the ESP32-S3:

UART Port	TX	RX	Remarks
UART0	GPIO 43	GPIO 44	Cannot be changed
UART1	GPIO 17	GPIO 18	Can be assigned to other GPIOs
UART2	--	--	Assign any pins of your choice

Strapping Pins

The ESP32-S3 chip has the following strapping pins:

- GPIO 0
- GPIO 3
- GPIO 45
- GPIO 46

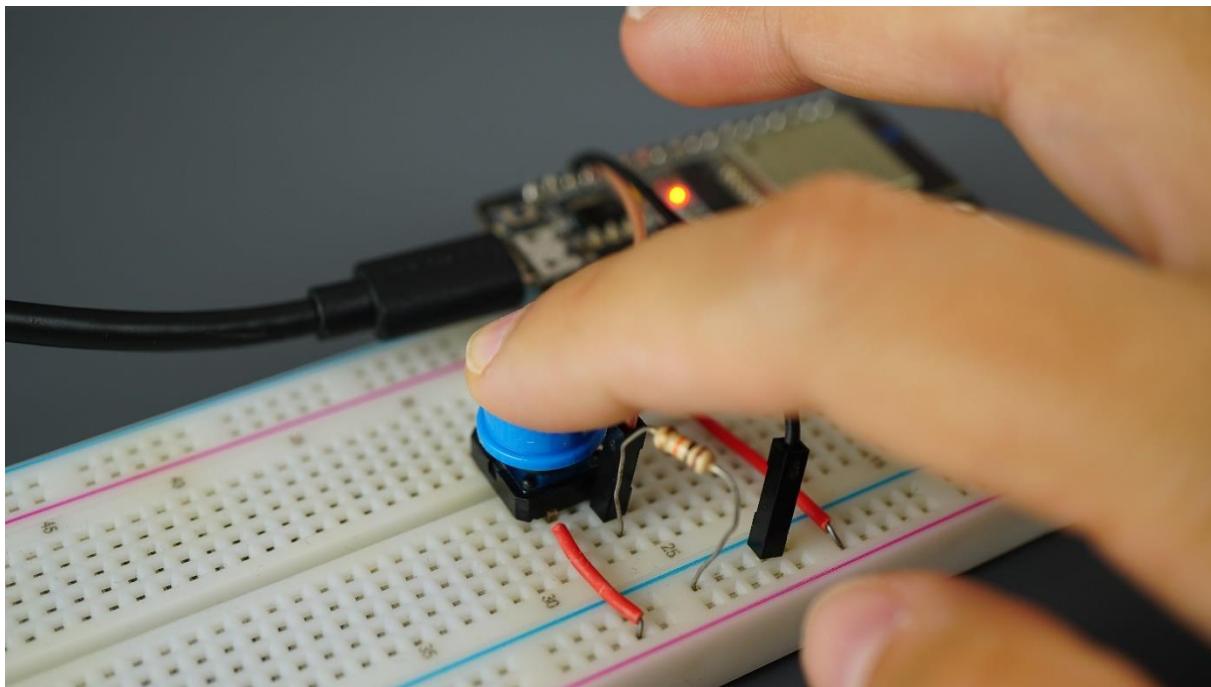
These pins are used to put the ESP32 into bootloader or flashing mode. On most development boards with built-in USB/Serial, you don't need to worry about the state of these pins. The board puts the pins in the right state for flashing or boot mode. However, you should avoid using these pins on your projects.

Enable (EN/RST)

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

2.3 - ESP32 Digital Inputs and Outputs

This section demonstrates how to read digital inputs, such as a button switch, and control digital outputs, like an LED. If you've previously programmed Arduino or ESP8266 boards with the Arduino IDE, this topic will be familiar to you.



Controlling Outputs: `digitalWrite()`

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

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

Reading Inputs: `digitalRead()`

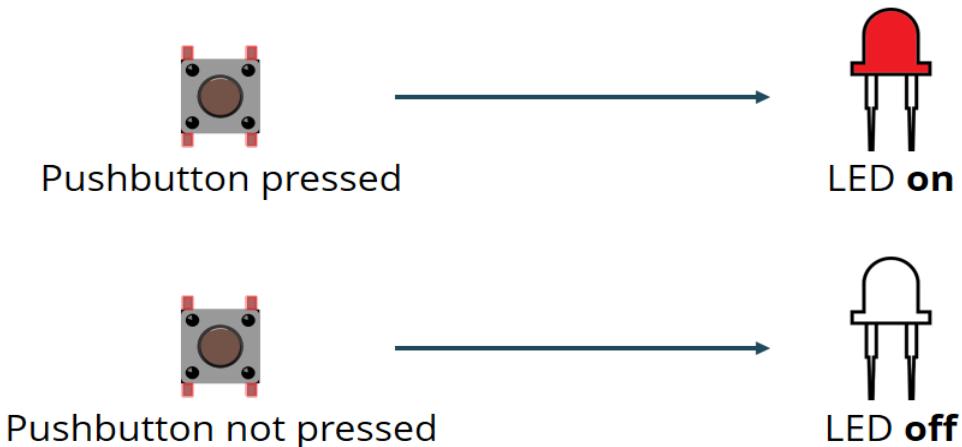
To read a digital input, like a button, you use the `digitalRead()` function. This function accepts `as` an argument the GPIO you are referring to.

```
digitalRead(buttonPin);
```

This function returns `0`, if the GPIO is `LOW`, or `1` if the GPIO is `HIGH`.

Project Example

Let's make a simple example to see how those functions work with the ESP32 using the Arduino IDE. In this example, you'll read the state of a pushbutton and light up an LED accordingly.



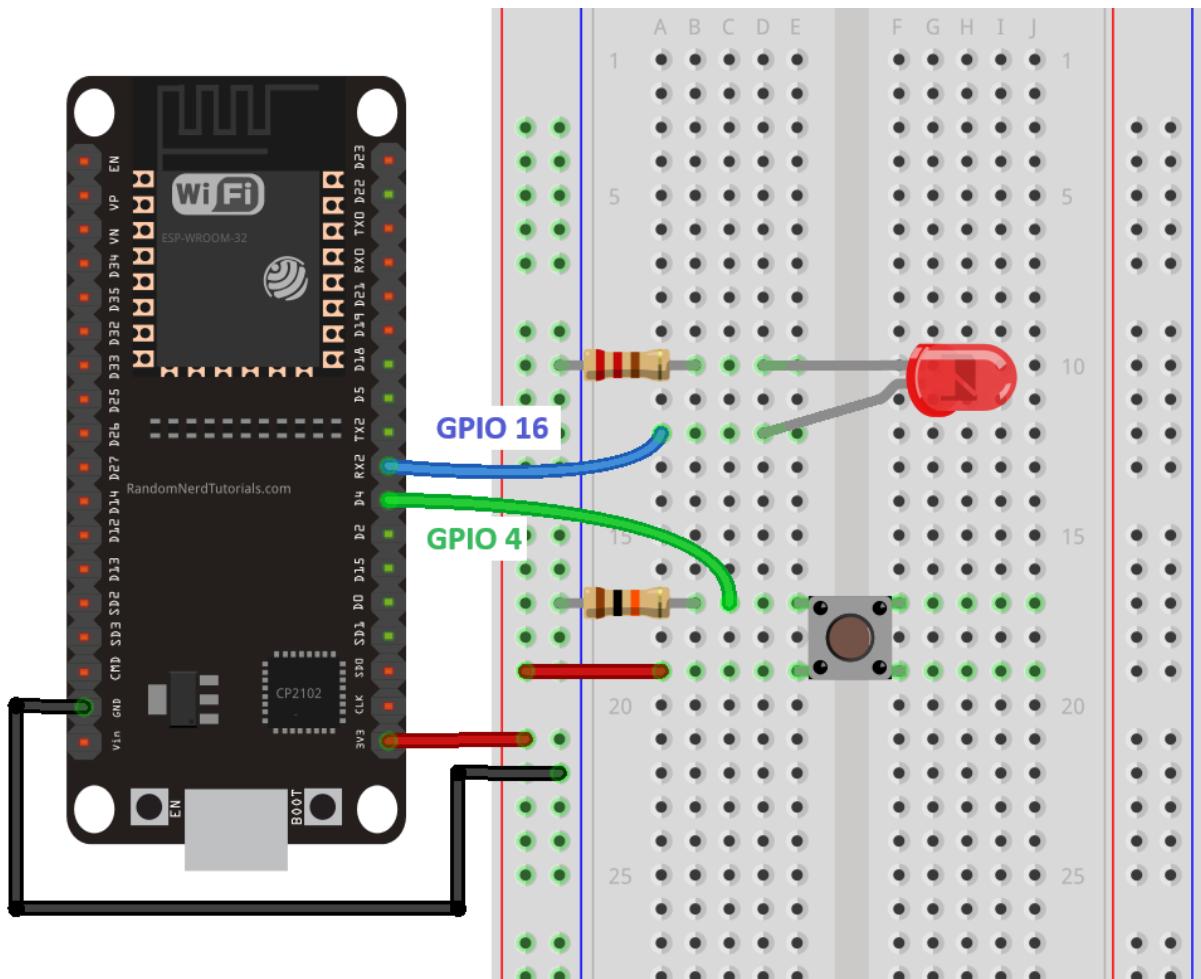
Wiring the Circuit

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [220 Ohm resistor or similar value](#)
- [Pushbutton](#)
- [10k Ohm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)

Assemble a circuit with a pushbutton and an LED as shown in the following schematic diagram:

- LED: GPIO 16 (220 Ohm resistor)
- Pushbutton: GPIO 4 (10K Ohm resistor)



Code

With the circuit ready, copy the code provided below to your Arduino IDE.

- [Click here to download the code](#)

```
// set pin numbers
const int buttonPin = 4;      // the number of the pushbutton pin
const int ledPin = 16;        // 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 {  
    // turn LED off  
    digitalWrite(ledPin, LOW);  
}  
}
```

How Does the Code Work?

Let's take a closer look at the code.

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

```
const int buttonPin = 4;      // the number of the pushbutton pin  
const int ledPin = 16;        // the number of the LED pin
```

The button is connected to GPIO 4, and the LED is connected to GPIO 16. When using the Arduino IDE with the ESP32, 4 corresponds to GPIO 4, and 16 corresponds to GPIO 16.

Next, you create a variable to hold the button state.

```
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—either `INPUT` or `OUTPUT`.

```
// initialize the pushbutton pin as an input  
pinMode(buttonPin, INPUT);  
// initialize the LED pin as an output  
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 need to 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 arguments the `ledPin`, and the state, `HIGH`.

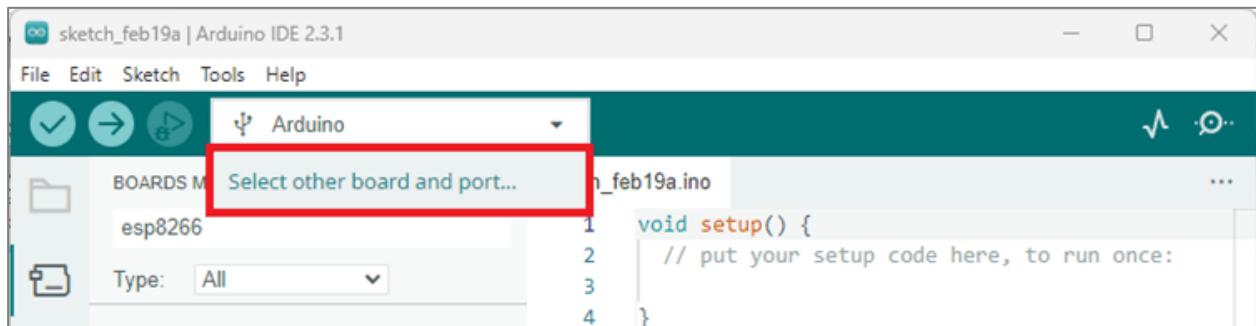
```
if (buttonState == HIGH) {  
    // turn LED on  
    digitalWrite(ledPin, HIGH);  
}
```

If the button state is not `HIGH`, you set the LED off, by writing `LOW` in the `digitalWrite()` function.

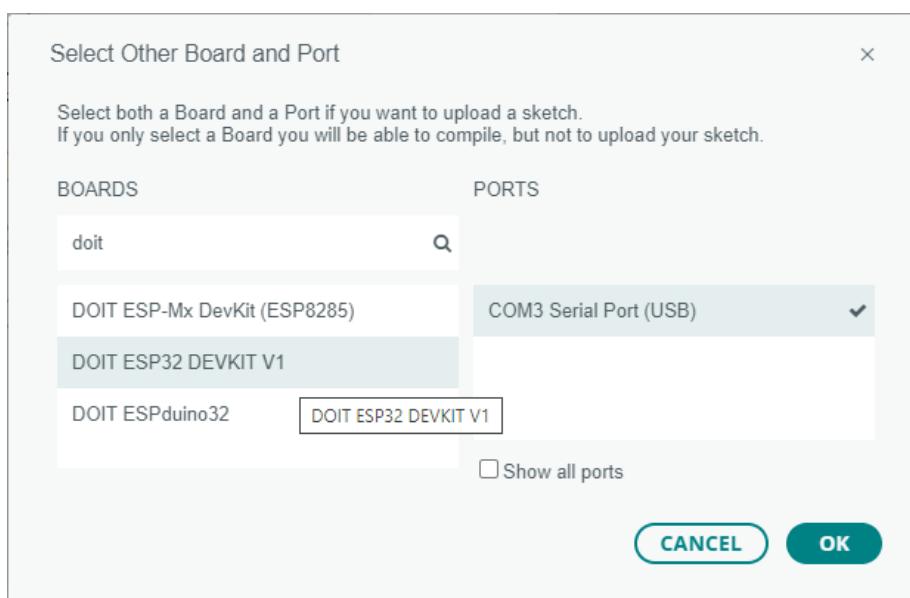
```
else {  
    // turn LED off  
    digitalWrite(ledPin, LOW);  
}
```

Uploading the Sketch

Select your Board in **Tools > Board** menu or on the top drop-down menu, click on "Select other board and port..."



A new window, as shown below, will open. Search for your ESP32 board model.



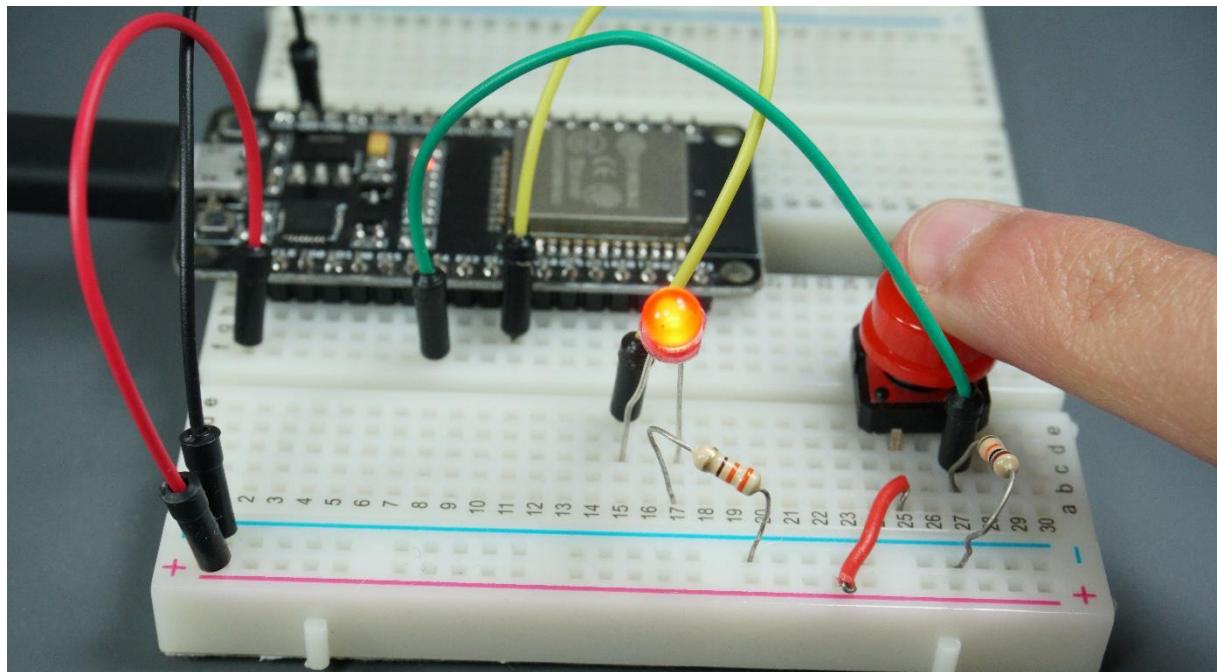
Select the board model you're using, and the COM port. In our example, we're using the DOIT ESP32 DEVKIT V1. Click **OK** when you're done.

Click on the Arduino IDE upload button, and wait a few seconds while it compiles and uploads your sketch to the board.



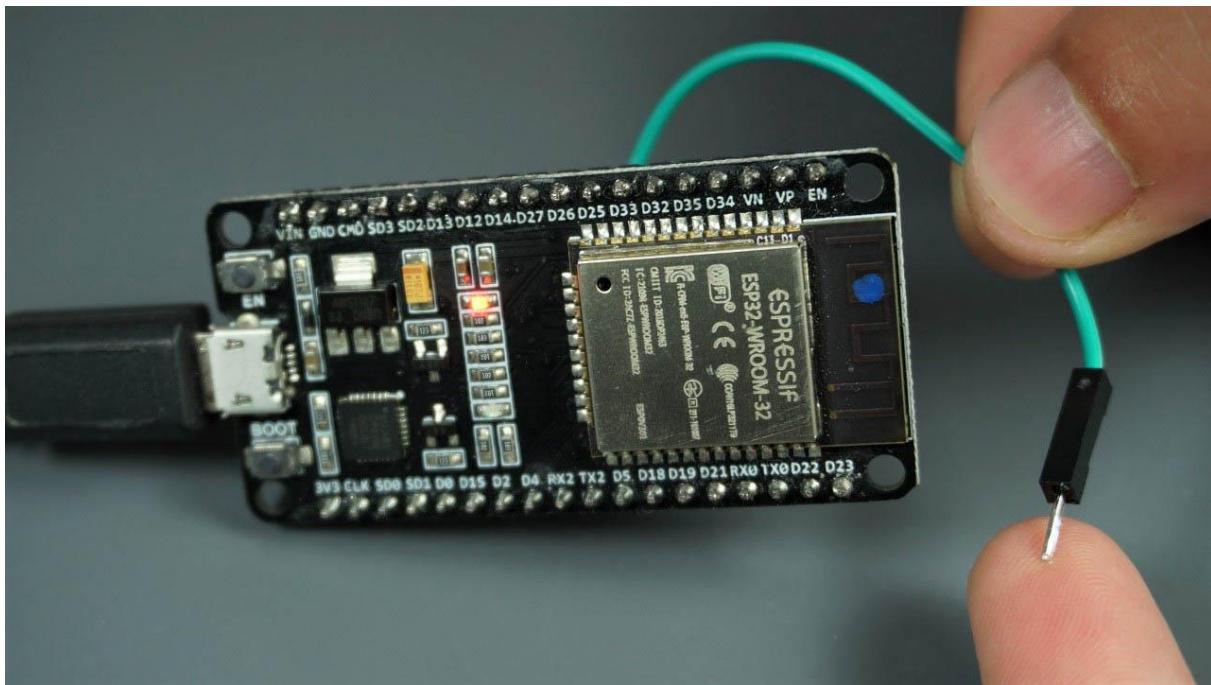
Testing Your Project

After uploading the code, test your circuit. Your LED should light up when you press the pushbutton and turn off when you release it.



2.4 – Capacitive Touch Pins

This Unit shows how to use the ESP32 touch pins. The ESP32 touch pins can sense variations in anything that holds an electrical charge. They are often used to wake up the ESP32 from deep sleep (we'll cover this later in this eBook).



Introducing the ESP32 Touch Pins

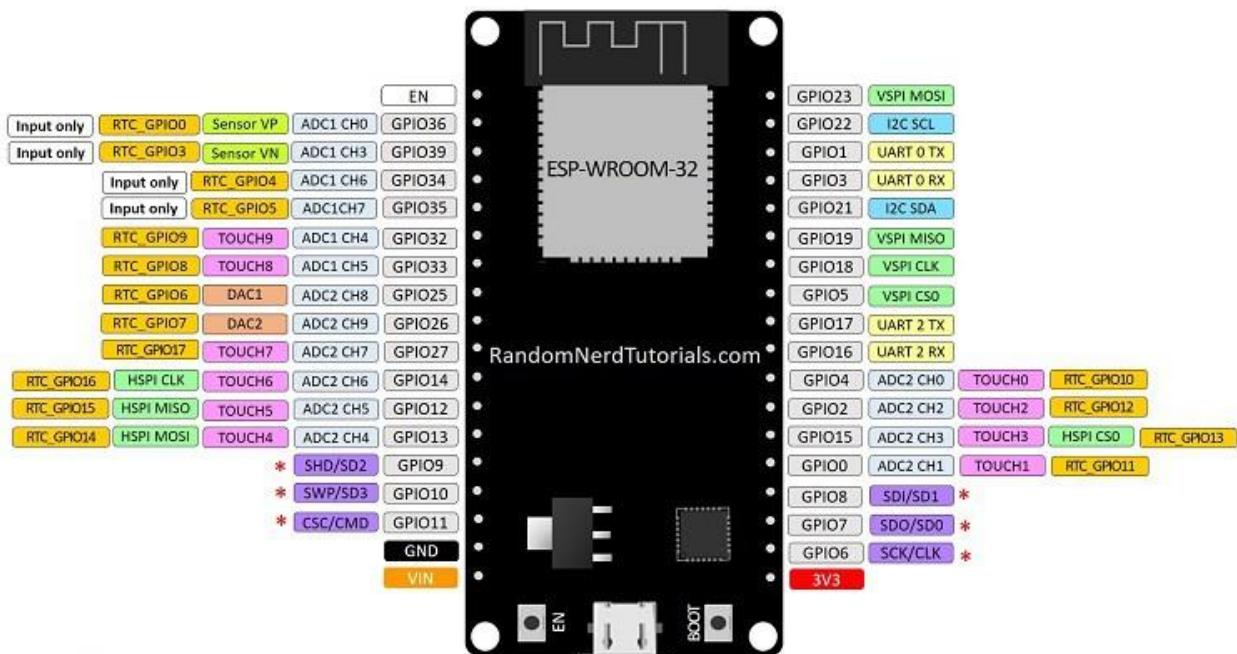
The ESP32 has ten (or more depending on the chip model) capacitive touch GPIOs. These GPIOs can sense variations in anything that holds an electrical charge, like the human skin. So, they can detect variations induced when touching the GPIOs with a finger.

These pins can be easily integrated into capacitive pads and replace mechanical buttons. Additionally, the touch pins can also be used as a wake-up source when the ESP32 is in deep sleep.

Take a look at your board pinout to locate the ten different touch-sensitive pins—highlighted in pink color.

ESP32 DEVKIT V1 – DOIT

version with 36 GPIOs



* Pins SCK/CLK, SDO/SD0, SDI/SD1, SHD/SD2, SWP/SD3 and SCS/CMD, namely, GPIO6 to GPIO11 are connected to the integrated SPI flash integrated on ESP-WROOM-32 and are not recommended for other uses.

You can see that touch sensor 0 corresponds to GPIO 4, touch sensor 2 to GPIO 2, and so on.

The touchRead() Function

Reading the touch pins is straightforward. In the Arduino IDE, you use the `touchRead()` function, which accepts the GPIO you want to read as an argument.

`touchRead(GPIO)`

Code - Reading the Touch Sensor

Let's see how that function works with a simple example (this was adapted from an example available in the Arduino IDE).

- [Click here to download the code](#)

```
// ESP32 Touch Test
// Just test touch pin - Touch0 is T0 which is on GPIO 4.

void setup() {
```

```
Serial.begin(115200);
delay(1000); // give me time to bring up serial monitor
Serial.println("ESP32 Touch Test");
}

void loop() {
  Serial.println(touchRead(T0)); // get value using T0
  delay(1000);
}
```

This example reads the touch pin 0 and displays the results in the serial monitor. The T0 pin (touch pin 0) corresponds to GPIO 4, as we've seen previously in the pinout.

In this code, in the `setup()`, you start by initializing the Serial Monitor to display the sensor readings.

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

In the `loop()` is where you read the touch sensor value on GPIO 4.

```
Serial.println(touchRead(T0)); // get value using T0
```

It uses the `touchRead()` function, and you pass as an argument the pin you want to read. In this case, the example uses T0, the touch sensor 0, in GPIO 4. You can either pass `T0` or `4`.

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

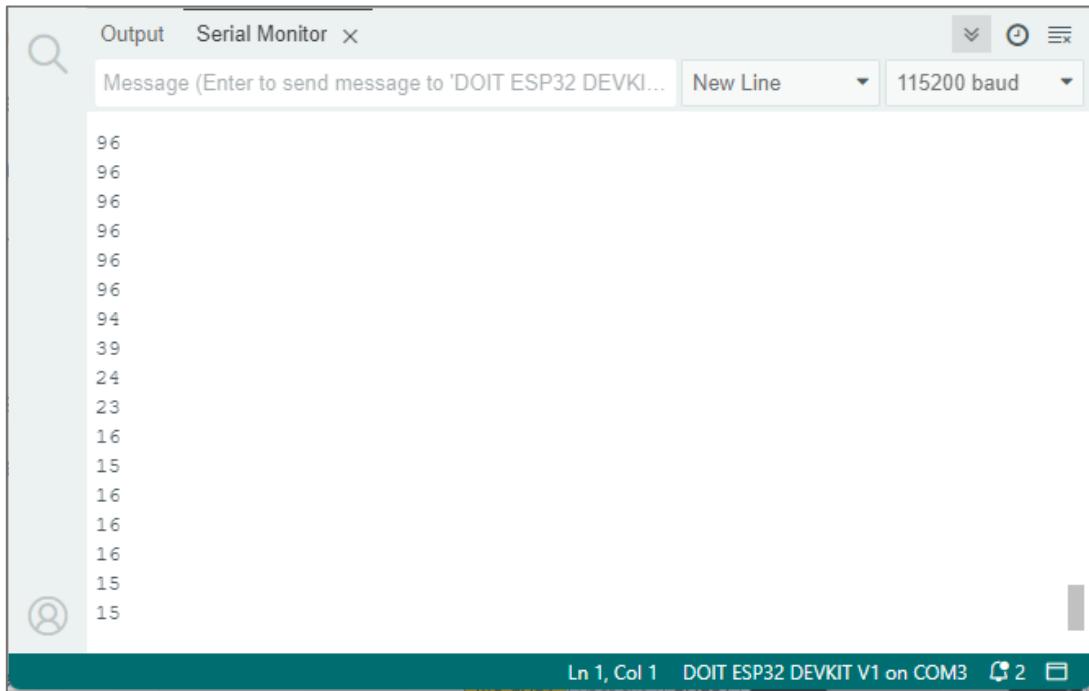
Testing the Example

Open the **Serial Monitor** at a baud rate of 115200.



Connect a jumper wire to GPIO 4. You will touch the metal part of this wire so that it senses the touch. You'll see the new values displayed every second.

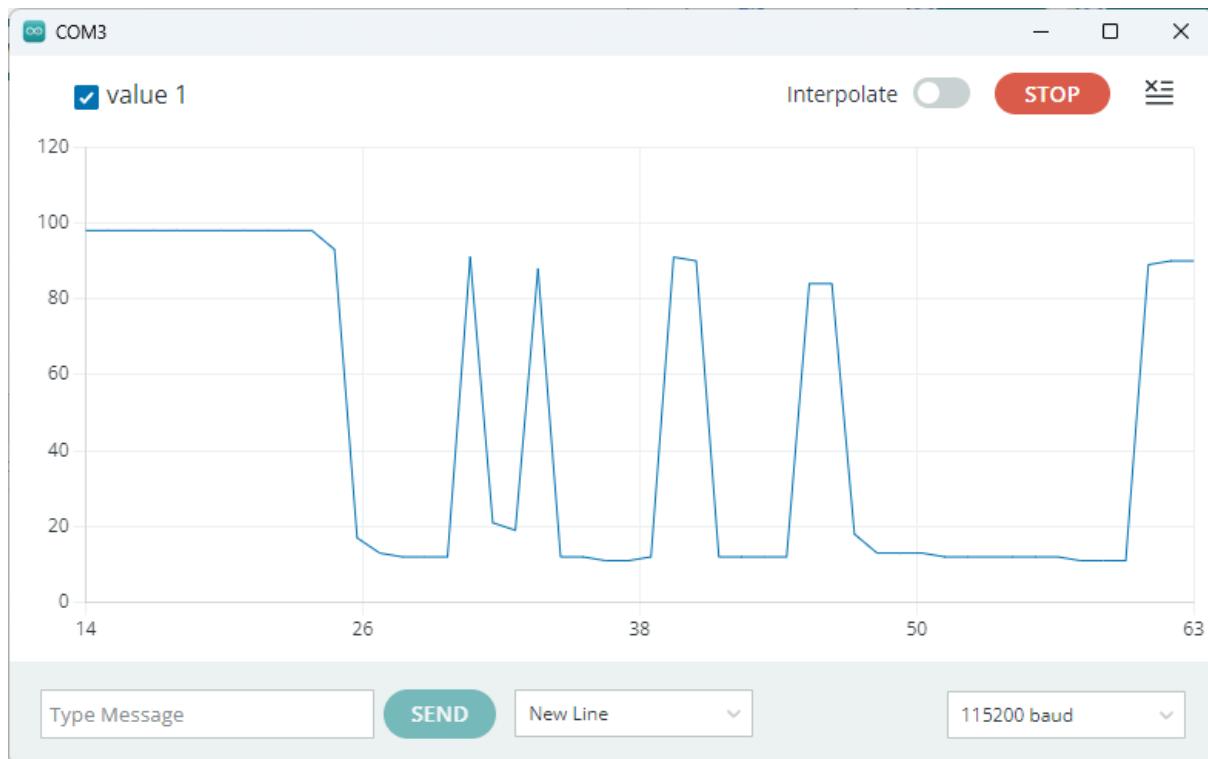
When you touch the wire, you'll see the values decreasing.



You can also use the serial plotter to better see the values. Open the Serial Plotter by clicking on its icon (it's right next to the Serial Monitor icon).



The following window will open and the values will be displayed on the plotter over time.



Touch Sensitive LED

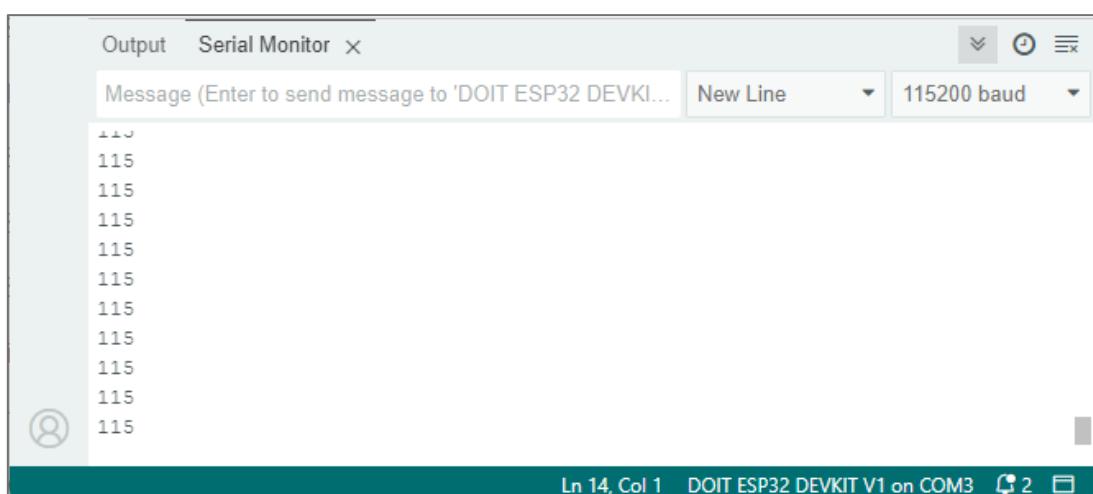
You can use this feature to control outputs. In this example, we'll build a simple touch-controlled LED circuit. When you touch the GPIO with your finger, the LED lights up.

Finding the Threshold

Grab a piece of aluminum foil, cut a small square, and wrap it around the wire, as shown in the following figure.



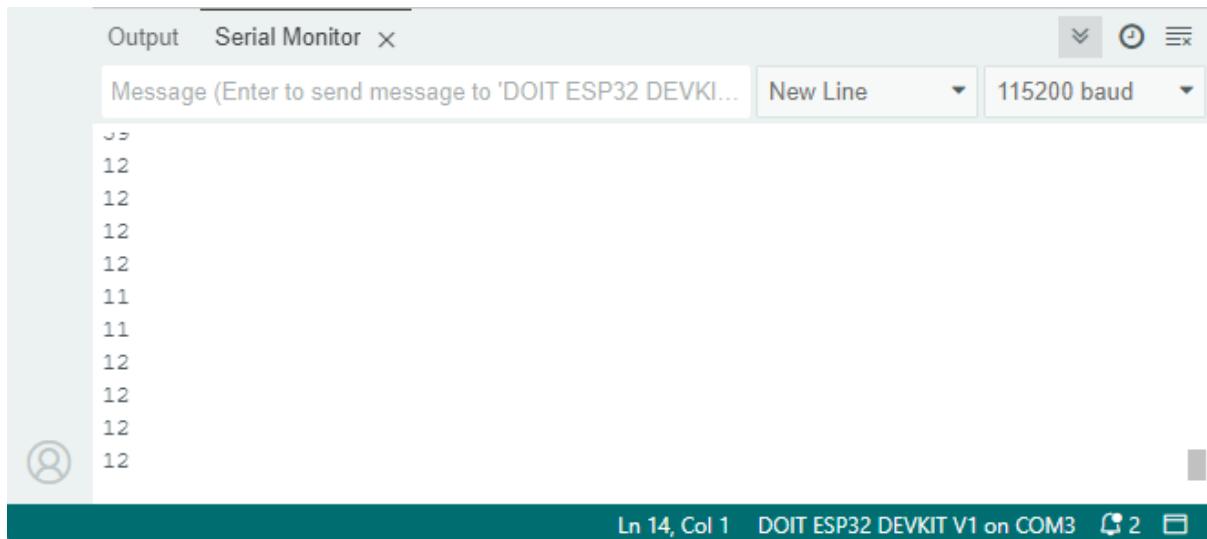
With the same code running, go back to the serial monitor. Touch the aluminum foil, and you'll see the values changing. In our case, when we aren't touching the pin, the typical value for GPIO 4 is above around 115.



The screenshot shows the Arduino Serial Monitor window. The title bar says "Output" and "Serial Monitor". The message input field contains "Message (Enter to send message to 'DOIT ESP32 DEVKIT...')". The baud rate is set to "115200 baud". The main area displays a series of "115" characters, indicating the value read from GPIO 4 when no touch is detected. The bottom status bar shows "Ln 14, Col 1" and "DOIT ESP32 DEVKIT V1 on COM3".

```
115
115
115
115
115
115
115
115
115
115
```

And when we touch the aluminum foil, it drops to some value around 12.



The screenshot shows the Arduino Serial Monitor window. The title bar says "Output" and "Serial Monitor". The main area displays the text "Message (Enter to send message to 'DOIT ESP32 DEVKIT V1...' New Line 115200 baud". Below this, there is a list of the number "12" repeated multiple times. At the bottom of the window, there is a status bar with the text "Ln 14, Col 1 DOIT ESP32 DEVKIT V1 on COM3" and icons for a bell and a refresh.

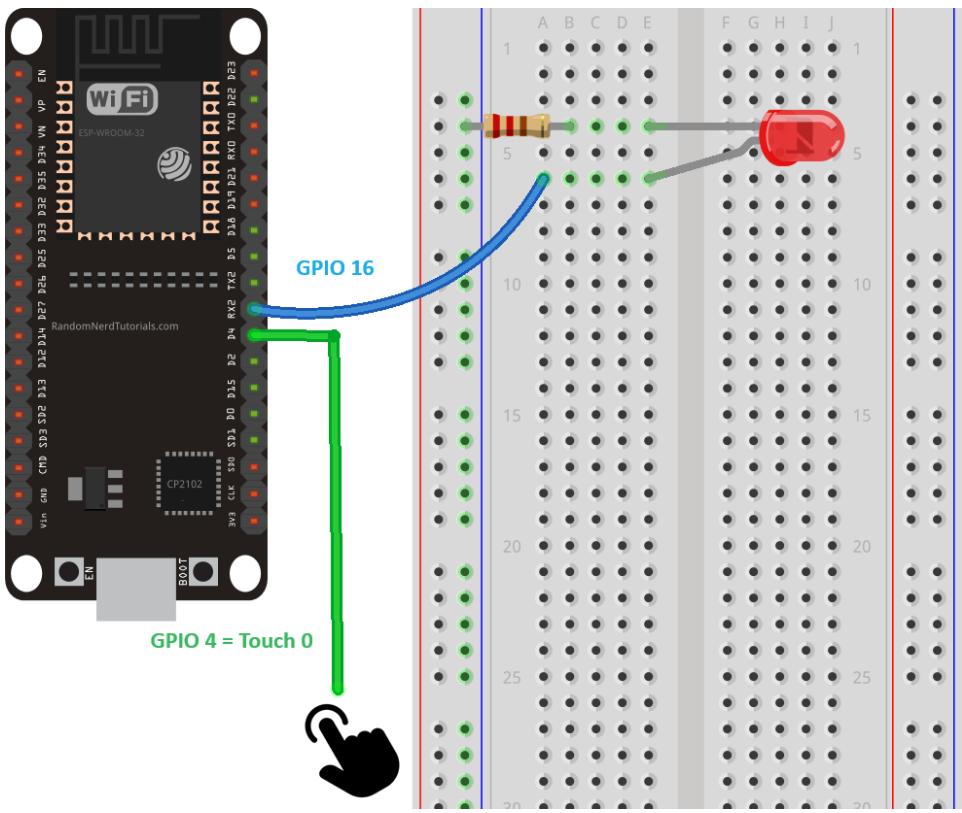
Taking these values into account, we can set a threshold value, and when the reading goes below that value, an LED lights up. A good threshold value, in this case, is 20, for example—yours might be different.

Wiring the Circuit

Add an LED to your circuit by following the next diagram. The LED should be connected to GPIO 16.

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [220 Ohm resistor \(or similar value\)](#)
- [Breadboard](#)
- [Jumper wires](#)



Code

Copy the following code to your Arduino IDE.

- [Click here to download the code](#)

```
// set pin numbers
const int touchPin = 4;
const int ledPin = 16;

// change with your threshold value
const int threshold = 20;
// variable for storing the touch pin value
int touchValue;

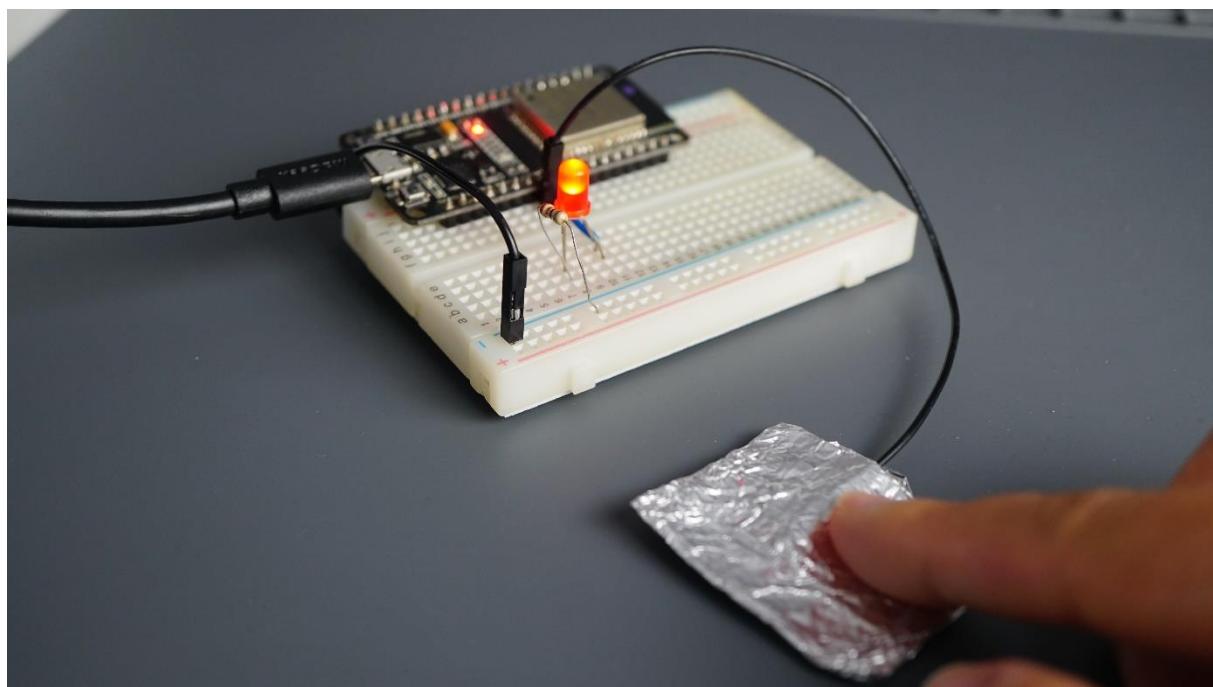
void setup(){
  Serial.begin(115200);
  delay(1000); // give me time to bring up serial monitor
  // initialize the LED pin as an output:
  pinMode (ledPin, OUTPUT);
}
void loop(){
  // read the state of the pushbutton value:
  touchValue = touchRead(touchPin);
  Serial.print(touchValue);
  // check if the touchValue is below the threshold
  // if it is, set ledPin to HIGH
  if(touchValue < threshold){
    // turn LED on
    digitalWrite(ledPin, HIGH);
    Serial.println(" - LED on");
  }
  else{
```

```
// turn LED off  
digitalWrite(ledPin, LOW);  
Serial.println(" - LED off");  
}  
delay(500);  
}
```

This code reads the touch value from the pin we've defined and lights up an LED when the value is below the threshold (this means when you place your finger in the aluminum pad).

Testing the Project

Upload the sketch to your ESP32. Now, test your circuit. Touch the aluminum foil and see the LED lighting up.



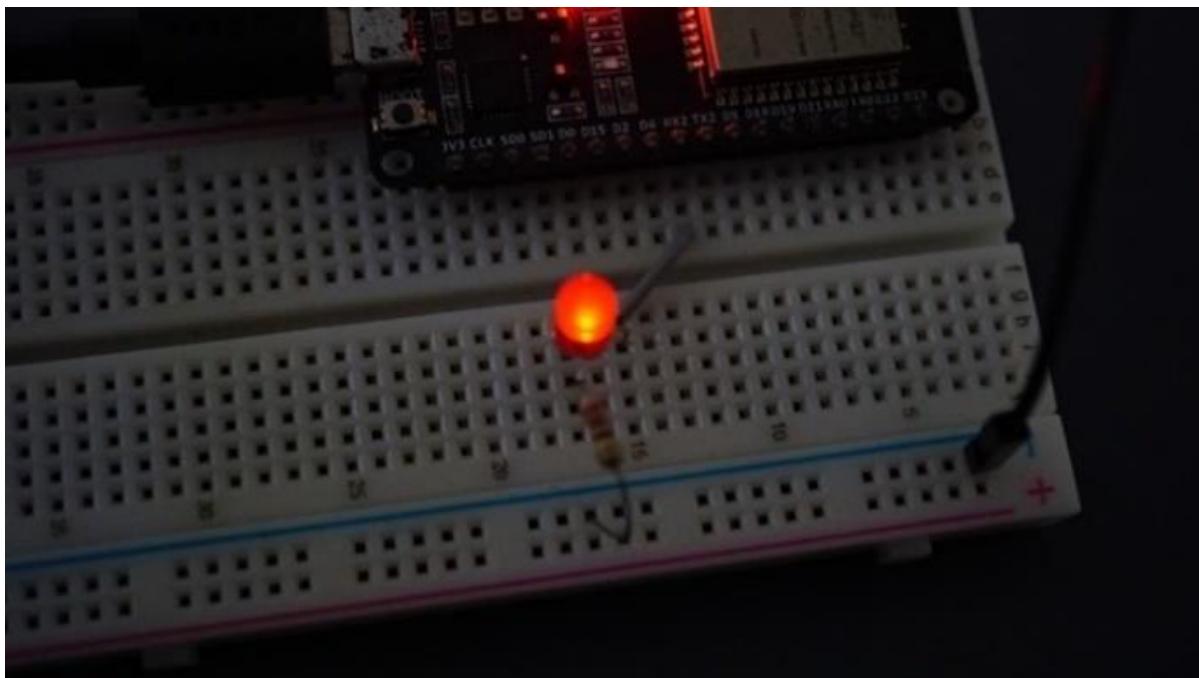
Wrapping Up

In summary, in this section, you've learned that:

- The ESP32 has ten or more capacitive touch GPIOs. In some boards, not all are physically available.
- When you touch a touch-sensitive GPIO, the value read by the sensor drops.
- You can set a threshold value to make something happen when it detects touch.

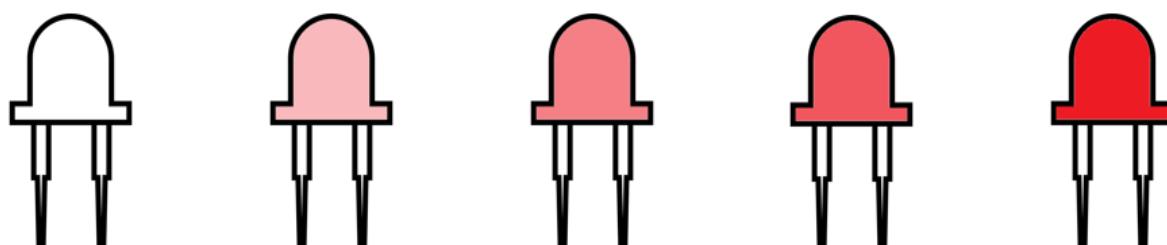
2.5 – ESP32 Pulse-Width Modulation (PWM)

In this section, you'll learn how to dim an LED using PWM. We'll explain two different methods: using `analogWrite` and using the LEDC controller of the ESP32. As an example, we'll build a simple circuit to fade an LED.



Pulse-Width Modulation

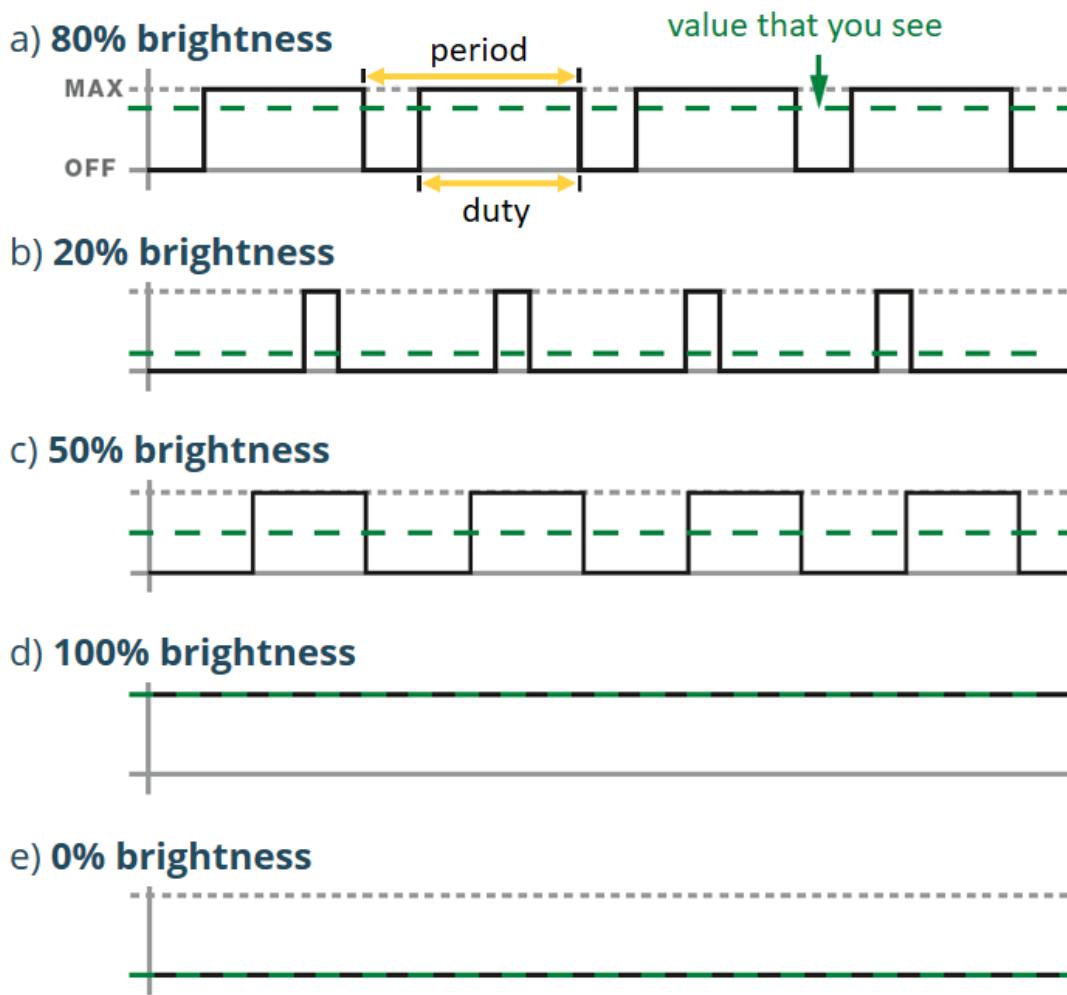
The ESP32 GPIOs can be set either to output 0V or 3.3V, but they can't output any voltages in between (apart from the DAC pins on the ESP32). However, you can output "fake" mid-level voltages using pulse-width modulation (PWM), which is how you'll produce varying levels of LED brightness for this project.



If you alternate an LED's voltage between HIGH and LOW very fast, your eyes can't keep up with the speed at which the LED switches on and off; you'll simply see some gradations in brightness.

That's how PWM works—by producing an output that changes between HIGH and LOW at a very high frequency.

The duty cycle is the fraction of the time at which the LED is set to HIGH. The following figure illustrates how PWM works.



For instance, a duty cycle of 50 percent results in 50 percent LED brightness, a duty cycle of 0 means the LED is fully off, and a duty cycle of 100 means the LED is fully on. Changing the duty cycle is how you produce different levels of brightness, and that's what we're going to do here.

ESP32 LED PWM Controller

The ESP32 has an LED PWM controller with 6 to 16 independent channels (depending on the ESP32 model) that can be configured to generate PWM signals with different properties.

There are different functions you can use to generate PWM signals and achieve the same results. You can use the `analogWrite()` function (like in Arduino boards) or you can use LEDC functions.

The `analogWrite()` Function

The most basic function to generate PWM signals is `analogWrite()` which accepts as arguments the GPIO where you want to generate the PWM signal and the duty cycle value (ranging from 0 to 255).

```
analogWrite(pin, value);
```

For example:

```
analogWrite(2, 180);
```

Set the Frequency and Resolution

You can set the resolution and frequency of the PWM signal on a selected pin by using the `analogWriteResolution()` and `analogWriteFrequency()` functions.

To set the resolution:

```
analogWriteResolution(pin, resolution);
```

To set the frequency:

```
analogWriteFrequency(pin, freq);
```

LEDC Functions

Alternatively, you can use the Arduino-ESP32 LEDC API. First, you need to set up an LEDC pin. You can use the `ledcAttach()` or `ledcAttachChannel()` functions.

ledcAttach

The `ledcAttach()` function sets up an LEDC pin with a given frequency and resolution. The LEDC channel will be selected automatically.

```
bool ledcAttach(uint8_t pin, uint32_t freq, uint8_t resolution);
```

This function will return `true` if the configuration is successful. If `false` is returned, an error occurs and the LEDC channel is not configured.

ledcAttachChannel

If you prefer to set up the LEDC channel manually, you can use the `ledcAttachChannel()` function instead.

```
bool ledcAttachChannel(pin, uint32_t freq, uint8_t resolution, uint8_t channel);
```

This function will return `true` if the configuration is successful. If `false` is returned, an error occurs and the LEDC channel is not configured.

ledcWrite

Finally, after setting the LEDC pin using one of the two previous functions, you use the `ledcWrite` function to set the duty cycle of the PWM signal.

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

This function will return `true` if setting the duty cycle is successful. If `false` is returned, an error occurs and the duty cycle is not set.

For more information and all functions of the LEDC PWM controller, [check the official documentation](#).

Dimming an LED

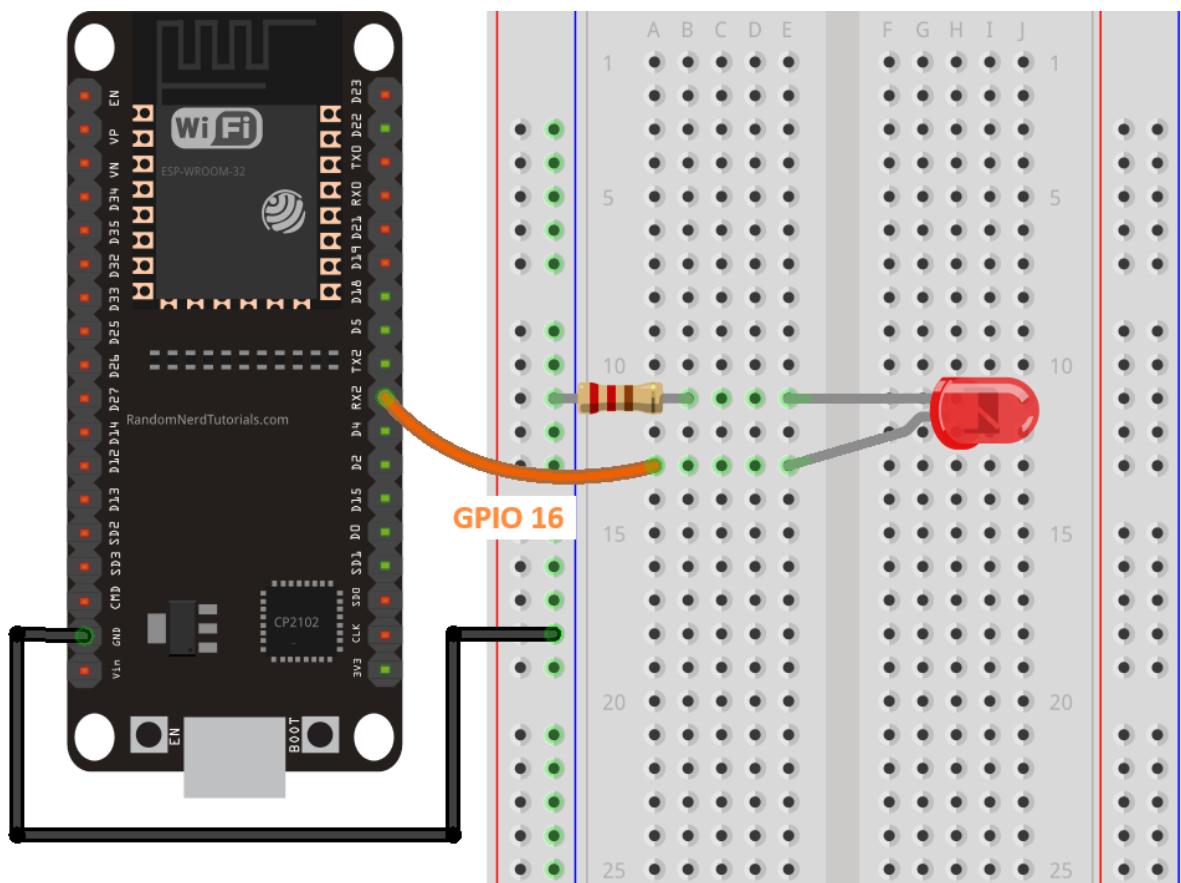
To show you how to generate PWM signals with the ESP32, we'll create two simple examples that dim the brightness of an LED (increase and decrease brightness over time). We'll provide an example using `analogWrite()` and another using the LEDC functions.

Wiring the Circuit

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [220 Ohm resistor \(or similar value\)](#)
- [Breadboard](#)
- [Jumper wires](#)

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



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.

ESP32 PWM Example using analogWrite - Code

Open your Arduino IDE and copy the following code. This example increases and decreases the LED brightness over time using the `analogWrite()` function.

- [Click here to download the code](#)

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

void setup() {
    // set the LED as an output
    pinMode(ledPin, OUTPUT);
}

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

    // decrease the LED brightness
    for(int dutyCycle = 255; dutyCycle >= 0; dutyCycle--){
        // changing the LED brightness with PWM
        analogWrite(ledPin, dutyCycle);
        delay(15);
    }
}
```

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

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

In the `setup()`, you need to configure the LED as an output using the `pinMode()` function.

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

In the `loop()`, you vary the duty cycle between `0` and `255` to increase the LED brightness.

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

Notice the use of the `analogWrite()` function to set up the duty cycle. You just need to pass as arguments the LED pin and the duty cycle.

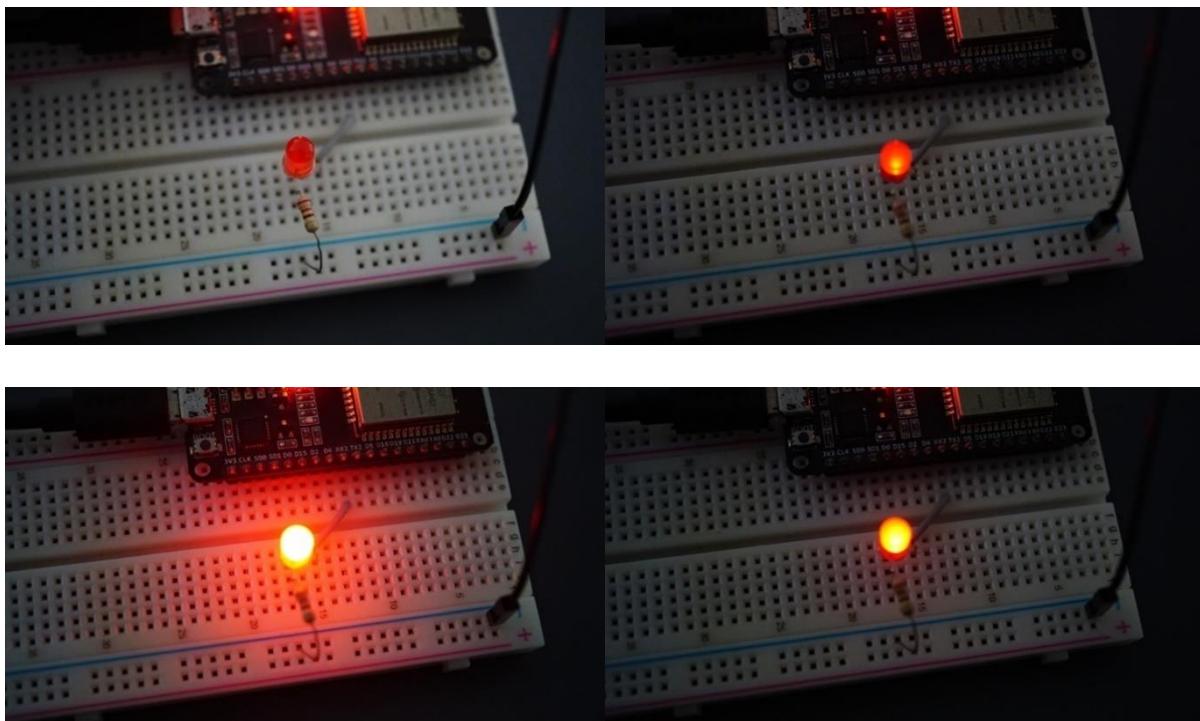
```
analogWrite(ledPin, dutyCycle);
```

Finally, we vary the duty cycle between `255` and `0` to decrease the brightness.

```
// decrease the LED brightness
for(int dutyCycle = 255; dutyCycle >= 0; dutyCycle--){
    // changing the LED brightness with PWM
    analogWrite(ledPin, dutyCycle);
    delay(15);
}
```

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 an LED that increases and decreases brightness over time.



ESP32 PWM Example using the LEDC API - Code

Open your Arduino IDE and copy the following code. This example increases and decreases the LED brightness over time using the ESP32 LEDC functions.

- [Click here to download the code](#)

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

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

void setup(){
    // configure LED PWM
    ledcAttach(ledPin, freq, resolution);
```

```
// if you want to attach a specific channel, use the following instead
//ledcAttachChannel(ledPin, freq, resolution, 0);
}

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

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

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

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

Set the PWM properties: frequency and resolution. As we're using 8-bit resolution, the duty cycle will be controlled using a value from 0 to 255.

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

In the `setup()`, set up the LEDC pin—use the `ledcAttach()` function as follows.

```
// configure LED PWM
ledcAttach(ledPin, freq, resolution);
```

This will configure the LEDC pin with the frequency and resolution defined previously on a default PWM channel.

If you want to set up the PWM channel yourself, you need to use `ledcAttachChannel` instead. The last argument of this function is the PWM channel number.

```
// if you want to attach a specific channel, use the following instead
//ledcAttachChannel(ledPin, freq, resolution, 0);
```

Finally, in the `loop()`, you increase and decrease the brightness of the LED over time. The following lines increase the LED brightness.

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

Notice the use of the `ledcWrite()` function to set up the duty cycle. You just need to pass as arguments the LED pin and the duty cycle.

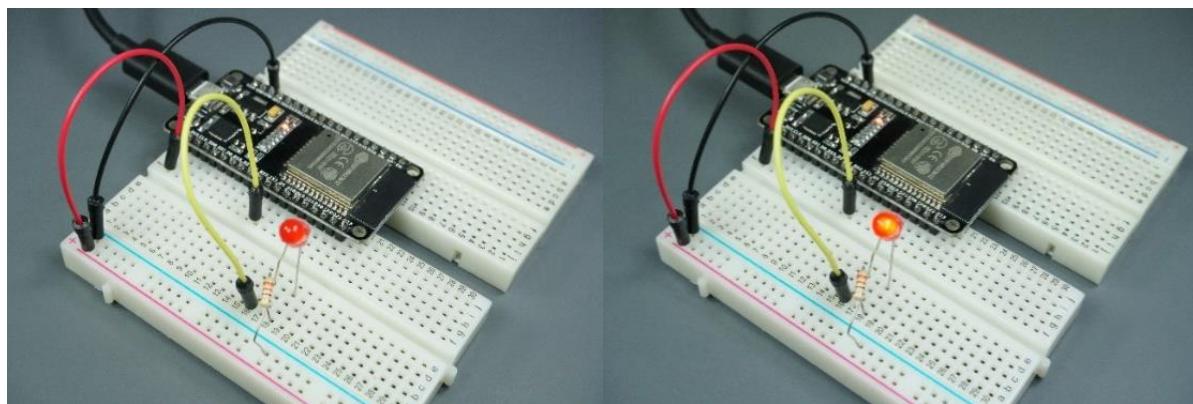
```
ledcWrite(ledPin, dutyCycle);
```

Finally, we vary the duty cycle between `255` and `0` to decrease the brightness.

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

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 an LED that increases and decreases brightness over time just like in the previous example.



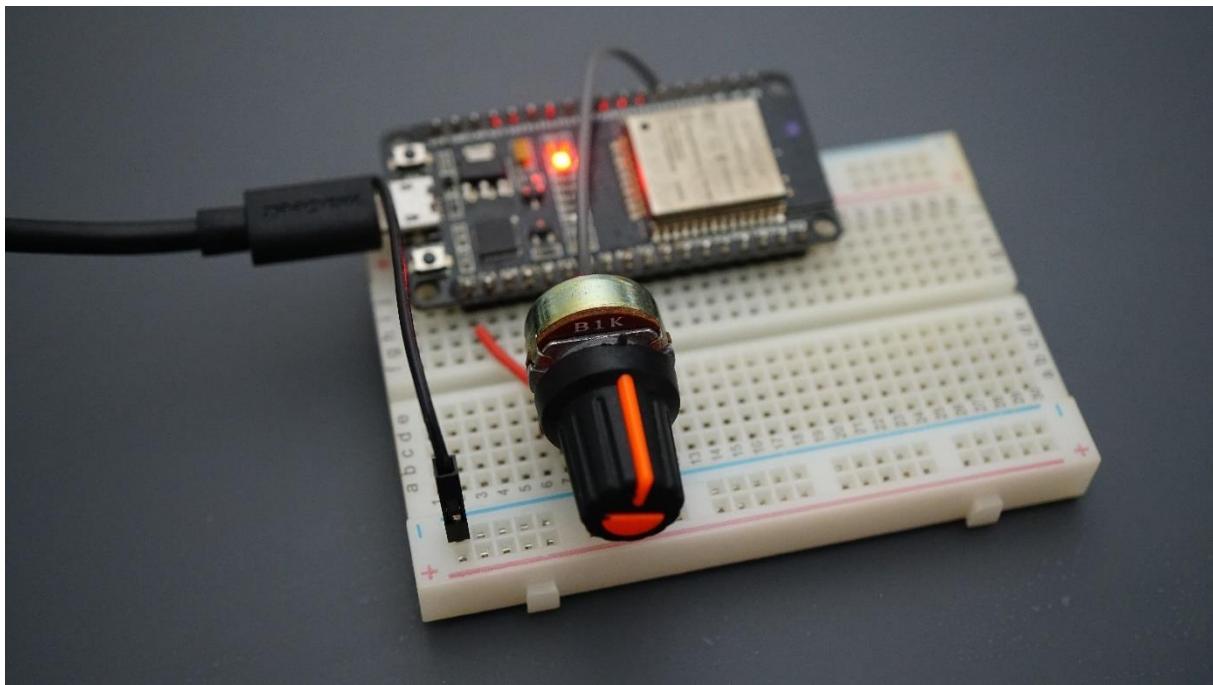
Wrapping Up

In summary, in this article, you learned how to use the LED PWM controller of the ESP32 with the Arduino IDE to dim an LED. You learned two different ways to achieve the same results.

The concepts learned can be used to control other outputs with PWM (like motors, for example) by setting the right properties to the signal.

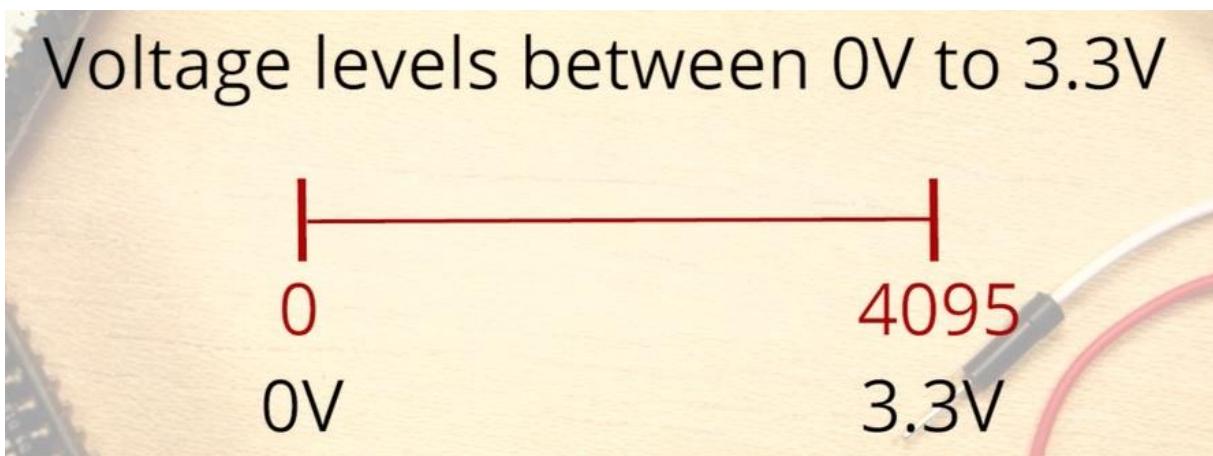
2.6 - Reading Analog Inputs

In this section, you'll learn how to read an analog input with the ESP32. This is useful to read values from variable resistors like potentiometers or analog sensors.



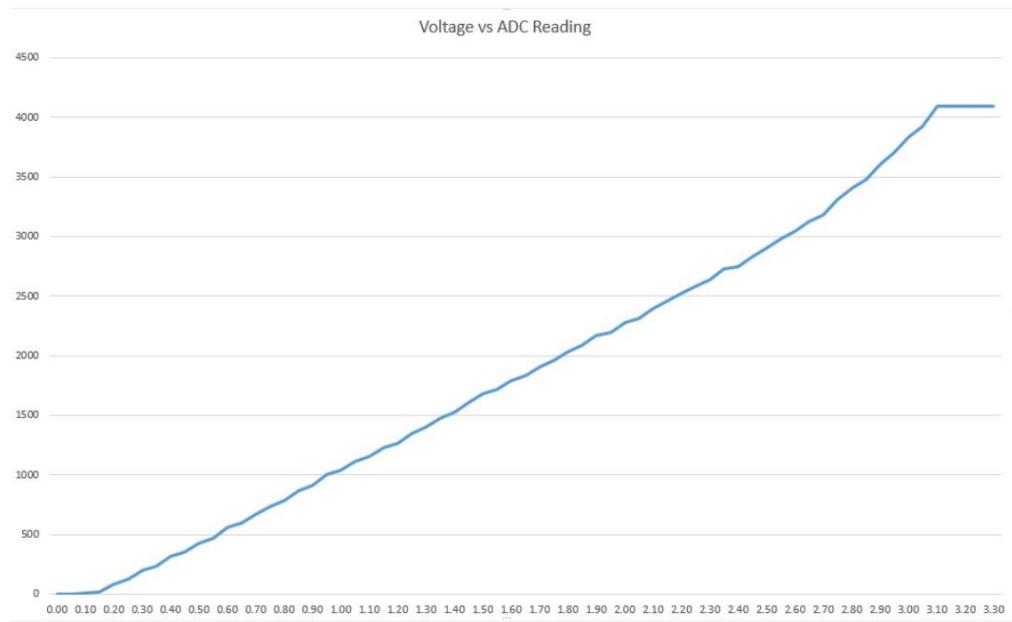
Analog Inputs

Reading an analog value with the ESP32 means you can measure varying voltage levels between 0V and 3.3V. The voltage measured is then assigned to a value between 0 and 4095, in which 0V corresponds to 0, and 3.3V corresponds to 4095. Any voltage between 0V and 3.3V will be given the corresponding value in between.



ESP32 ADC is Non-linear

Ideally, you would expect a linear behavior when using the ESP32 ADC pins. 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 3.3V from 3.2V. You'll get the same value for both voltages: 4095. The same happens for very small voltages: for 0V and 0.1V 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, which accepts as an argument the GPIO you want to read, as follows:

```
analogRead(GPIO);
```

The ESP-WROOM-32 supports measurements in 18 different channels. But not all are available in most ESP32 board models. Grab your ESP32 board pinout and locate the ADC pins.

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.

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

Project Example

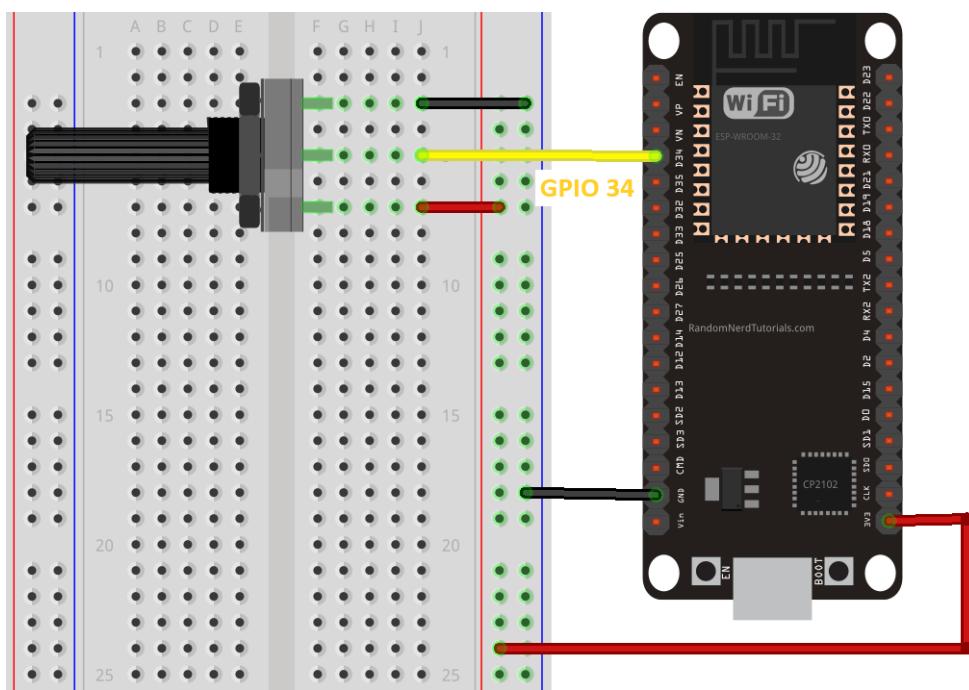
Let's make a simple example to read an analog value from a potentiometer to see how this works.

Wiring the Circuit

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

- ESP32 DOIT DEVKIT V1 Board
 - Potentiometer
 - Breadboard
 - Jumper wires

Wire a potentiometer to your ESP32 using the following schematic diagram as a reference, with the potentiometer middle pin connected to GPIO 34.



Code

Copy the following code to your Arduino IDE.

- [Click here to download the 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);
}
```

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

How Does the Code Work?

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

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

In the `setup()`, we initialize serial communication at a baud rate of 115200.

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

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

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

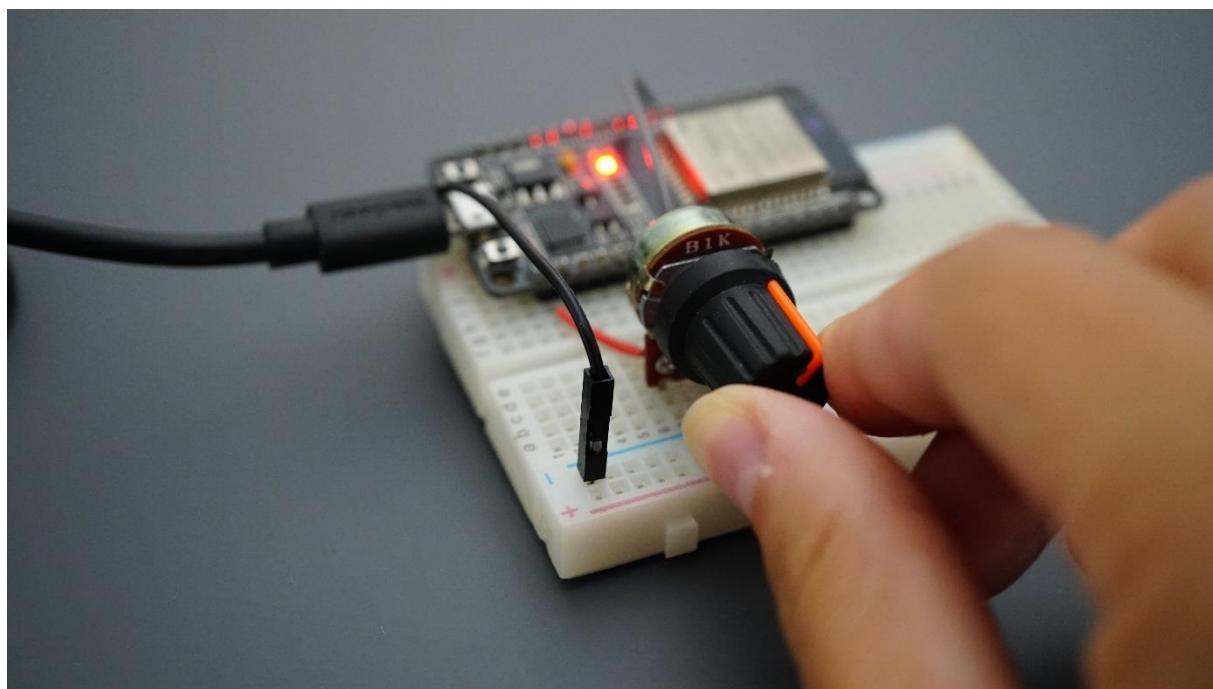
Finally, you print the values read from the potentiometer in the serial monitor.

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

Testing the Example

Upload the code provided to your ESP32.

Open the Serial Monitor at a baud rate of 115200. Rotate the potentiometer and see the values changing.



The maximum value you can get is 4095, and the minimum value is 0.

```
4095
4095
3494
3467
3487
3497
3524
3526
2405
1734
1269
646
128
0
0
0
427
1235
1232
```

Ln 17, Col 2 DOIT ESP32 DEVKIT V1 on COM3

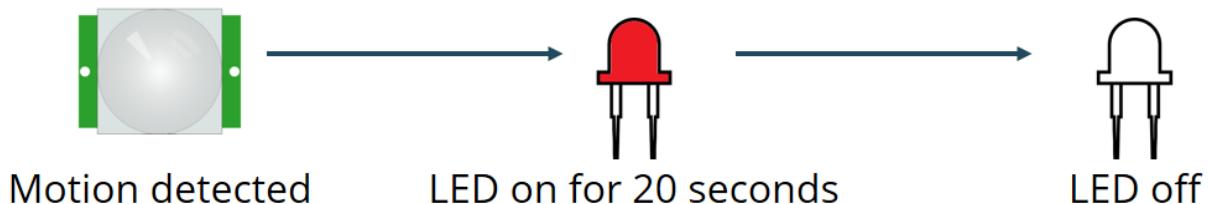
Wrapping Up

In summary:

- The ADC 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 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.

2.7 - ESP32 with PIR Motion Sensor: Interrupts and Timers

In this unit, you're going to learn how to detect motion using a PIR motion sensor. In our example, when motion is detected, the ESP32 starts a timer and turns an LED on for a predefined number of seconds. After that time, the LED will turn off.



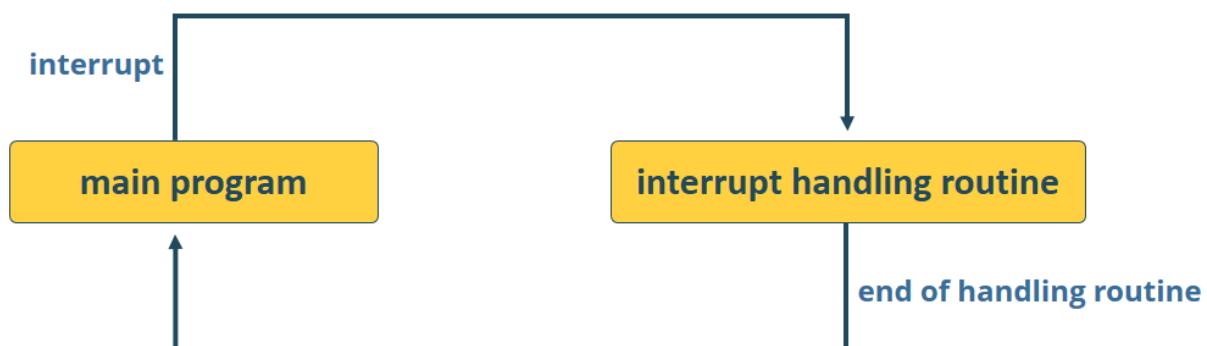
With this example, we'll introduce two new concepts: **interrupts** and **timers**.

Introducing Interrupts

Interrupts are useful for making things happen automatically in microcontroller programs and can help solve timing problems. Interrupts and event handling provide mechanisms to respond to external events, enabling the ESP32 to react quickly to changes without continuously polling (continuously checking the current value of a pin or variable).

What are Interrupts?

Interrupts are signals that pause the normal execution flow of a program to handle a specific event. When an interrupt happens, the processor stops the execution of the main program to execute a task and then gets back to the main program. That task is also referred to as an *interrupt handling/service routine*.



Using interrupts is especially useful to trigger an action whenever motion is detected or whenever a pushbutton is pressed without the need for constantly checking its state.

Setting Up an Interrupt

To set an interrupt in the Arduino IDE, you use the `attachInterrupt()` function, which accepts as arguments the GPIO pin, the name of the function to be executed, and mode (we'll see the different modes next).

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

GPIO Interrupt Pin

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

```
digitalPinToInterruption(27)
```

With an ESP32 board, all pins that can be used as normal Inputs or Outputs can be set as interrupts. In this project, we'll use GPIO 27 as an interrupt connected to the PIR Motion sensor.

Function to be triggered - ISR

The second argument of the `attachInterrupt()` function is the name of the function that will be called every time the interrupt is triggered – the interrupt service routine (**ISR**). The ISR function should be as simple as possible, so the processor gets back to the execution of the main program quickly.

The best approach is to signal the main code that the interrupt has happened by using a global variable and within the `loop()` check and clear that flag, and execute code.

ISRs need to have the `IRAM_ATTR` prefix before the function definition to run the interrupt code in RAM.

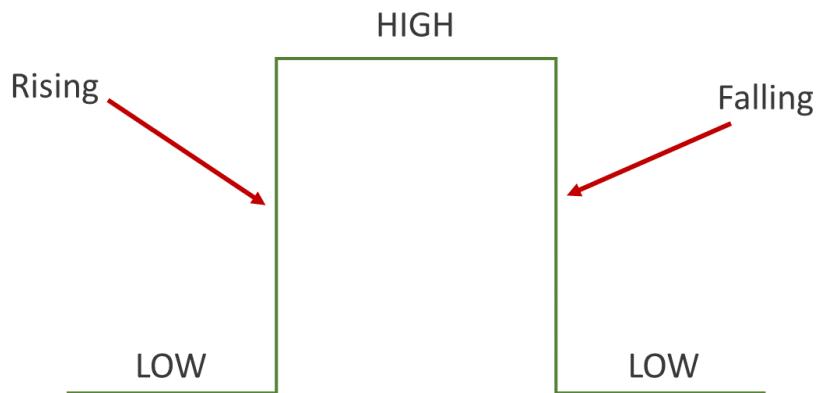
Mode

The third argument is the mode. There are five 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**;

When the PIR motion sensor detects motion, it sends a **HIGH** signal—the GPIO it is connected to goes from **LOW** to **HIGH**. So, we should use **RISING** mode.

The following picture will help you better understand the different trigger modes.



Introducing Timers



In this unit, 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 call `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 simultaneously, 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 check how much time has passed without blocking your code.

Blinking an LED with millis()

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

- [Click here to download the code](#)

```
// 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 hold time  
// The value will quickly become too large for an int to store  
unsigned long previousMillis = 0; // will store last time LED was updated
```

```

// constants won't change :
const long interval = 1000;    // interval at which to blink (milliseconds)

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 the time.
  // check to see if it's time to blink the LED; that is, if the
  // difference between the current time and last time you blinked
  // the LED is bigger than the interval at which you want to blink the LED.
  unsigned long currentMillis = millis();

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

    // if the LED is off turn it on and vice-versa:
    if (ledState == LOW) {
      ledState = HIGH;
    } else {
      ledState = LOW;
    }

    // set the LED with the ledState of the variable:
    digitalWrite(ledPin, ledState);
  }
}

```

How Does the Code Work?

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

This code subtracts the previously recorded time (`previousMillis`) from the current time (`currentMillis`).

```
if (currentMillis - previousMillis >= interval) {
```

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.

```
previousMillis = currentMillis;

// if the LED is off turn it on and vice-versa:
if (ledState == LOW) {
  ledState = HIGH;
} else {
  ledState = LOW;
}
```

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

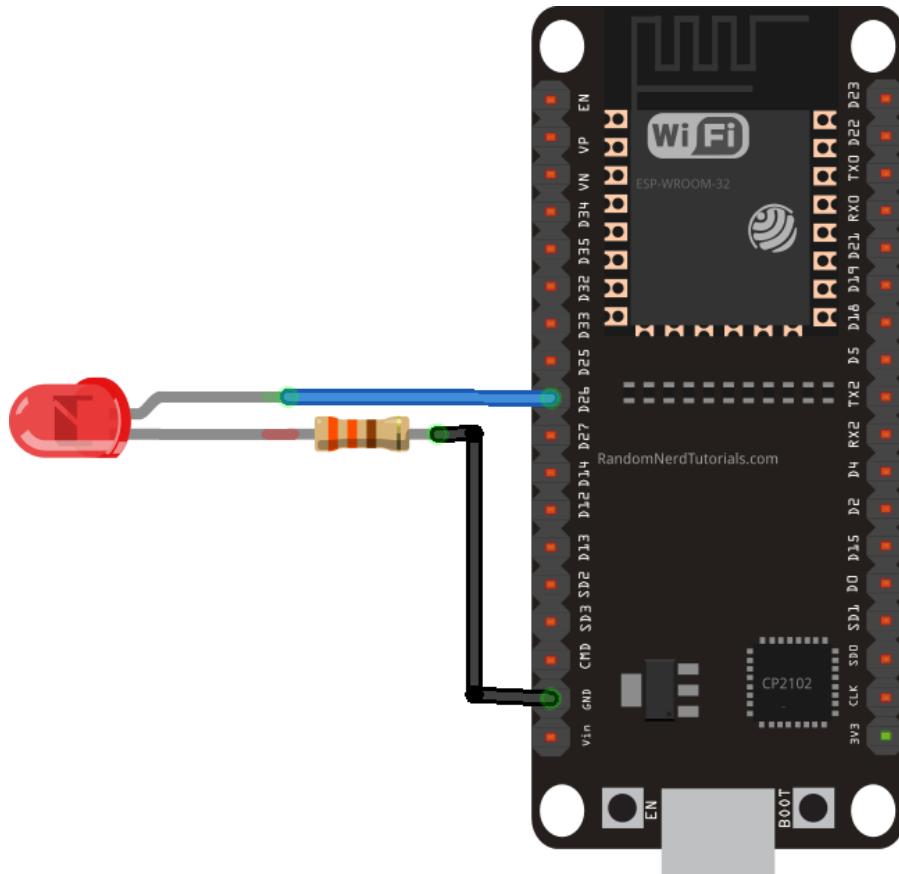
You should now be able to understand that you can add other tasks to your `loop()` function, and your code would still be blinking the LED every second.

Testing the Code

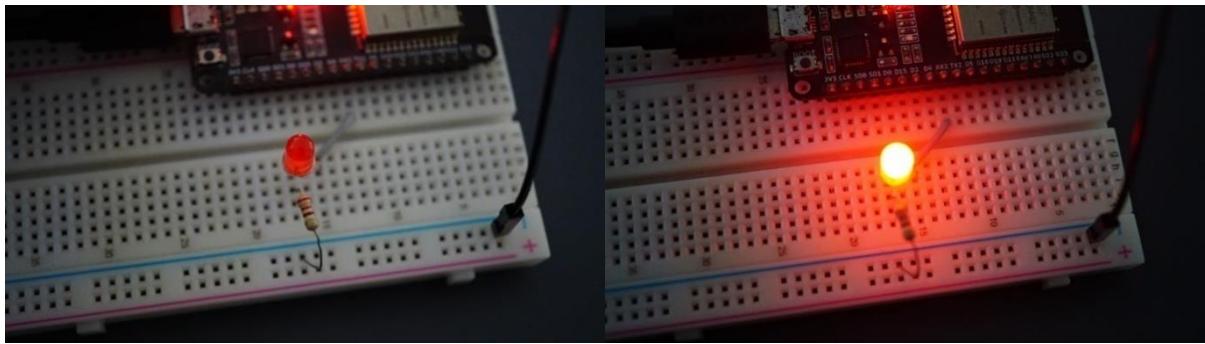
You can upload the code provided to your ESP32. To test this project, connect an LED to GPIO 26 (you can use the schematic diagram below as a reference). You can modify the number of milliseconds to see how it works.

Here's a list of parts you need to assemble this circuit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [220 Ohm resistor \(or similar value\)](#)
- [Jumper wires](#)
- [Breadboard](#)



The LED connected to the ESP32 should be blinking every second.

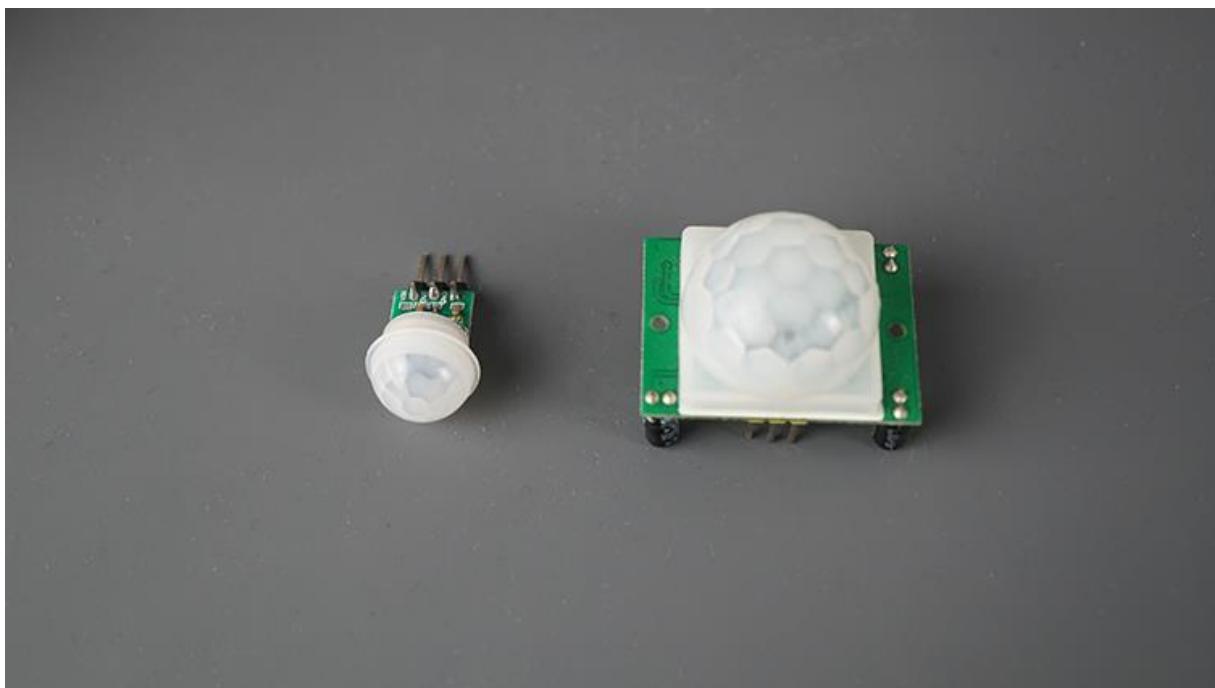


ESP32 with PIR Motion Sensor

After understanding these two new concepts: **interrupts** and **timers**, let's continue with the project to detect motion with a PIR sensor.

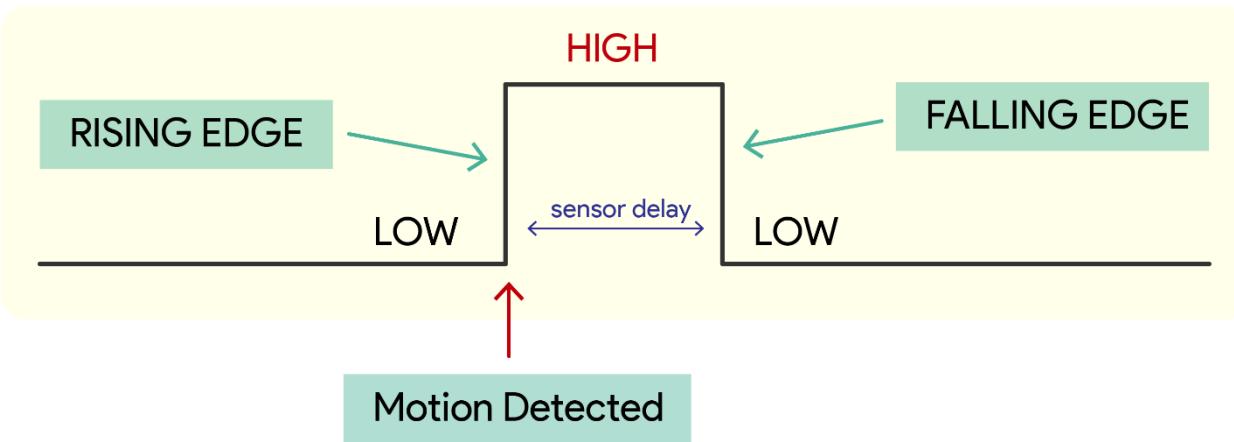
The PIR Motion Sensor

A Passive Infrared (PIR) motion sensor is designed to detect infrared radiation emitted by objects in its field of view. PIR sensors detect changes in infrared radiation, primarily emitted by living things. When there is motion in the sensor's range, there are fluctuations in the detected infrared radiation. The following picture shows two of the most popular motion sensors used in hobbyist electronics projects: the: mini PIR motion sensor (AM312) and PIR motion sensor (HC-SR501).



These sensors output a **HIGH** signal when movement is detected, or **LOW** if it doesn't detect any movement. The digital output from the PIR sensor can be read by an ESP32 GPIO pin, allowing you to program specific actions based on the detected motion status. We'll create a simple example that will light up an LED when motion is detected. But, after learning how it works, the same way of thinking can be used for useful applications like sending an email, triggering an alarm, etc.

An important concept about PIR motion sensors is the *dwell time* (reset time or sensor delay). The dwell time is the duration during which a PIR motion sensor's output remains HIGH after detecting motion before returning to a LOW state. Only after the dwell time, the sensor is able to detect motion again.



Note: the AM312 PIR motion sensor that we'll use in this tutorial has a default delay time of 8 seconds. This means that it won't be triggered before 8 seconds have passed since the last trigger.

Wiring the Circuit

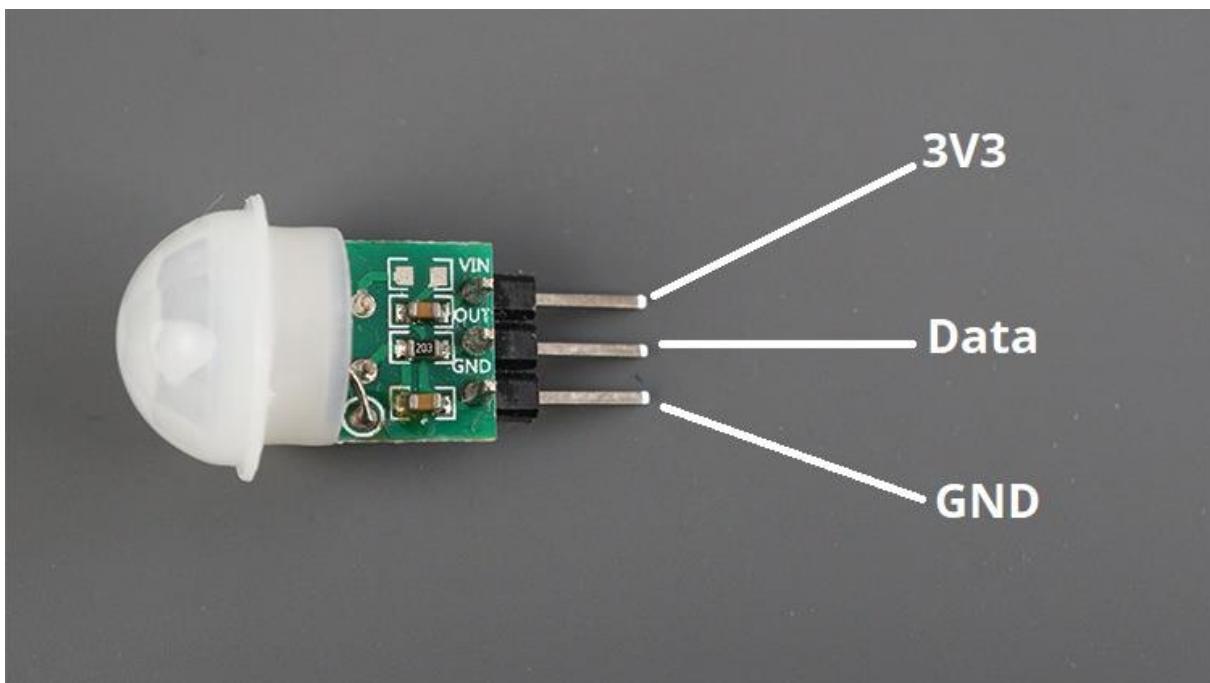
Here's a list of parts you need to assemble the circuit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [Mini PIR motion sensor \(AM312\)](#) or [PIR motion sensor \(HC-SR501\)](#)
- [5mm LED](#)
- [220 Ohm resistor \(or similar value\)](#)
- [Jumper wires](#)
- [Breadboard](#)

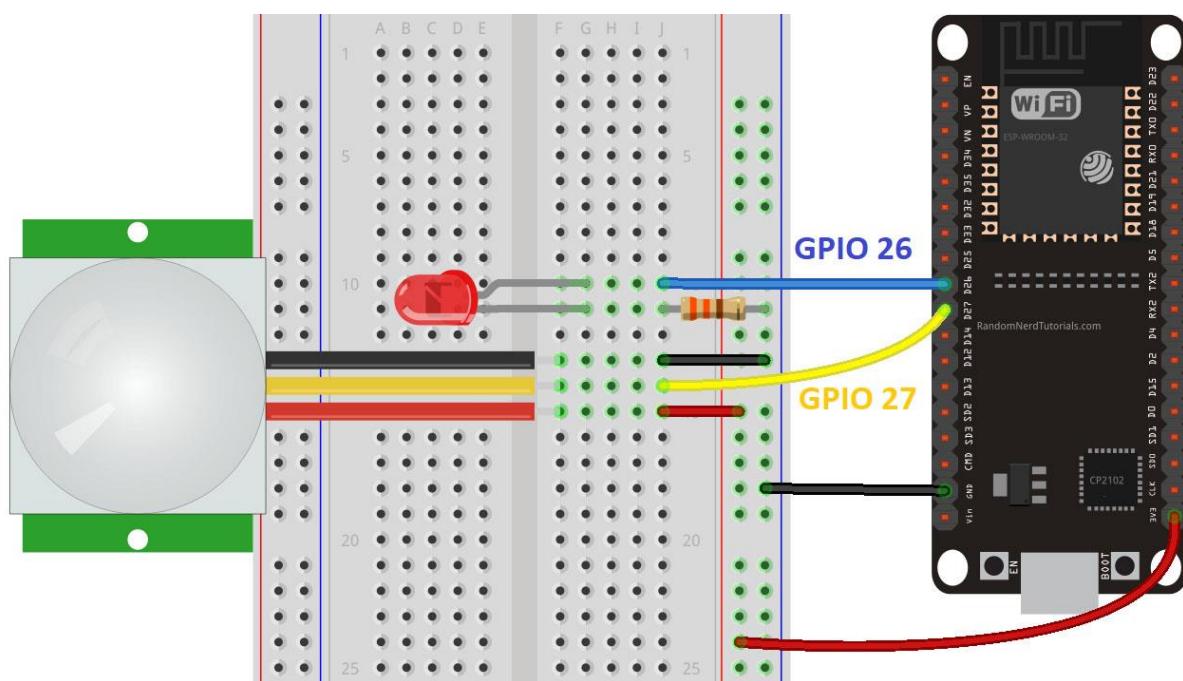
PIR sensors have a GND, VCC, and a data line. Connect the GND to the ESP32 GND, VCC to 3.3V, and the data line to an available GPIO. We'll use the following pins:

PIR Sensor	GND	VCC	Data
ESP32	GND	3.3V	GPIO27

For this circuit, the LED is connected to GPIO 26 and the PIR motion sensor data pin is connected to GPIO 27. We'll use the AM312 PIR motion sensor that works with 3.3V. The following figure shows the AM312 PIR motion sensor pinout.



You can use the following diagram as a reference to wire your circuit.



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-SR01](#), it usually operates at 5V. You can either [modify it to operate at 3.3V](#) or simply power it using the ESP32 Vin pin.

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 desired number of seconds.

- [Click here to download the code](#)

```
#define timeSeconds 20

// 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 timer
void IRAM_ATTR detectsMovement() {
    digitalWrite(led, HIGH);
    startTimer = true;
    lastTrigger = millis();
}

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

    // PIR Motion Sensor mode INPUT_PULLUP
    pinMode(motionSensor, INPUT_PULLUP);
    // Set motionSensor pin as interrupt, assign interrupt function and set RISING mode
    attachInterrupt(digitalPinToInterrupt(motionSensor), detectsMovement, RISING);

    // Set LED to LOW
    pinMode(led, OUTPUT);
    digitalWrite(led, LOW);
}

void loop() {
    // Current time
    now = millis();
    if((digitalRead(led) == HIGH) && (motion == false)) {
```

```
    Serial.println("MOTION DETECTED!!!");
    motion = true;
}
// Turn off the LED after the number of seconds defined in the timeSeconds variable
if(startTimer && (now - lastTrigger > (timeSeconds*1000))) {
    Serial.println("Motion stopped...");
    digitalWrite(led, LOW);
    startTimer = false;
    motion = false;
}
}
```

How Does the Code Work?

Let's take a look at the code.

Start by assigning two GPIO pins to the `led` and `motionSensor` variables.

```
const int led = 26;
const int motionSensor = 27;
```

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

```
unsigned long now = millis();
unsigned 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.

Finally, the `motion` variable saves whether motion is being detected or not.

```
boolean motion = false;
```

setup()

In the `setup()`, start by initializing the serial monitor at a 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 described earlier.

```
attachInterrupt(digitalPinToInterruption(motionSensor), detectsMovement, RISING);
```

The pin that will detect motion is GPIO 27, and it will call the `detectsMovement()` function 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();
```

The following condition checks if the LED is currently turned on (`HIGH`) and if the `motion` flag is `false`. The `motion` flag indicates whether motion has been detected since the LED was turned on.

```
if((digitalRead(led) == HIGH) && (motion == false)) {
    Serial.println("MOTION DETECTED!!!");
    motion = true;
}
```

If both conditions are true, it means motion was detected while the LED was on. It prints "MOTION DETECTED!!!" in the Serial Monitor and sets the `motion` flag to `true`.

Why do we need to check the state of the led and motion simultaneously?

- **Avoiding repeated detection:** without checking `motion == false`, every loop iteration where the sensor detects motion would repeatedly print "MOTION DETECTED!!!" and set `motion` to `true` again. This would lead to redundant processing and potentially overwhelm the serial monitor with repeated messages.
- **Ensuring state transition:** by combining these conditions, the system ensures that the transition from "no motion" to "motion detected" happens only once for each new motion event. This change of state

triggers the appropriate actions (like printing the message and setting `motion` to `true`) only when a new motion is detected

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 runs when motion is triggered. It turns the LED on, sets the `startTimer` boolean variable to `True`, and updates the `lastTrigger` variable with the current time.

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

Note: `IRAM_ATTR` is used to run the interrupt code in RAM. Otherwise, the 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;
    motion = false;
}
```

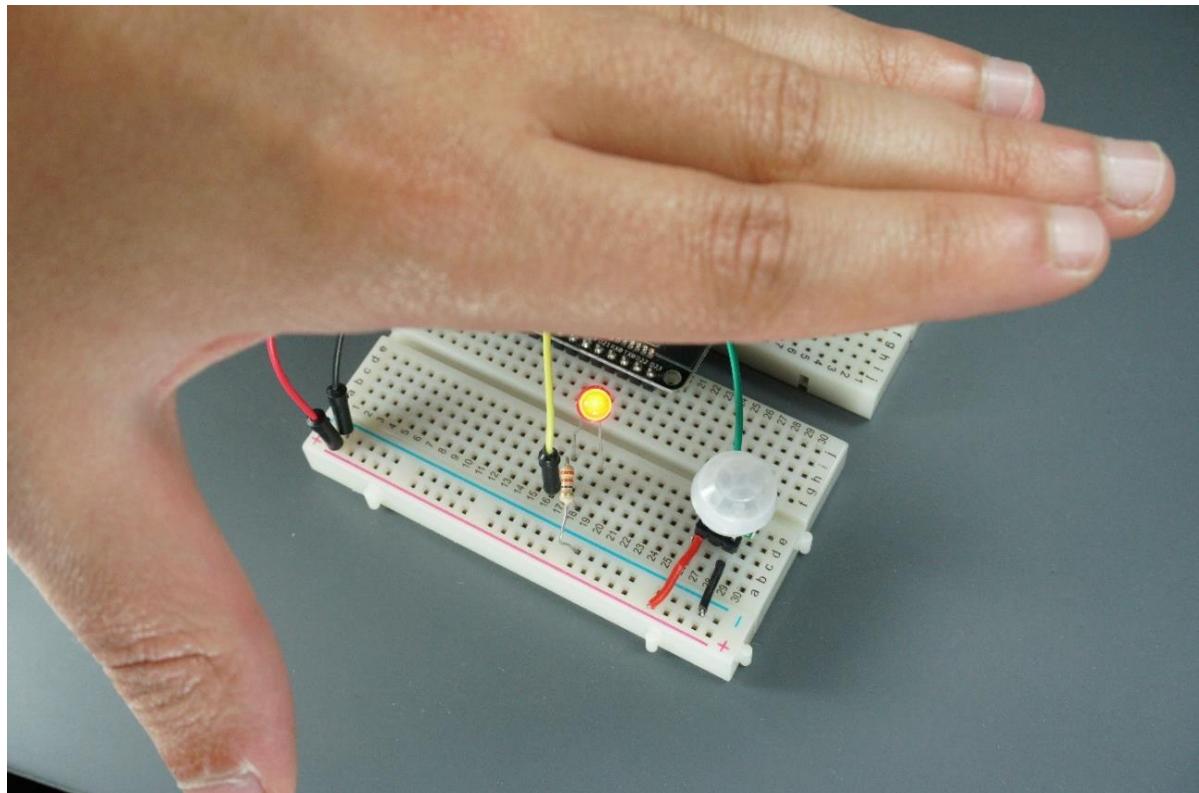
The "Motion stopped..." message will be printed in the serial monitor, the LED is turned off, and the `startTimer` variable is set to `false`.

Demonstration

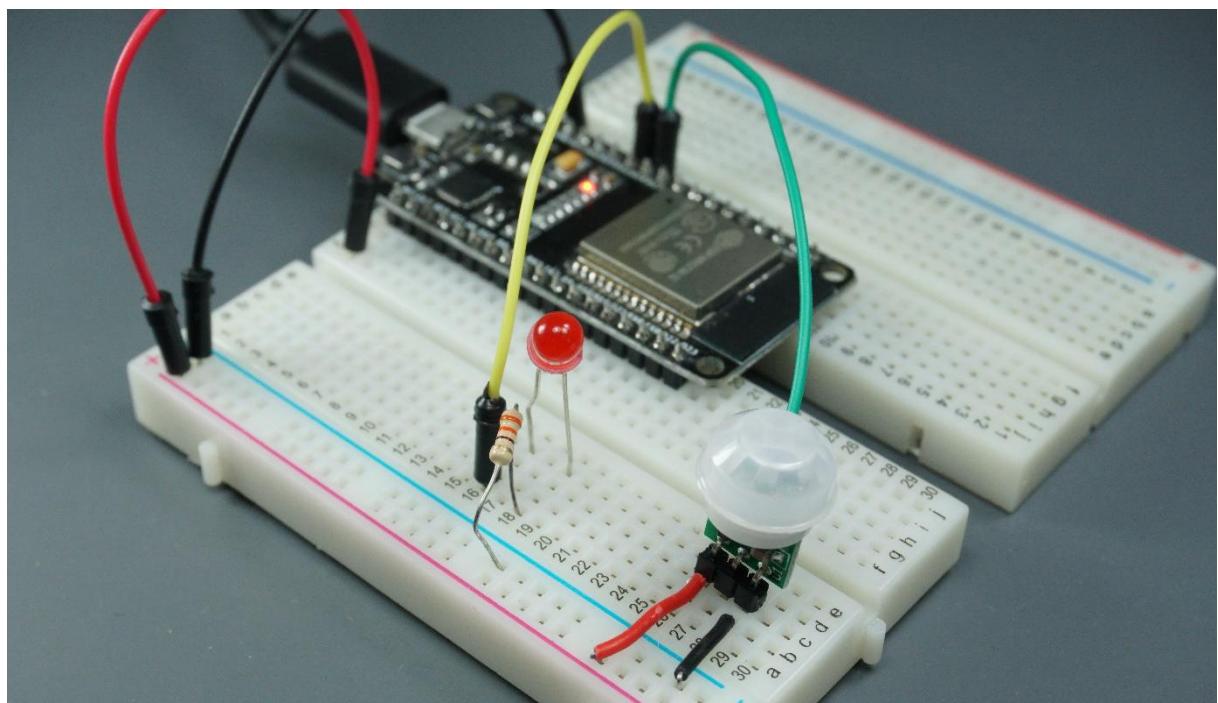
Upload the code to your ESP32 board. Make sure you have the right board and COM port selected. Open the Serial Monitor at a baud rate of 115200.



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

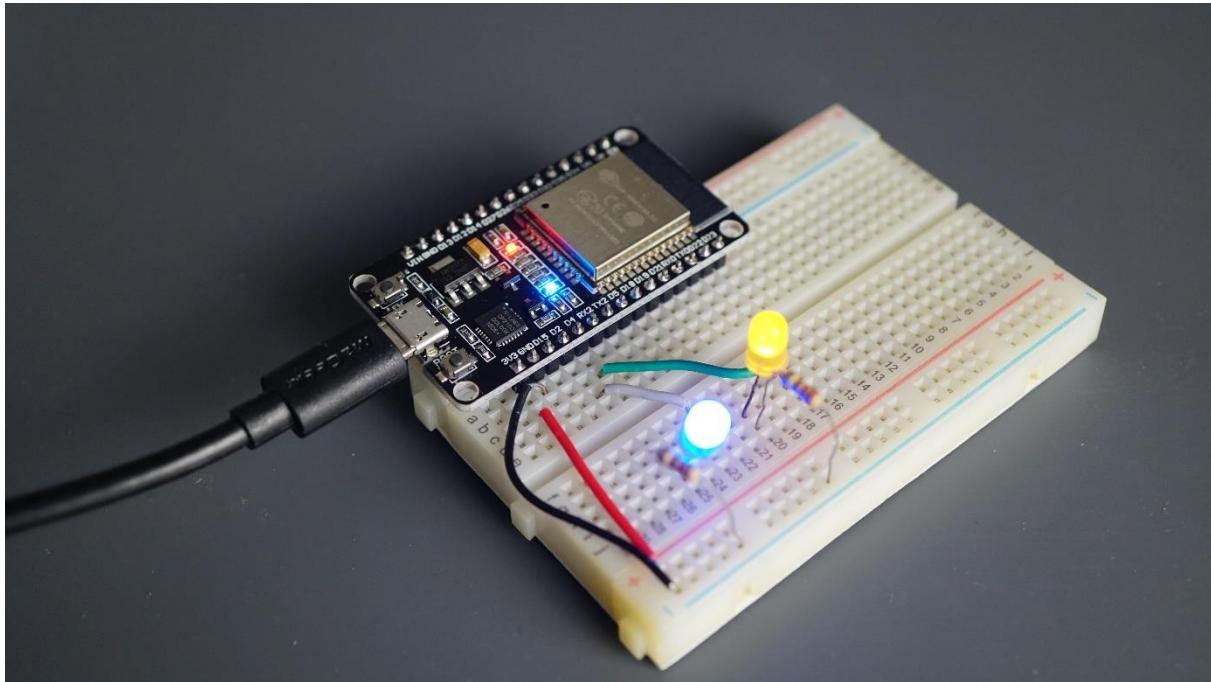
To wrap up, interrupts are used to detect a change in the GPIO state without the need to constantly read the current GPIO value. With interrupts, when a change is detected, a function is triggered.

You've also learned how to set a simple timer that allows you to check if a predefined number of seconds have passed without blocking your code.

Interrupts and timers are two essential concepts that will be very useful in your projects later on.

2.8 - ESP32 Dual Core: Create Tasks

The ESP32 comes with 2 Xtensa 32-bit LX6 microprocessors: core 0 and core 1. In this Unit, we'll show you how to run code on the ESP32 second core by creating tasks. You can run pieces of code simultaneously on both cores to have multitasking.



Note: you don't necessarily need to run dual core to achieve multitasking.

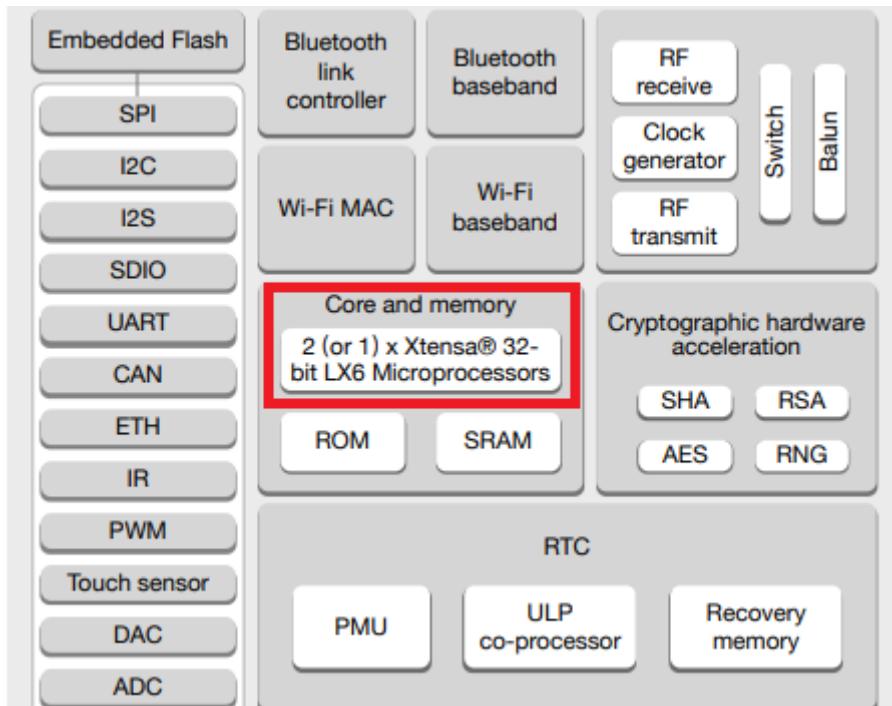
Parts required:

Here's a list of the parts required for this Unit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- 2x [5mm LED](#)
- 2x [220 Ohm resistor \(or similar value\)](#)
- [Breadboard](#)
- [Jumper wires](#)

Introduction

The ESP32 comes with 2 Xtensa 32-bit LX6 microprocessors: core 0 and core 1.



By default, when we run code on the ESP32, it runs on core 1. When we upload code to the ESP32 using the Arduino IDE, it just runs – we don't have to worry about which core executes the code.

There's a function that returns in which core the code is running:

```
xPortGetCoreID();
```

If you use that function in an Arduino sketch, you'll see that both the `setup()` and `loop()` are running on core 1. Test it yourself by uploading the following sketch to your ESP32.

- [Click here to download the code.](#)

```
void setup() {
    Serial.begin(115200);
    Serial.print("setup() running on core ");
    Serial.println(xPortGetCoreID());
}

void loop() {
    Serial.print("loop() running on core ");
    Serial.println(xPortGetCoreID());
```

The screenshot shows the Arduino Serial Monitor window. The title bar says "Output" and "Serial Monitor". The message area displays FreeRTOS task output. A red box highlights the following lines:

```
configSip: 0, SPIWR: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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
setup() running on core 1
loop() running on core 1
loop() running on core 1
```

Below the message area, the status bar shows "Ln 15, Col 15 DOIT ESP32 DEVKIT V1 on COM3" and icons for a serial port and a file.

Create Tasks

The Arduino IDE supports FreeRTOS for the ESP32, which is a Real Time Operating system. This allows us to handle several tasks in parallel that run independently. Tasks are pieces of code that execute something. For example, it can be blinking an LED, making a network request, getting sensor readings, publishing sensor readings, etc...

To assign specific parts of code to a specific core, you need to create tasks. When creating a task you can choose in which core it will run as well as its priority. Priority values start at 0, which 0 is the lowest priority. The processor will run the tasks with higher priority first. To create tasks, you need to follow the next steps:

- 1) Create a task handle. An example for Task1:

```
TaskHandle_t Task1;
```

- 2) In the `setup()` create a task assigned to a specific core using the `xTaskCreatePinnedToCore()` function. This function takes several arguments, including the priority and the core where the task should run (the last parameter). See how to use that function below:

```
xTaskCreatePinnedToCore(
    Task1code, /* Function to implement the task */
    "Task1",   /* Name of the task */
    10000,     /* Stack size in words */
    NULL,      /* Task input parameter */
```

```
0,          /* Priority of the task */
&Task1,    /* Task handle. */
0);        /* Core where the task should run */
```

- 3) After creating the task, you should create a function that contains the code for the created task. In this example, you need to create the `Task1code()` function. Here's how the task function looks like:

```
void Task1code( void * parameter) {
    for(;;){
        /*Code for task 1 - infinite loop
        (...)*/
    }
}
```

The `for(;;)` expression creates an infinite loop. So, this function runs similarly to the `loop()` function. You can use it as a second loop in your code, for example.

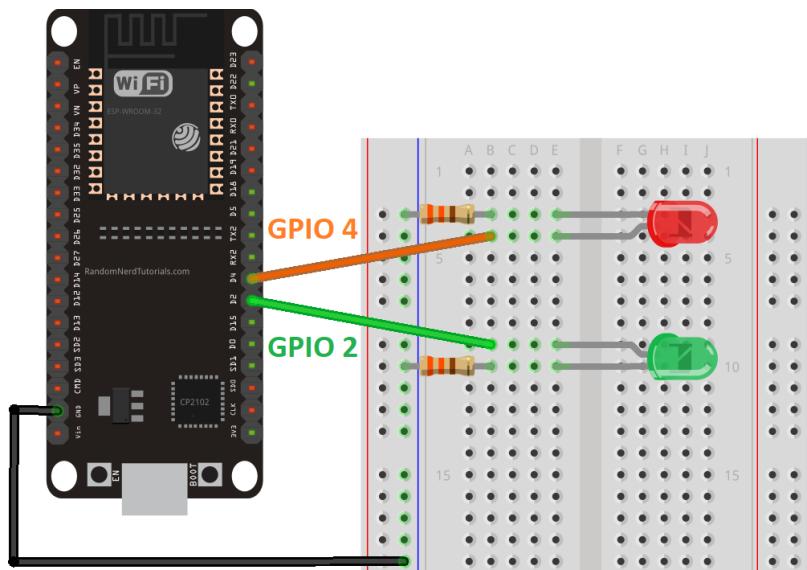
During the code execution, if you want to delete the created task, you can use the `vTaskDelete()` function, that accepts the task handle (`Task1`) as an argument:

```
vTaskDelete(Task1);
```

Let's see how these concepts work with a simple example.

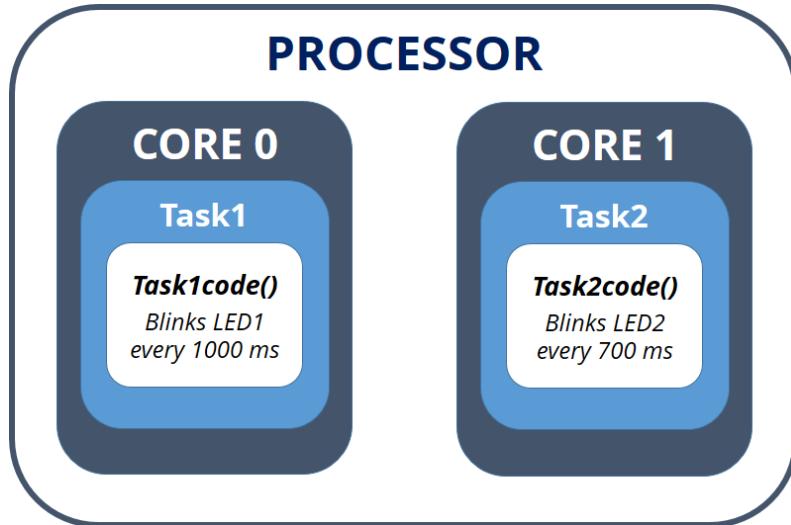
Create Tasks in Different Cores - Example

To create different tasks running on different cores, we'll create two tasks that blink LEDs with different delay times. Wire two LEDs to the ESP32 as shown in the following diagram:



We'll create two tasks running on different cores:

- Task1 runs on core 0;
- Task2 runs on core 1;



Upload the next sketch to your ESP32 to blink each LED in a different core:

- [Click here to download the code.](#)

```
TaskHandle_t Task1;
TaskHandle_t Task2;

// LED pins
const int led1 = 2;
const int led2 = 4;

void setup() {
    Serial.begin(115200);
    pinMode(led1, OUTPUT);
    pinMode(led2, OUTPUT);

    // create a task that will be executed in the Task1code() function
    // with priority 1 and executed on core 0
    xTaskCreatePinnedToCore(
        Task1code, /* Task function. */
        "Task1", /* name of task. */
        10000, /* Stack size of task */
        NULL, /* parameter of the task */
        1, /* priority of the task */
        &Task1, /* Task handle to keep track of created task */
        0); /* pin task to core 0 */

    delay(500);

    // create a task that will be executed in the Task2code() function
    // with priority 1 and executed on core 1
    xTaskCreatePinnedToCore(
        Task2code, /* Task function. */
        "Task2", /* name of task. */
        10000, /* Stack size of task */
        NULL, /* parameter of the task */
```

```

        1,          /* priority of the task */
&Task2,      /* Task handle to keep track of created task */
        1);        /* pin task to core 1 */

    delay(500);
}

// Task1code: blinks an LED every 1000 ms
void Task1code( void * pvParameters ){
    Serial.print("Task1 running on core ");
    Serial.println(xPortGetCoreID());

    for(;;){
        digitalWrite(led1, HIGH);
        delay(1000);
        digitalWrite(led1, LOW);
        delay(1000);
    }
}

// Task2code: blinks an LED every 700 ms
void Task2code( void * pvParameters ){
    Serial.print("Task2 running on core ");
    Serial.println(xPortGetCoreID());

    for(;;){
        digitalWrite(led2, HIGH);
        delay(700);
        digitalWrite(led2, LOW);
        delay(700);
    }
}

void loop() {
}

```

How Does the Code Work?

Note: in the code, we create two tasks and assign one task to core 0 and another to core 1. Arduino sketches run on core 1 by default. Alternatively, you can write the code for `Task2` in the `loop()` (there was no need to create another task). In this case, we create two different tasks for learning purposes. But, depending on your project requirements, it may be more practical to organize your code in tasks as demonstrated in this example.

The code starts by creating a task handle for Task1 and Task2 called `Task1` and `Task2`.

```

TaskHandle_t Task1;
TaskHandle_t Task2;

```

Assign GPIO 2 and GPIO 4 to the LEDs:

```
const int led1 = 2;
const int led2 = 4;
```

In the `setup()`, initialize the Serial Monitor at a baud rate of 115200:

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

Declare the LEDs as outputs:

```
pinMode(led1, OUTPUT);
pinMode(led2, OUTPUT);
```

Then, create `Task1` using the `xTaskCreatePinnedToCore()` function:

```
xTaskCreatePinnedToCore(
    Task1code,      /* Task function. */
    "Task1",        /* name of task. */
    10000,          /* Stack size of task */
    NULL,           /* parameter of the task */
    1,              /* priority of the task */
    &Task1,         /* Task handle to keep track of created task */
    0);             /* pin task to core 0 */
```

`Task1` will be implemented with the `Task1code()` function. So, we need to create that function later in the code. We give the task priority 1, and pinned it to core 0.

We create `Task2` using the same method:

```
xTaskCreatePinnedToCore(
    Task2code,      /* Task function. */
    "Task2",        /* name of task. */
    10000,          /* Stack size of task */
    NULL,           /* parameter of the task */
    1,              /* priority of the task */
    &Task2,         /* Task handle to keep track of created task */
    1);             /* pin task to core 1 */
```

After creating the tasks, we need to create the functions that will execute those tasks.

```
void Task1code( void * pvParameters ){
    Serial.print("Task1 running on core ");
    Serial.println(xPortGetCoreID());

    for(;){
        digitalWrite(led1, HIGH);
        delay(1000);
        digitalWrite(led1, LOW);
        delay(1000);
    }
}
```

The function for Task1 is called `Task1code()` (you can call it whatever you want).

For debugging purposes, we first print the core in which the task is running:

```
Serial.print("Task1 running on core ");
Serial.println(xPortGetCoreID());
```

Then, we have an infinite loop similar to the `loop()` on the Arduino sketch. In that loop, we blink LED1 every one second.

The same thing happens for `Task2`, but we blink the LED with a different delay time.

```
void Task2code( void * pvParameters ){
    Serial.print("Task2 running on core ");
    Serial.println(xPortGetCoreID());

    for(;;){
        digitalWrite(led2, HIGH);
        delay(700);
        digitalWrite(led2, LOW);
        delay(700);
    }
}
```

Finally, the `loop()` function is empty:

```
void loop() {  
}
```

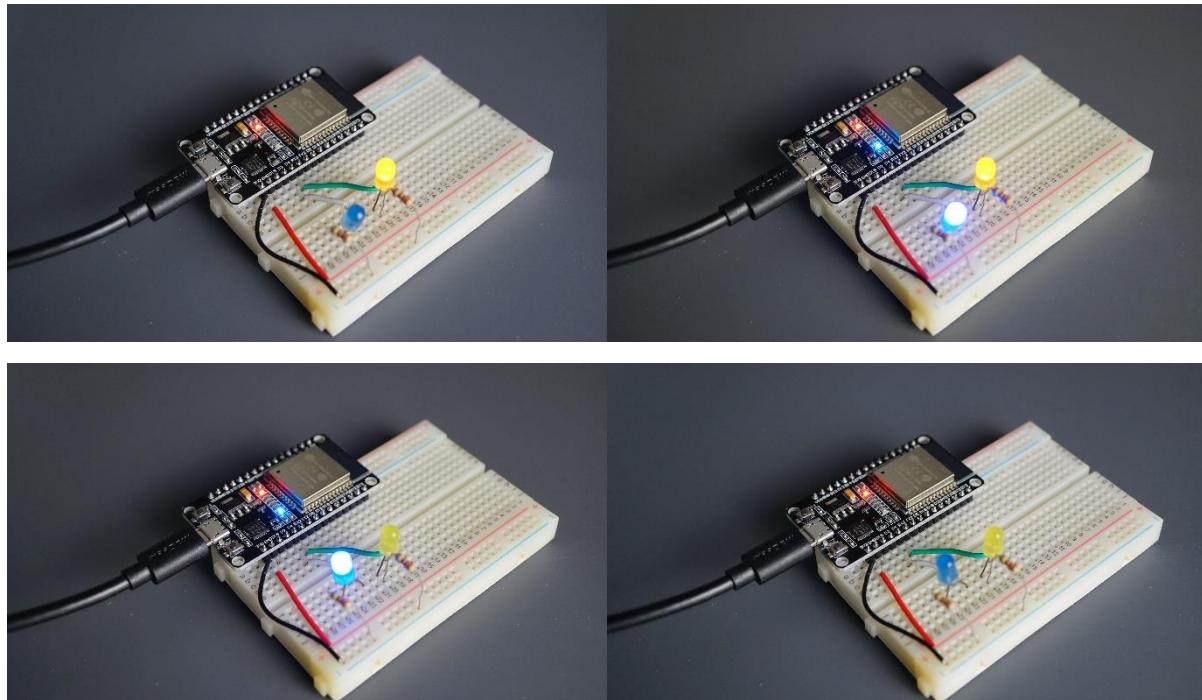
Demonstration

Upload the 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. You should get the following indicating where each task is running.



As expected Task1 is running on core 0, while Task2 is running on core 1.

In your circuit, one LED should be blinking every 1 second, and the other should be blinking every 700 milliseconds



Wrapping Up

In summary:

- The ESP32 is dual-core;
- Arduino sketches run on core 1 by default;
- To use core 0, you need to create tasks;
- You can use the `xTaskCreatePinnedToCore()` function to pin a specific task to a specific core;
- Using this method, you can run two different tasks independently and simultaneously using the two cores.

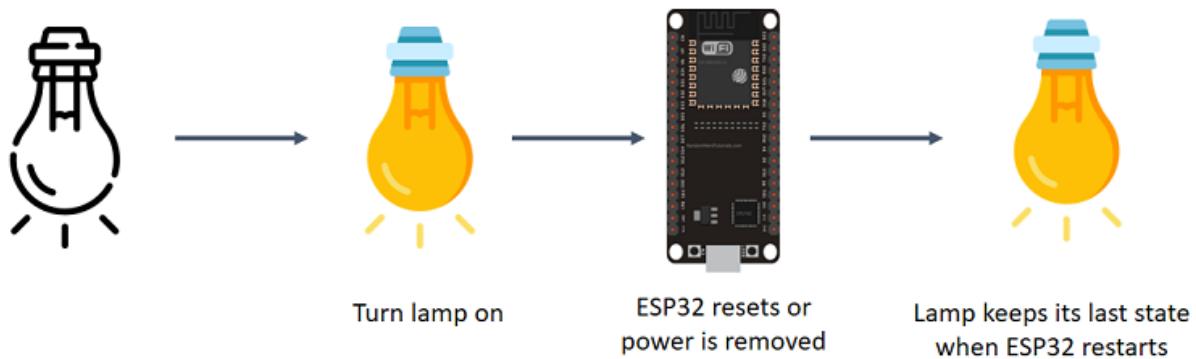
In this Unit, we've provided a simple example with LEDs. The idea is to use this method with more advanced projects with real-world applications. For example, it may be useful to use one core to take sensor readings and the other to publish those readings to a home automation system.

MODULE 3

Saving Data and Handling Files

3.1 - Save Data Permanently using the Preferences Library

In this Unit, you'll learn how to save data permanently on the ESP32 non-volatile memory using the `Preferences.h` library. The data held in this memory persists across resets or power failures. Using the `Preferences.h` library is useful to save data like network credentials, API keys, threshold values, or even the last state of a GPIO.



You'll learn how to save and read data from `Preferences`, and we'll build a basic example in which the ESP32 "remembers" the last state of a GPIO after a reset.

The `Preferences.h` Library

The `Preferences.h` library is "installed" automatically when you install the ESP32 boards in your Arduino IDE.

It stores variable values through `key:value` pairs. Saving data permanently can be important to:

- remember the last state of a variable;
- save settings;
- save how many times an appliance was activated;
- or any other data type you need to save permanently.

If, instead of variables, you need to save files on the ESP32, we recommend using the filesystem (SPIFFS or LittleFS) instead. We cover this in [Unit 3.2 and 3.3](#).

Save Data Using Preferences.h Library

The data saved using preferences is structured like this:

```
namespace {  
    key: value  
}
```

You can save different keys on the same namespace, for example:

```
namespace {  
    key1: value1,  
    key2: value2  
}
```

In a practical example, this configuration could be used to save your network credentials. For example:

```
credentials {  
    ssid: "your_ssid",  
    pass: "your_pass"  
}
```

In the preceding example, `credentials` is the namespace, and `ssid` and `pass` are the keys.

You can also have multiple namespaces with the same key (but each key with its value):

```
namespace1 {  
    key: value1  
}  
namespace2 {  
    key: value2  
}
```

When using the `Preferences.h` library, you should define the data type you want to save. Later, if you want to read that data, you must know the saved data type. In other words, the data type of writing and reading should be the same.

You can save the following data types using `Preferences.h`: `char`, `Uchar`, `short`, `Ushort`, `int`, `Uint`, `long`, `Ulong`, `long64`, `Ulong64`, `float`, `double`, `bool`, `string`, and `bytes`.

For more information, you can check the [Preferences.cpp file here](#).

Preferences.h Library Useful Functions

To use the `Preferences.h` library to store data, first, you need to include it in your sketch:

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

Then, you must initiate an instance of the `Preferences` library. You can call it `preferences`, for example:

```
Preferences preferences;
```

After this, you can use the following methods to handle data using the `Preferences.h` library.

Start Preferences

The `begin()` method opens a “storage space” with a defined namespace. The `false` argument means that we’ll use it in read/write mode. Use `true` to open or create the namespace in read-only mode.

```
preferences.begin("my-app", false);
```

In this case, the namespace name is `my-app`. The namespace name is limited to 15 characters.

Clear Preferences

Use `clear()` to clear all preferences under the opened namespace (it doesn’t delete the namespace):

```
preferences.clear();
```

Remove Key

Remove a key from the opened namespace:

```
preferences.remove(key);
```

Close Preferences

Use the `end()` method to close the preferences under the opened namespace:

```
preferences.end();
```

Put a Key Value (Save a value)

You should use different methods depending on the variable type you want to save.

Char	<code>putChar(const char* key, int8_t value)</code>
Unsigned Char	<code>putUChar(const char* key, int8_t value)</code>
Short	<code>putShort(const char* key, int16_t value)</code>
Unsigned Short	<code>putUShort(const char* key, uint16_t value)</code>
Int	<code>putInt(const char* key, int32_t value)</code>
Unsigned Int	<code>putUInt(const char* key, uint32_t value)</code>
Long	<code>putLong(const char* key, int32_t value)</code>
Unsigned Long	<code>putULong(const char* key, uint32_t value)</code>
Long64	<code>putLong64(const char* key, int64_t value)</code>
Unsigned Long64	<code>putULong64(const char* key, uint64_t value)</code>
Float	<code>putFloat(const char* key, const float_t value)</code>
Double	<code>putDouble(const char* key, const double_t value)</code>
Bool	<code>putBool(const char* key, const bool value)</code>
String	<code>putString(const char* key, const String value)</code>
Bytes	<code>putBytes(const char* key, const void* value, size_t len)</code>

Get a Key Value (Read Value)

Similarly, you should use different methods depending on the variable type you want to get.

Char	<code>getChar(const char* key, const int8_t defaultValue)</code>
Unsigned Char	<code>getUChar(const char* key, const uint8_t defaultValue)</code>

Short	getShort(const char* key, const int16_t defaultValue)
Unsigned Short	getUShort(const char* key, const uint16_t defaultValue)
Int	getInt(const char* key, const int32_t defaultValue)
Unsigned Int	getUInt(const char* key, const uint32_t defaultValue)
Long	getLong(const char* key, const int32_t defaultValue)
Unsigned Long	getULong(const char* key, const uint32_t defaultValue)
Long64	getLong64(const char* key, const int64_t defaultValue)
Unsigned Long64	gettULong64(const char* key, const uint64_t defaultValue)
Float	getFloat(const char* key, const float_t defaultValue)
Double	getDouble(const char* key, const double_t defaultValue)
Bool	getBool(const char* key, const bool defaultValue)
String	getString(const char* key, const String defaultValue)
String	getString(const char* key, char* value, const size_t maxlen)
Bytes	getBytes(const char* key, void * buf, size_t maxlen)

Remove a Namespace

In the Arduino implementation of `Preferences`, there is no method of completely removing a namespace. As a result, over the course of several projects, the ESP32 non-volatile storage (NVS) `Preferences` partition may become full. To completely erase and reformat the NVS memory used by `Preferences`, you can run the following sketch on your board—don't run it now as it is not needed, we're just leaving it here for future reference.

- [Click here to download the code.](#)

```
#include <nvs_flash.h>

void setup() {
    nvs_flash_erase(); // erase the NVS partition and...
    nvs_flash_init(); // initialize the NVS partition
    while(true);
}
void loop() {
```

You should download a new sketch to your board immediately after running the above, or it will reformat the NVS partition every time it is powered up.

Preferences.h – Save key:value Pairs

For a simple example of how to save and get data using `Preferences.h`, in the Arduino IDE, go to **File > Examples > Preferences > StartCounter** (make sure you have an ESP32 board selected in **Tools > Board**).

- [Click here to download the code.](#)

```
#include <Preferences.h>

Preferences preferences;

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

  // Open Preferences with my-app namespace. Each application module, library, etc
  // has to use a namespace name to prevent key name collisions. We open storage in
  // RW-mode (second parameter has to be false).
  // Note: Namespace name is limited to 15 chars.
  preferences.begin("my-app", false);

  // Remove all preferences under the opened namespace
  //preferences.clear();

  // Or remove the counter key only
  //preferences.remove("counter");

  // Get the counter value, if the key does not exist, return a default value of 0
  // Note: Key name is limited to 15 chars.
  unsigned int counter = preferences.getInt("counter", 0);

  // Increase counter by 1
  counter++;

  // Print the counter to Serial Monitor
  Serial.printf("Current counter value: %u\n", counter);

  // Store the counter to the Preferences
  preferences.putInt("counter", counter);

  // Close the Preferences
  preferences.end();

  // Wait 10 seconds
  Serial.println("Restarting in 10 seconds...");
  delay(10000);

  // Restart ESP
  ESP.restart();
```

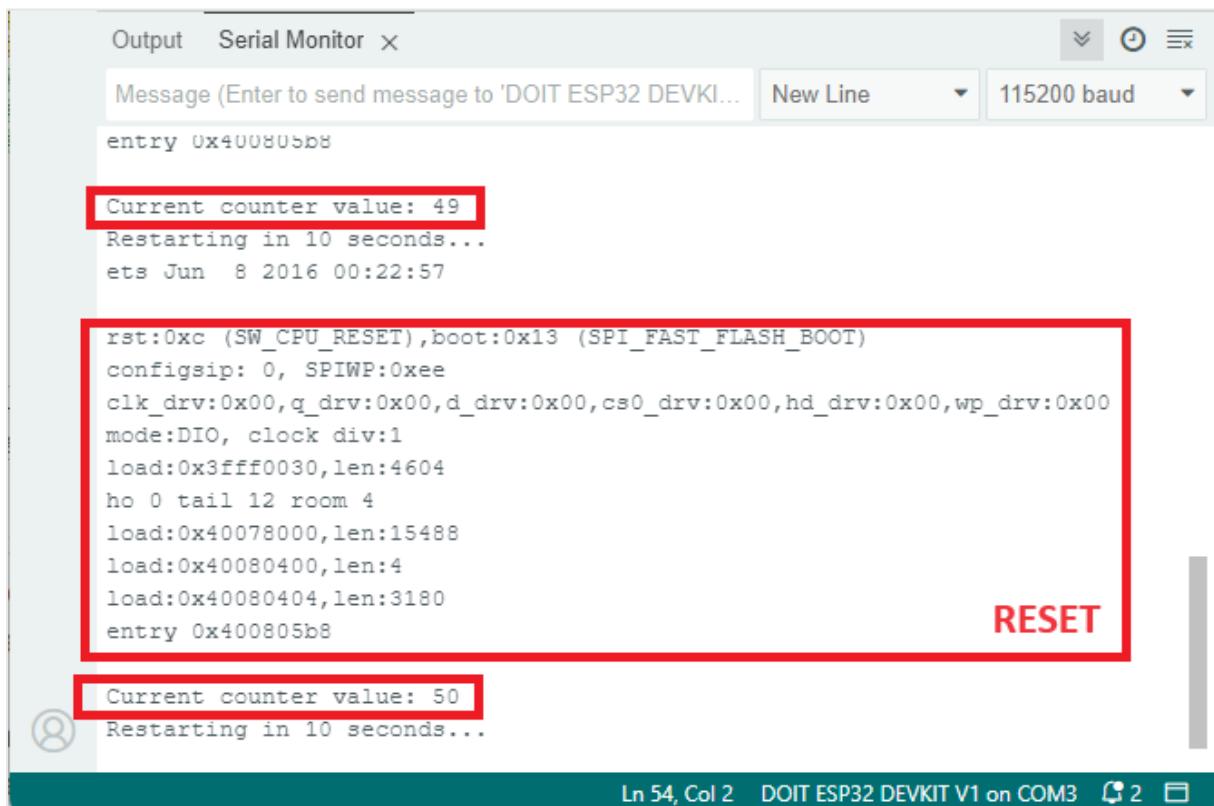
```
}

void loop() {

}
```

This example increases a variable called `counter` between resets. This illustrates that the ESP32 “remembers” the value even after a reset.

Upload the previous sketch to your ESP32 board. Open the Serial Monitor at a baud rate of 115200 and press the on-board RST button. You should see the counter variable increasing between resets.



How Does the Code Work?

This example uses the functions we've seen in the previous sections. First, include the `Preferences.h` library.

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

Then, create an instance of the library called `preferences`.

```
Preferences preferences;
```

In the `setup()`, initialize the Serial Monitor at a baud rate of 115200.

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

Create a “storage space” in the flash memory called `my-app` in read/write mode. You can give it any other name.

```
preferences.begin("my-app", false);
```

Get the value of the counter key saved on preferences. If it doesn’t find any value, it returns 0 by default (which happens when this code runs for the first time).

```
unsigned int counter = preferences.getInt("counter", 0);
```

The `counter` variable is increased by one unit every time the ESP runs:

```
counter++;
```

Print the value of the `counter` variable:

```
Serial.printf("Current counter value: %u\n", counter);
```

Store the new value on the “`counter`” key:

```
preferences.putInt("counter", counter);
```

Close the Preferences.

```
preferences.end();
```

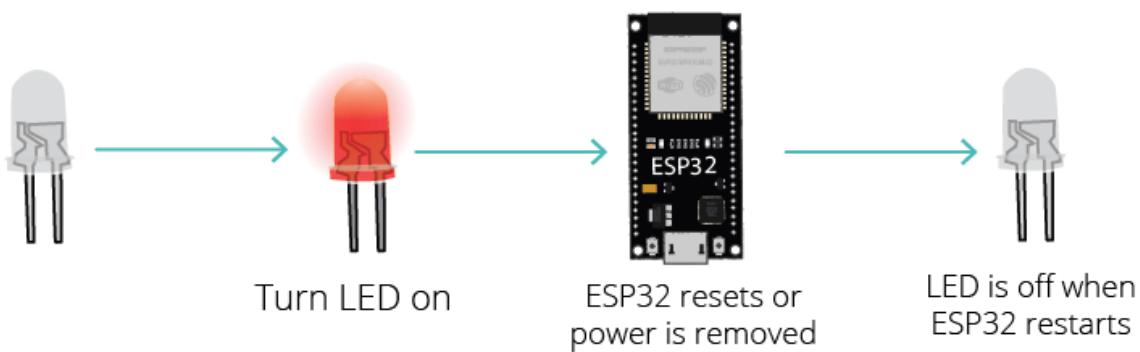
Finally, restart the ESP32 board:

```
ESP.restart();
```

Remember the Last GPIO State After RESET

Another application of the `Preferences.h` library is to save the last state of an output. For example, imagine the following scenario:

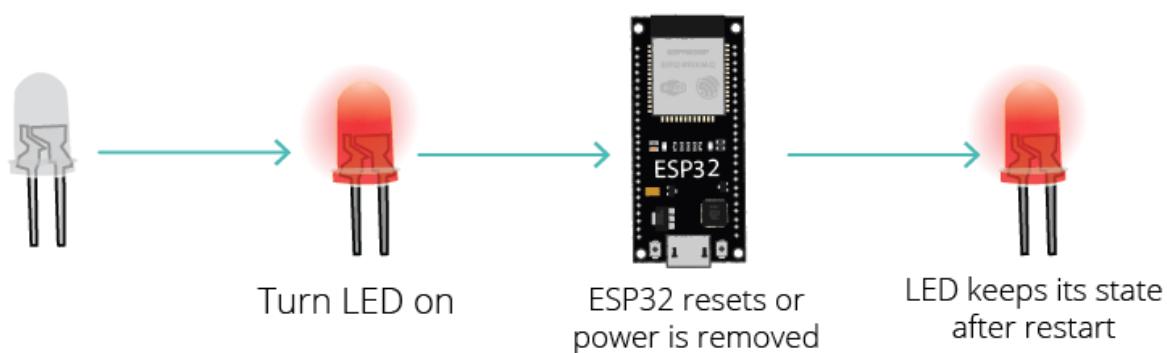
- You're controlling an output with the ESP32;
- You set your output to turn on;
- The ESP32 suddenly loses power;
- When the power comes back on, the output stays off – because it didn't keep its last state.



You don't want this to happen. You want the ESP32 to remember what was happening before losing power and return to the last state.

To solve this problem, you can save the output's state in the flash memory. Then, you need to add a condition at the beginning of your sketch to check the last output state and turn it on or off accordingly.

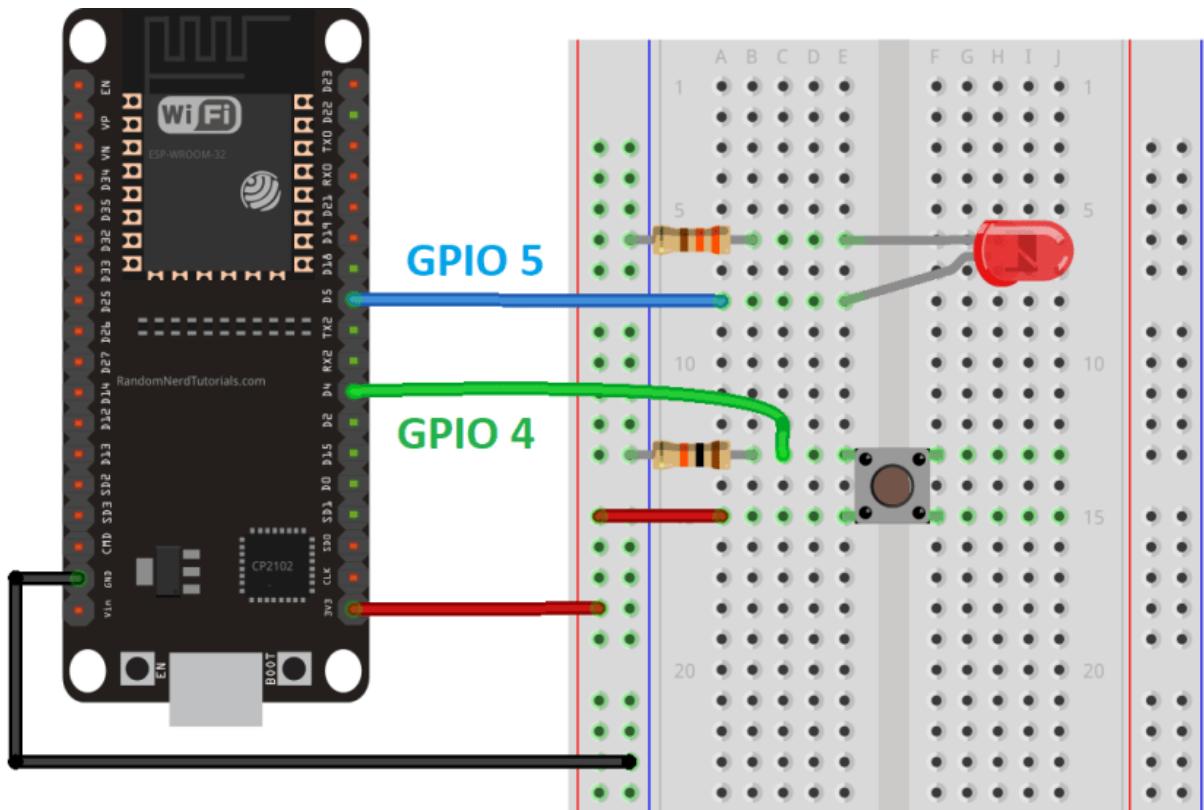
The following figure shows what we're going to do:



We'll show you an example using an LED and a pushbutton. The pushbutton controls the LED state. The LED keeps its state between resets. This means that if the LED is lit when you remove power, it will be lit when it gets powered again.

Wiring the Circuit

Wire a pushbutton and an LED to the ESP32 as shown in the following diagram.



Code

This is a debounce code that changes the LED state every time you press the pushbutton. But there's something special about this code – it remembers the last LED state, even after resetting or removing power from the ESP32. This is possible because we save the led state on Preferences whenever it changes.

- [Click here to download the code.](#)

```
#include <Preferences.h>

Preferences preferences;

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

bool ledState;
```

```

bool buttonState;
int lastButtonState = LOW;

unsigned long lastDebounceTime = 0; // the last time the output pin was toggled
unsigned long debounceDelay = 50; // the debounce time; increase if the output flickers

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

  //Create a namespace called "gpio"
  preferences.begin("gpio", false);

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

  // read the last LED state from flash memory
  ledState = preferences.getBool("state", false);
  Serial.printf("LED state before reset: %d \n", ledState);
  // set the LED to the last stored state
  digitalWrite(ledPin, ledState);
}

void loop() {
  int reading = digitalRead(buttonPin);

  if (reading != lastButtonState) {
    lastDebounceTime = millis();
  }
  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (reading != buttonState) {
      buttonState = reading;
      if (buttonState == HIGH) {
        ledState = !ledState;
      }
    }
  }
  lastButtonState = reading;
  if (digitalRead(ledPin) != ledState) {
    Serial.println("State changed");
    // change the LED state
    digitalWrite(ledPin, ledState);

    // save the LED state in flash memory
    preferences.putBool("state", ledState);

    Serial.printf("State saved: %d \n", ledState);
  }
}

```

How Does the Code Work?

Let's take a quick look at the relevant parts of code for this example.

In the `setup()`, start by creating a section in the flash memory to save the GPIO state. In this example, we've called it `gpio`.

```
preferences.begin("gpio", false);
```

Get the GPIO state saved on Preferences on the `state` key. It is a boolean variable, so use the `getBool()` function. If there isn't any `state` key yet (which happens when the ESP32 first runs), return `false` (the LED will be off).

```
ledState = preferences.getBool("state", false);
```

Print the state and set the LED to the right state:

```
// read the last LED state from flash memory
ledState = preferences.getBool("state", false);
Serial.printf("LED state before reset: %d \n", ledState);
// set the LED to the last stored state
digitalWrite(ledPin, ledState);
```

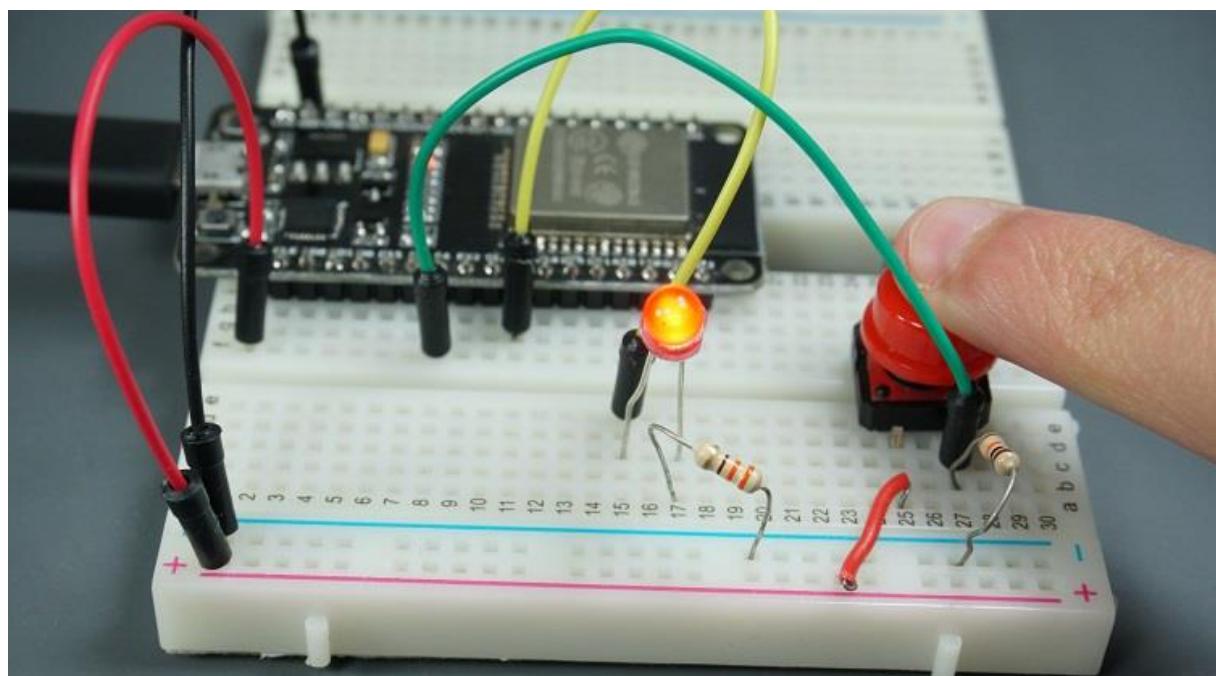
Finally, in the `loop()` update the `state` key on Preferences whenever there's a change.

```
// save the LED state in flash memory
preferences.putBool("state", ledState);
```

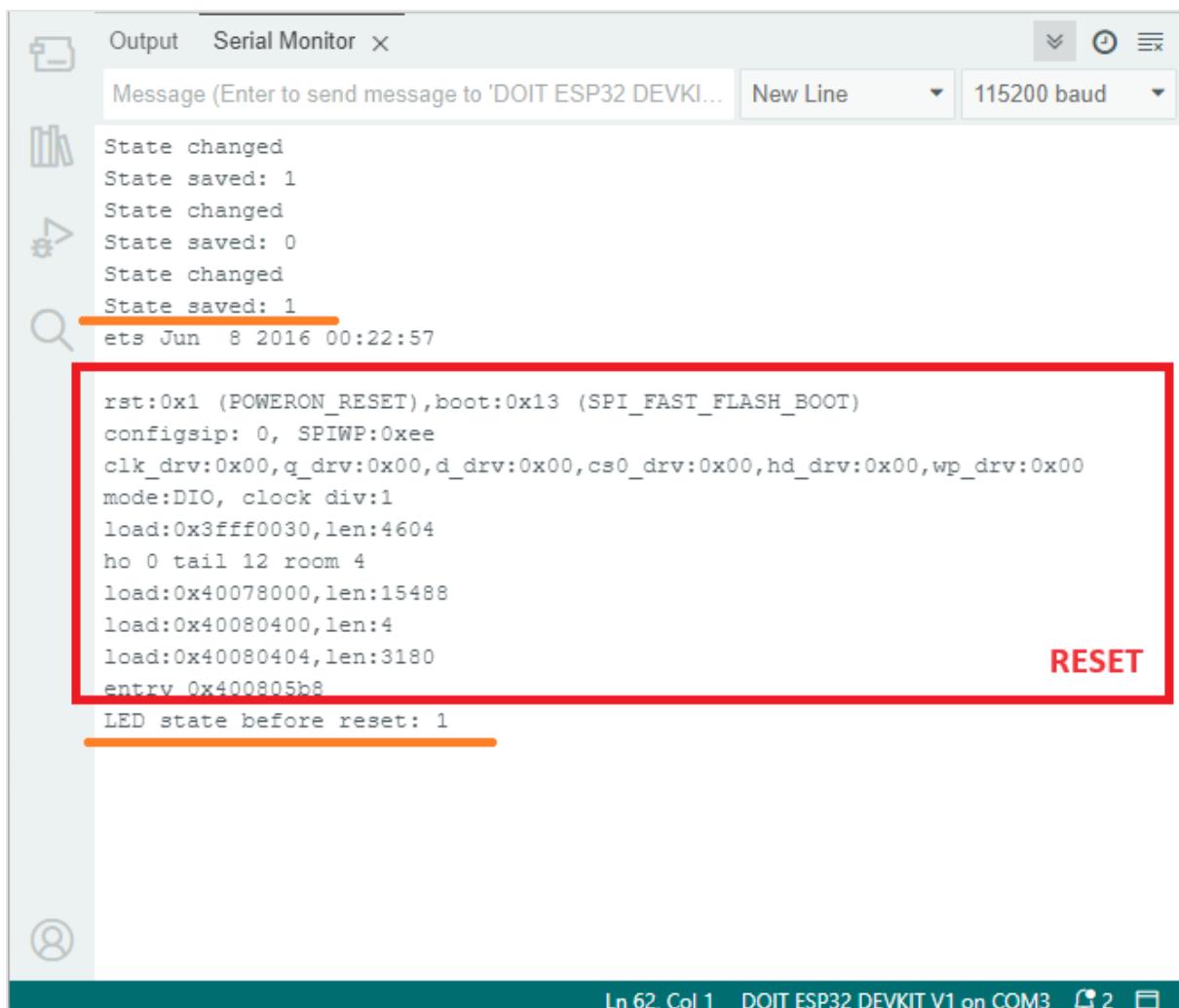
Demonstration

After wiring the circuit, upload the code to your board. Open the Serial Monitor at a baud rate of 115200 and press the on-board RST button.

Press the pushbutton to change the LED state and then remove power or press the RST button.



When the ESP32 restarts, it will read the last state saved on Preferences and set the LED to that state. It also prints a message on the Serial Monitor whenever there's a change in the GPIO state.



The screenshot shows the Arduino Serial Monitor window. The title bar says "Output" and "Serial Monitor". The message area displays the following text:

```
Message (Enter to send message to 'DOIT ESP32 DEVKIT...')  
New Line 115200 baud  
  
State changed  
State saved: 1  
State changed  
State saved: 0  
State changed  
State saved: 1  
ets Jun 8 2016 00:22:57  
  
rst:0x1 (POWERON_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:0x3fff0030,len:4604  
ho 0 tail 12 room 4  
load:0x40078000,len:15488  
load:0x40080400,len:4  
load:0x40080404,len:3180  
entry 0x400805b8  
LED state before reset: 1
```

A red rectangular box highlights the first few lines of the log, specifically the boot sequence and configuration details. The word "RESET" is written in red at the end of the highlighted text. The status bar at the bottom of the monitor shows "Ln 62, Col 1 DOIT ESP32 DEVKIT V1 on COM3" and some icons.

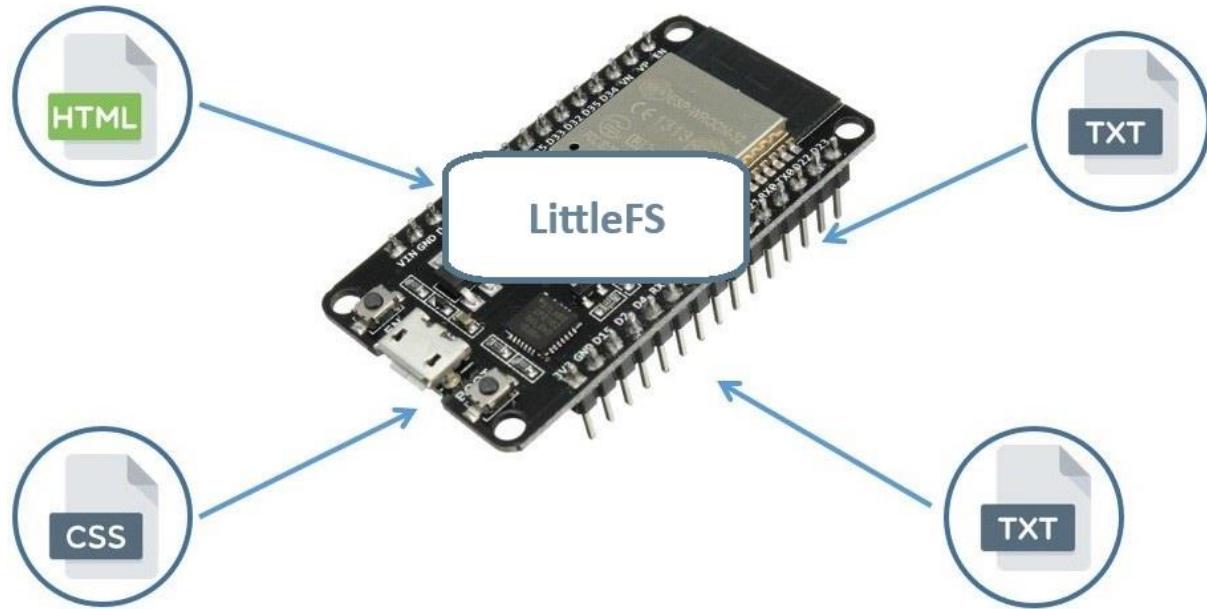
Wrapping Up

In summary, in this Unit you learned how to save data permanently on the ESP32 flash memory using the Preferences.h library. This library is handy to save key:value pairs. Data held on the flash memory remains there even after resetting the ESP32 or removing power.

If you need to store bigger amounts of data or files, you should use the ESP32 filesystem (LittleFS or SPIFFS) or a microSD card instead. We'll cover these methods in the upcoming units.

3.2 - ESP32 LittleFS Filesystem

In this Unit, we're going to show how to use the ESP32 LittleFS filesystem: how to upload files to the ESP32 flash memory, how to read data from those files, and how to write and append data to the files using the ESP32 filesystem.



Introducing LittleFS

LittleFS is a lightweight filesystem created for microcontrollers that lets you access the flash memory as you do in a standard file system on your computer, but it's simpler and more limited. You can read, write, close, and delete files. Using LittleFS with the ESP32 boards is useful to:

- Create configuration files with settings;
- Save data permanently;
- Create files to save small amounts of data instead of using a microSD card;
- Save HTML, CSS, and JavaScript files to build a web server;
- Save images, figures, and icons;
- And much more.

Installing the Arduino ESP32 Filesystem Uploader

You can create, save, and write files to the ESP32 filesystem by writing the code yourself on the Arduino IDE. This is not very useful, because you'd have to type the content of your files in the Arduino sketch.

Fortunately, there is a plugin for the Arduino IDE that allows you to upload files directly to the ESP32 filesystem from a folder in your computer. This makes it easy and simple to work with files. Let's install it.

Windows Instructions

- 1) Go to the [releases page](#) and click the **.vsix** file to download.

[Release 1.1.8 - ESP32, ESP8266, and Pico Support with ESP32 upload speed selection](#)

[Latest](#)

For the ESP32, use the new menu names to grab upload speed from the menus. Lets you upload at < 921600 baud.

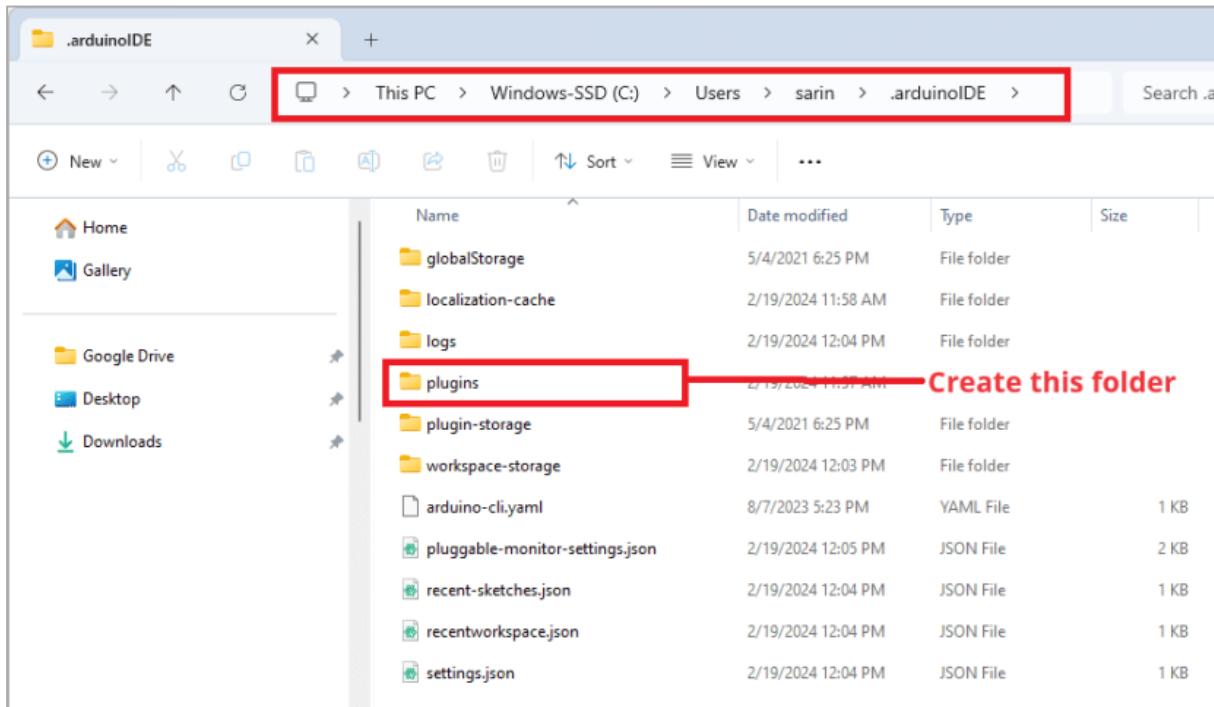
▼ Assets 3

arduino-littlefs-upload-1.1.8.vsix	1.03 MB	Jun 8
Source code (zip)		Jun 8
Source code (tar.gz)		Jun 8

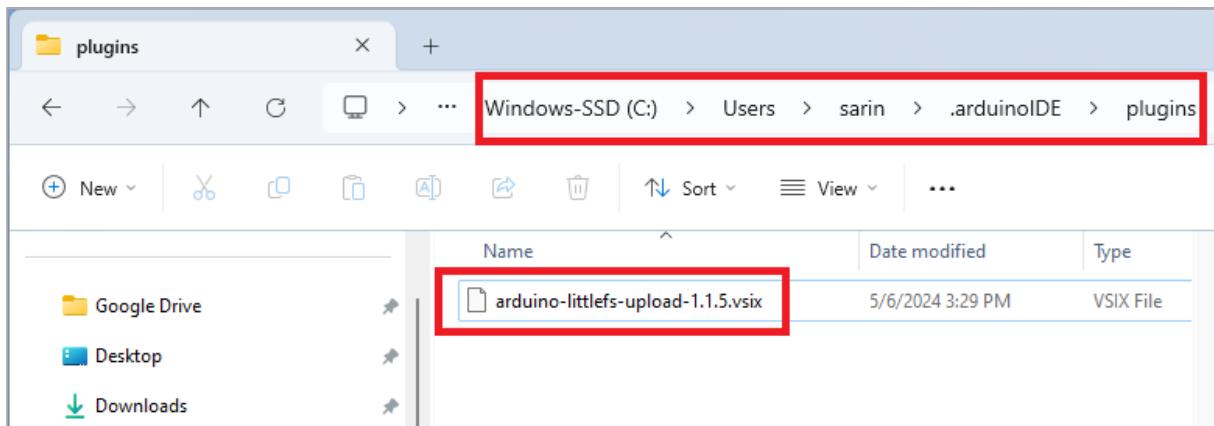
- 2) On your computer, go to the following path:

```
C:\Users\<username>\.arduinoIDE\
```

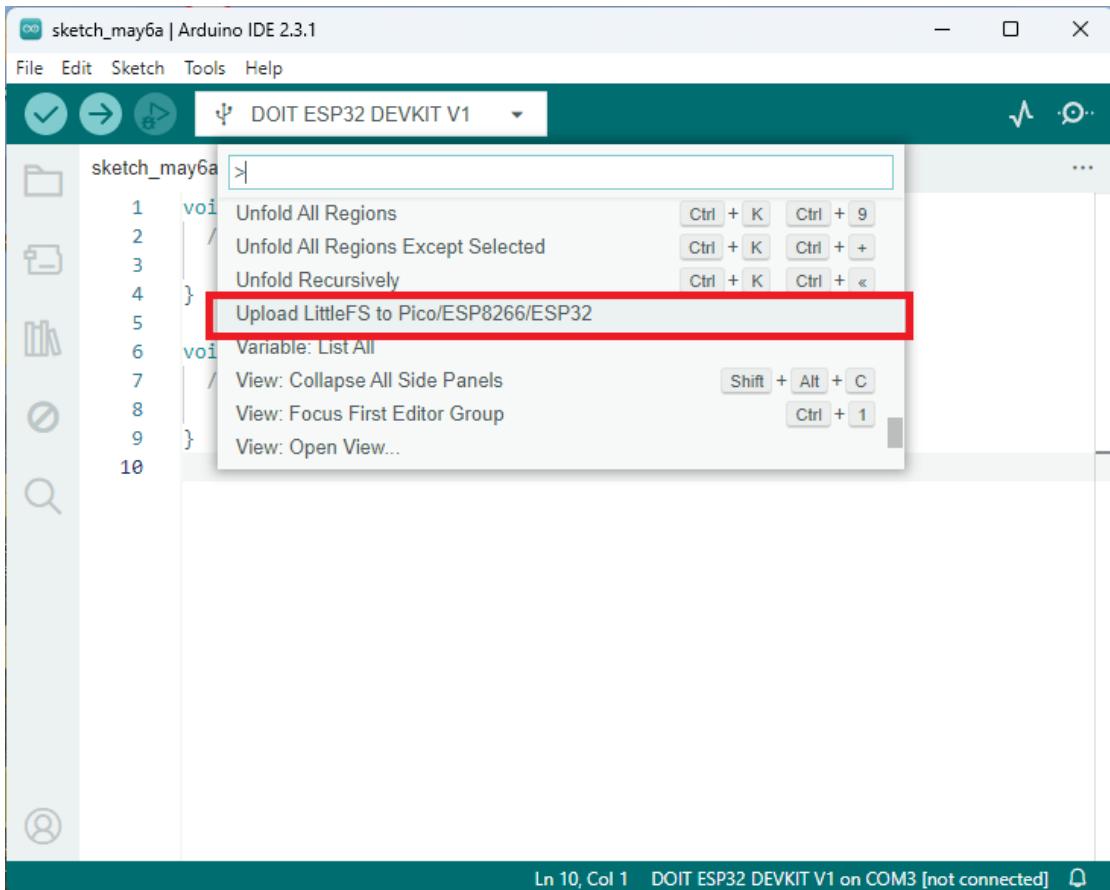
Create a new folder called **plugins** if you haven't already.



3) Move the **.vsix** file you downloaded previously to the **plugins** folder (remove any other previous versions of the same plugin if that's the case).



4) Restart or open the Arduino IDE 2. To check if the plugin was successfully installed, press **[Ctrl] + [Shift] + [P]** to open the command palette. An instruction called '**Upload Little FS to Pico/ESP8266/ESP32**' should be there (just scroll down or search for the name of the instruction).



That means the plugin was installed successfully. Next, we'll test if the filesystem uploader plugin is working properly.

Mac OS X Instructions

Follow the next steps to install the filesystem uploader if you're using Mac OS X:

- 1) Go to the [releases page](#) and click the **.vsix** file to download.

[Release 1.1.8 - ESP32, ESP8266, and Pico Support with ESP32 upload speed selection](#)

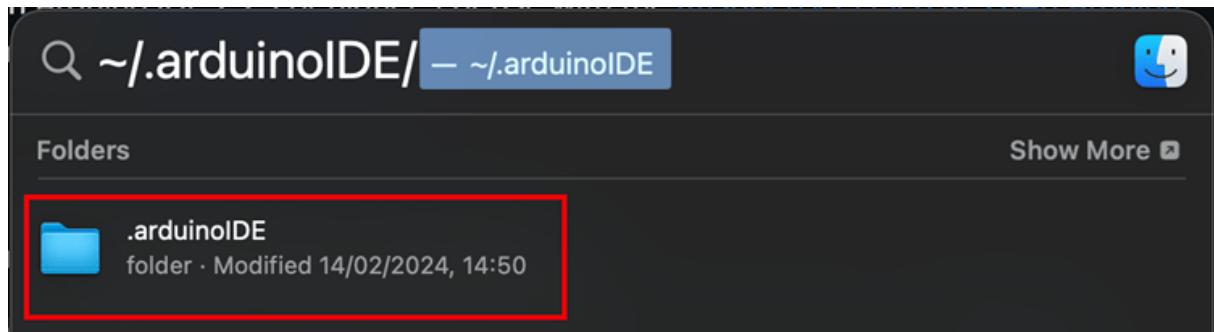
[Latest](#)

For the ESP32, use the new menu names to grab upload speed from the menus. Lets you upload at < 921600 baud.

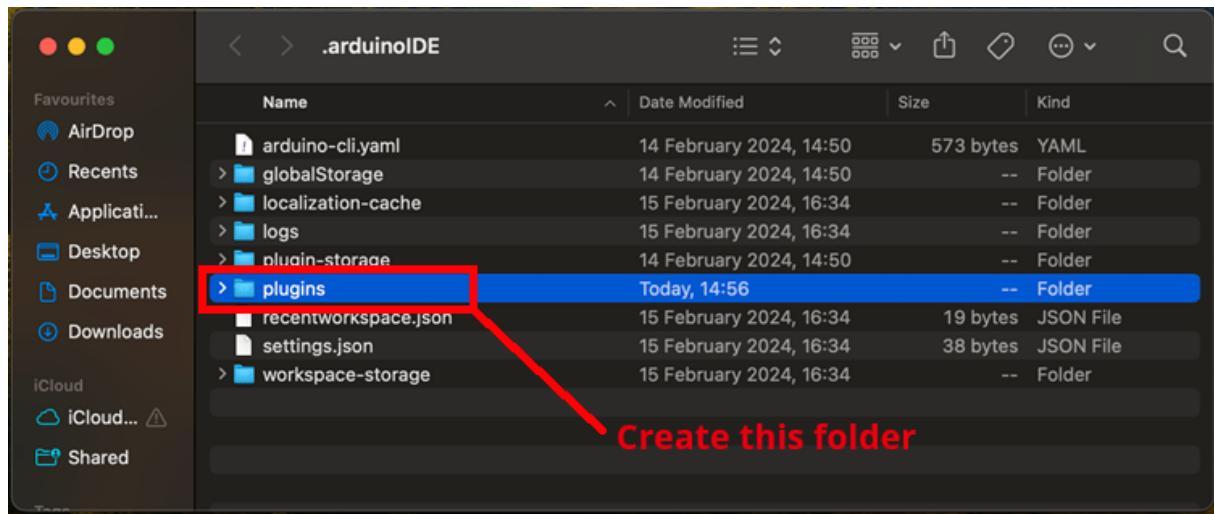
Assets 3

arduino-littlefs-upload-1.1.8.vsix	1.03 MB	Jun 8
Source code (zip)		Jun 8
Source code (tar.gz)		Jun 8

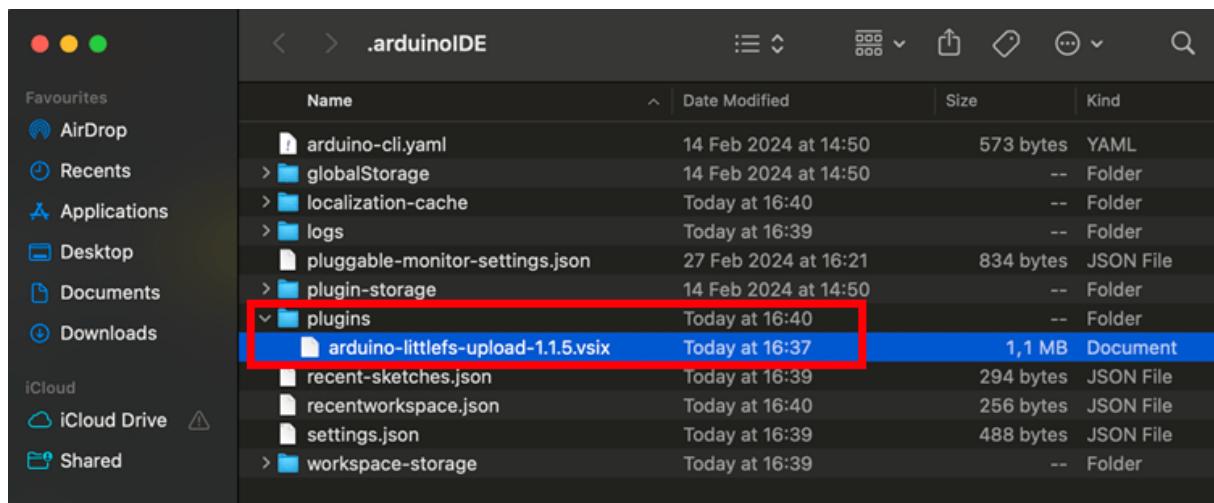
2) In Finder, type `~/arduinoIDE/` and open that directory.



3) Create a new folder called **plugins**.

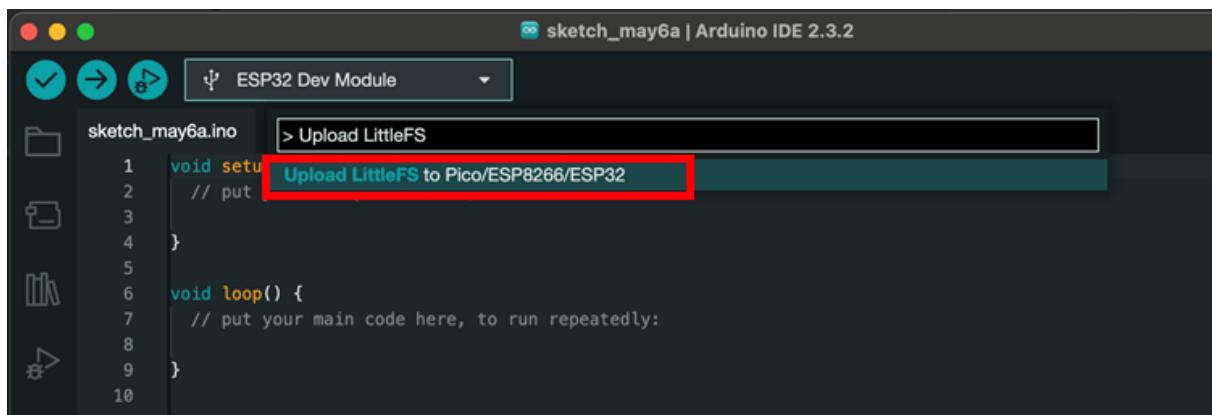


4) Move the **.vsix** file to the **plugins** folder (remove any other previous versions of the same plugin if that's the case).



5) Restart or open the Arduino IDE 2. To check if the plugin was successfully installed, press **[⌘]** + **[Shift]** + **[P]** to open the command palette. An instruction

called '**Upload LittleFS to Pico/ESP8266/ESP32**' should be there (just scroll down or search for the name of the instruction).

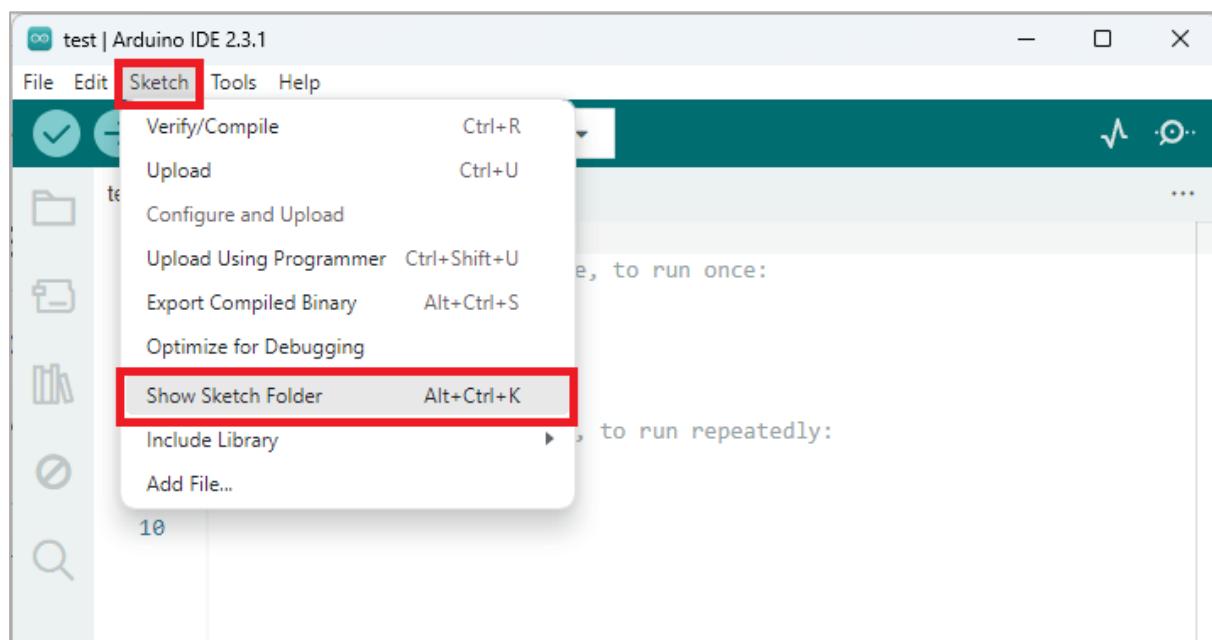


A screenshot of the Arduino IDE interface. The title bar says 'sketch_may6a | Arduino IDE 2.3.2'. Below it, a toolbar has icons for checkmark, upload, and refresh. A dropdown menu shows 'ESP32 Dev Module'. The main area shows a code editor with 'sketch_may6a.ino' open. The code contains:1 void setup() {
2 // put your setup code here, to run once:
3 }
4
5 void loop() {
6 // put your main code here, to run repeatedly:
7 }
8
9 }
10The line 'Upload LittleFS to Pico/ESP8266/ESP32' is highlighted with a red box.

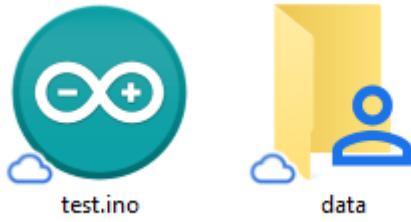
Uploading Files using the Filesystem Uploader

To upload files to the ESP32 filesystem, follow the next instructions.

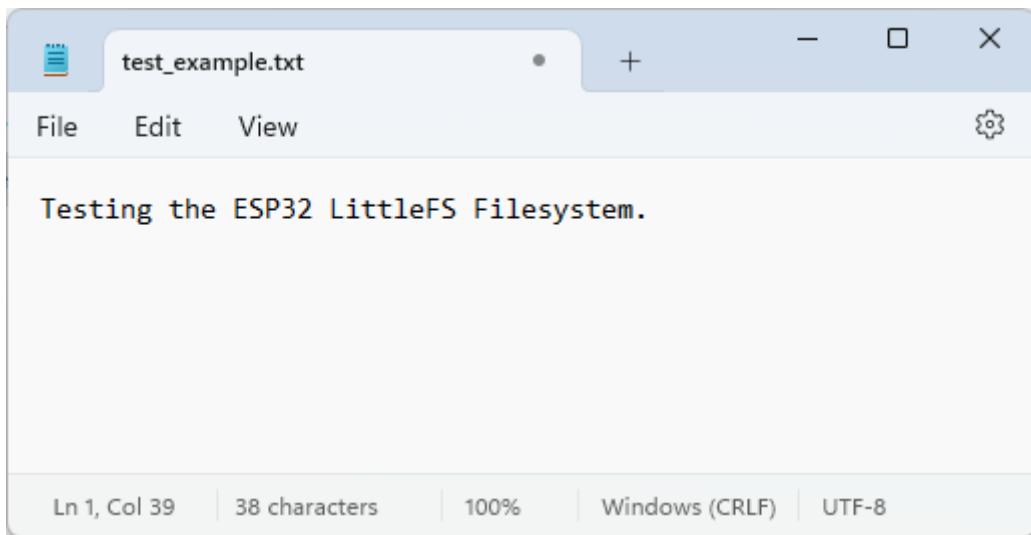
- 1) Create an Arduino sketch and save it. For demonstration purposes, you can save an empty sketch.
- 2) Then, open the sketch folder. You can go to **Sketch > Show Sketch Folder**. The folder where your sketch is saved should open.



- 3) Inside that folder, create a new folder called **data**.



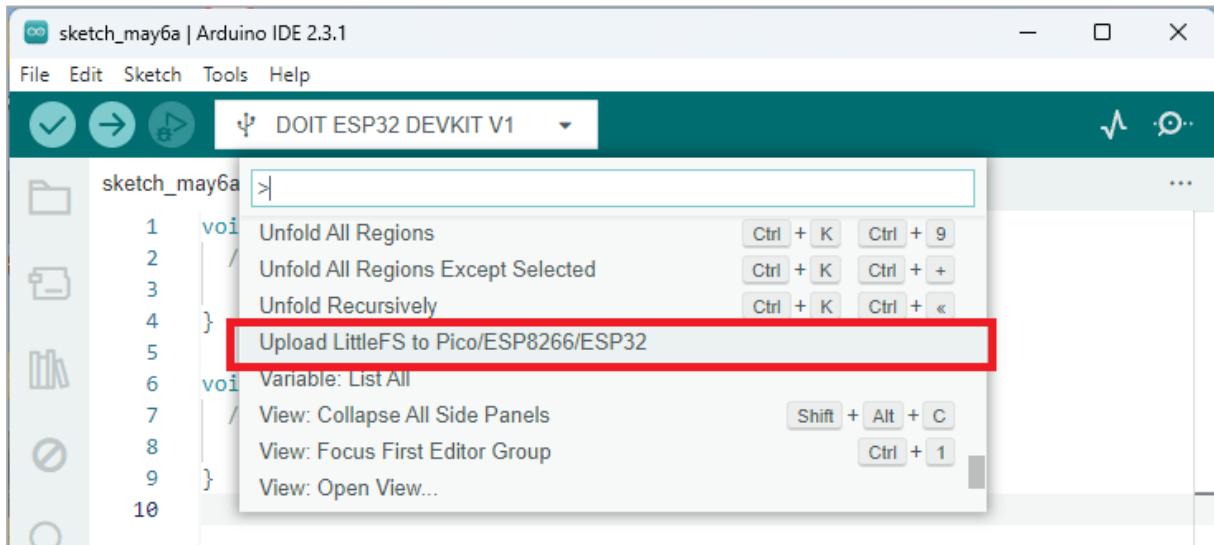
4) Inside the data folder is where you should put the files you want to upload to the ESP32 filesystem. As an example, create a **.txt** file with some text called **test_example.txt** (and save it inside the **data** folder).



5) Make sure you have the right board (**Tools > Board**) and COM port selected (**Tools > Port**).

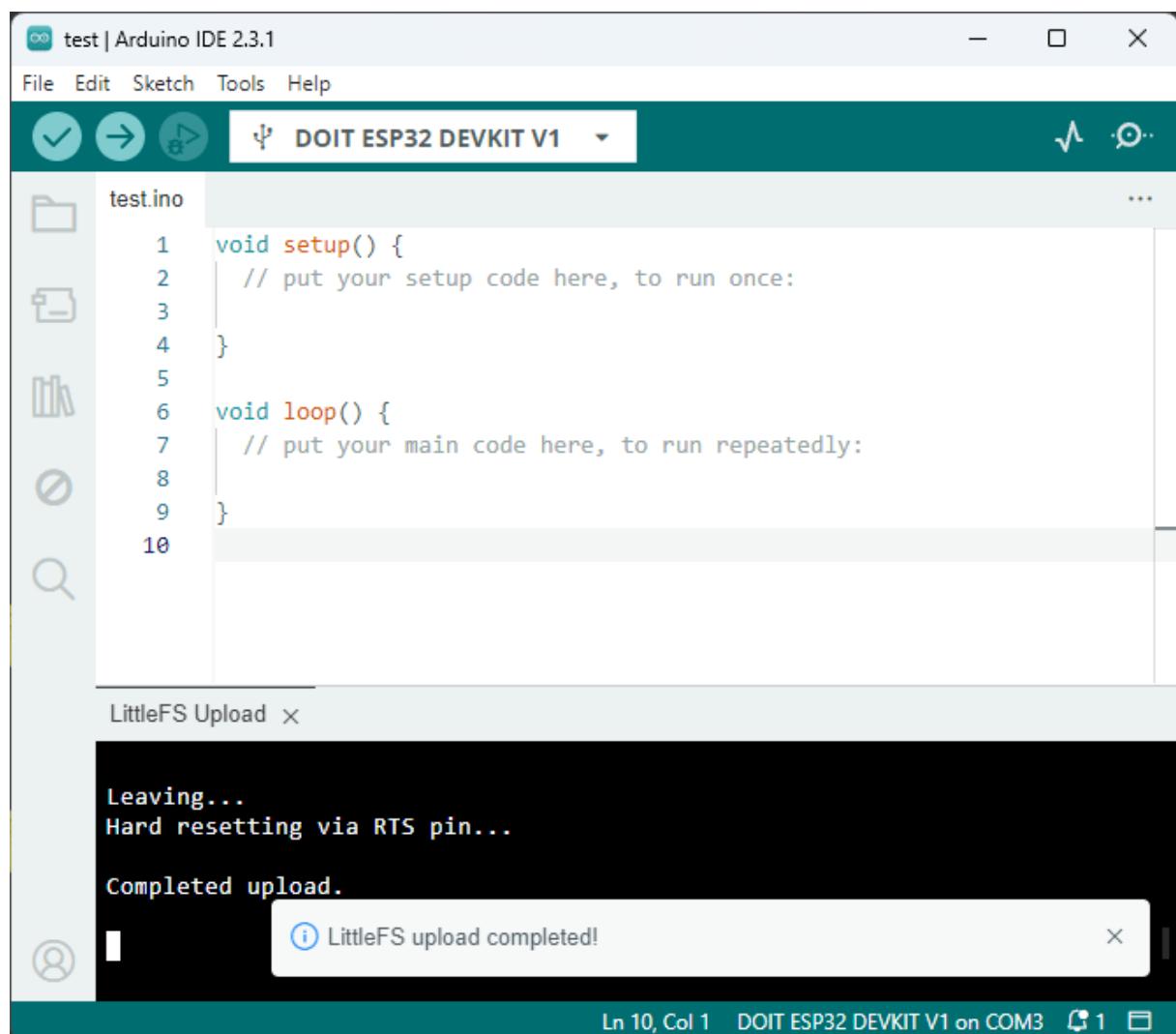
6) Depending on the ESP32 board selected, you may need to select the desired flash size (some boards don't have that option, don't worry). In the Arduino IDE, in **Tools > Flash size**, select the desired flash size (this will depend on the size of your files).

7) Then, upload the files to the ESP32 board. Press **[Ctrl] + [Shift] + [P]** on Windows or **[⌘] + [Shift] + [P]** on MacOS to open the command palette. Search for the **Upload LittleFS to Pico/ESP8266/ESP32** command and click on it.



Important: ensure the Serial Monitor is closed. Otherwise, the upload will fail.

After a few seconds, you should get the message "**Completed upload.**". The files were successfully uploaded to the ESP32 filesystem.



Troubleshooting

If you get the following error message “[ERROR: No port specified, check IDE menus](#)”, restart the Arduino IDE, and try again.

Note: in some ESP32 development boards you need to keep the ESP32 on-board “BOOT” button pressed while it’s uploading the files. When you see the “**Connecting—.....**” message, you need to press the ESP32 on-board “BOOT” button.

Testing the Uploader

Now, let’s just check if the file was actually saved into the ESP32 filesystem. Simply upload the following code to your ESP32 board.

- [Click here to download the code.](#)

```
#include "LittleFS.h"

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

  if(!LittleFS.begin()){
    Serial.println("An Error has occurred while mounting LittleFS");
    return;
  }

  File file = LittleFS.open("/test_example.txt", "r");
  if(!file){
    Serial.println("Failed to open file for reading");
    return;
  }

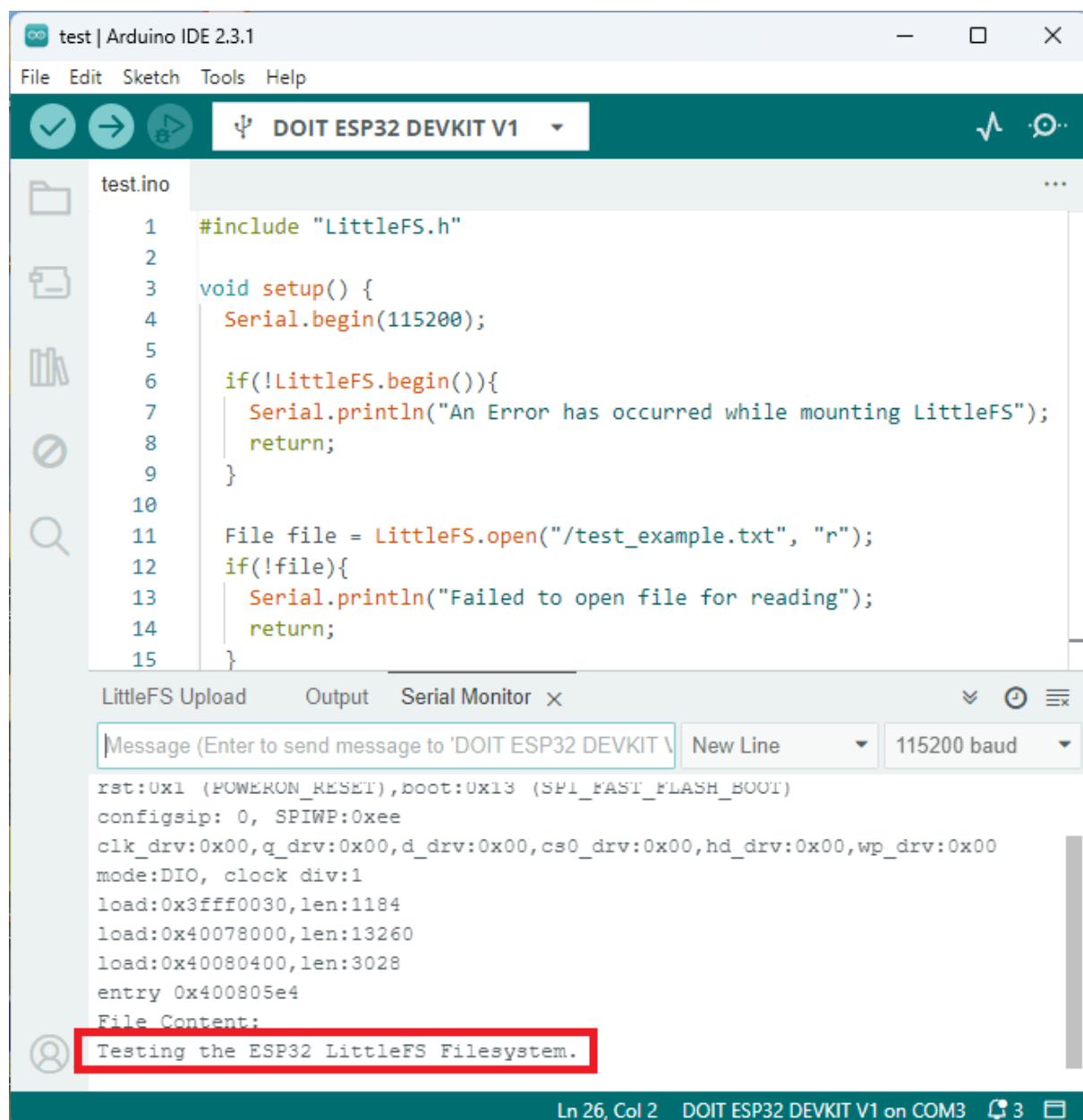
  Serial.println("File Content:");
  while(file.available()){
    Serial.write(file.read());
  }
  file.close();
}

void loop() {
```

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



Press the ESP32 on-board “RST” button. It should print the content of your **.txt** file in the Serial Monitor.



The screenshot shows the Arduino IDE 2.3.1 interface with the DOIT ESP32 DEVKIT V1 plugin selected. A sketch named "test.ino" is open, containing code to test the LittleFS filesystem. The Serial Monitor window displays the output of the uploaded code, which includes configuration details and the content of the "test_example.txt" file, which is "Testing the ESP32 LittleFS Filesystem." The message "Testing the ESP32 LittleFS Filesystem." is highlighted with a red box.

```
1 #include "LittleFS.h"
2
3 void setup() {
4     Serial.begin(115200);
5
6     if(!LittleFS.begin()){
7         Serial.println("An Error has occurred while mounting LittleFS");
8         return;
9     }
10
11     File file = LittleFS.open("/test_example.txt", "r");
12     if(!file){
13         Serial.println("Failed to open file for reading");
14         return;
15     }
```

LittleFS Upload Output Serial Monitor ▼ ⌂ ⌂ ⌂

Message (Enter to send message to 'DOIT ESP32 DEVKIT V1') New Line ▾ 115200 baud ▾

```
rst:0x1 (POWERON_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:0x3fff0030,len:1184
load:0x40078000,len:13260
load:0x40080400,len:3028
entry 0x400805e4
File Content:
Testing the ESP32 LittleFS Filesystem.
```

Ln 26, Col 2 DOIT ESP32 DEVKIT V1 on COM3 ⌂ 3

Congratulations! You've successfully uploaded files to the ESP32 LittleFS filesystem using the plugin.

Manipulating Files on the Filesystem

In this section, we'll show you how to manipulate files on the filesystem using code on the Arduino IDE. You can write to the files, append data to the files, read the files' content, delete files, create new files and folders, and check the file size. The following example shows how to do practically any task that you may need when dealing with files and folders.

- [Click here to download the code.](#)

This code was adapted from the official example that [you can find here](#).

```
#include <Arduino.h>
#include "FS.h"
#include <LittleFS.h>

// You only need to format LittleFS the first time you run a
// test or else use the LITTLEFS plugin to create a partition
// https://github.com/lorol/arduino-esp32littlefs-plugin

#define FORMAT_LITTLEFS_IF_FAILED true

void listDir(fs::FS &fs, const char * dirname, uint8_t levels){
    Serial.printf("Listing directory: %s\r\n", dirname);

    File root = fs.open(dirname);
    if(!root){
        Serial.println("- failed to open directory");
        return;
    }
    if(!root.isDirectory()){
        Serial.println(" - not a directory");
        return;
    }

    File file = root.openNextFile();
    while(file){
        if(file.isDirectory()){
            Serial.print(" DIR : ");
            Serial.println(file.name());
            if(levels){
                listDir(fs, file.path(), levels -1);
            }
        } else {
            Serial.print(" FILE: ");
            Serial.print(file.name());
            Serial.print("\tSIZE: ");
            Serial.println(file.size());
        }
        file = root.openNextFile();
    }
}

void createDir(fs::FS &fs, const char * path){
    Serial.printf("Creating Dir: %s\n", path);
    if(fs.mkdir(path)){
        Serial.println("Dir created");
    } else {
        Serial.println("mkdir failed");
    }
}

void removeDir(fs::FS &fs, const char * path){
    Serial.printf("Removing Dir: %s\n", path);
    if(fs.rmdir(path)){
        Serial.println("Dir removed");
    } else {
        Serial.println("rmdir failed");
    }
}
```

```

    }

}

void readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return;
    }

    Serial.println("- read from file:");
    while(file.available()){
        Serial.write(file.read());
    }
    file.close();
}

void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
    file.close();
}

void appendFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Appending to file: %s\r\n", path);

    File file = fs.open(path, FILE_APPEND);
    if(!file){
        Serial.println("- failed to open file for appending");
        return;
    }
    if(file.print(message)){
        Serial.println("- message appended");
    } else {
        Serial.println("- append failed");
    }
    file.close();
}

void renameFile(fs::FS &fs, const char * path1, const char * path2){
    Serial.printf("Renaming file %s to %s\r\n", path1, path2);
    if (fs.rename(path1, path2)) {
        Serial.println("- file renamed");
    } else {
        Serial.println("- rename failed");
    }
}

void deleteFile(fs::FS &fs, const char * path){
}

```

```

Serial.printf("Deleting file: %s\r\n", path);
if(fs.remove(path)){
    Serial.println("- file deleted");
} else {
    Serial.println("- delete failed");
}

void testFileIO(fs::FS &fs, const char * path){
    Serial.printf("Testing file I/O with %s\r\n", path);

    static uint8_t buf[512];
    size_t len = 0;
    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }

    size_t i;
    Serial.print("- writing" );
    uint32_t start = millis();
    for(i=0; i<2048; i++){
        if ((i & 0x001F) == 0x001F){
            Serial.print(".");
        }
        file.write(buf, 512);
    }
    Serial.println("");
    uint32_t end = millis() - start;
    Serial.printf("- %u bytes written in %u ms\r\n", 2048 * 512, end);
    file.close();

    file = fs.open(path);
    start = millis();
    end = start;
    i = 0;
    if(file && !file.isDirectory()){
        len = file.size();
        size_tflen = len;
        start = millis();
        Serial.print("- reading" );
        while(len){
            size_t toRead = len;
            if(toRead > 512){
                toRead = 512;
            }
            file.read(buf, toRead);
            if ((i++ & 0x001F) == 0x001F){
                Serial.print(".");
            }
            len -= toRead;
        }
        Serial.println("");
        end = millis() - start;
        Serial.printf("- %u bytes read in %u ms\r\n",flen, end);
        file.close();
    } else {
        Serial.println("- failed to open file for reading");
    }
}

```

```

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

    if(!LittleFS.begin(FORMAT_LITTLEFS_IF_FAILED)){
        Serial.println("LittleFS Mount Failed");
        return;
    }
    // Create a mydir folder
    createDir(LittleFS, "/mydir");
    // Create a hello1.txt file with the content "Hello1"
    writeFile(LittleFS, "/mydir/hello1.txt", "Hello1");
    // List the directories up to one level beginning at the root directory
    listDir(LittleFS, "/", 1);
    // Delete the previously created file
    deleteFile(LittleFS, "/mydir/hello1.txt");
    // Delete the previously created folder
    removeDir(LittleFS, "/mydir");
    // list all directories to make sure they were deleted
    listDir(LittleFS, "/", 1);
    // Create and write a new file in the root directory
    writeFile(LittleFS, "/hello.txt", "Hello ");
    // Append some text to the previous file
    appendFile(LittleFS, "/hello.txt", "World!\r\n");
    readFile(LittleFS, "/hello.txt"); // Read the complete file
    renameFile(LittleFS, "/hello.txt", "/foo.txt"); // Rename the previous file
    readFile(LittleFS, "/foo.txt"); // Read the file with the new name
    deleteFile(LittleFS, "/foo.txt"); // Delete the file
    testFileIO(LittleFS, "/test.txt"); // Testin
    deleteFile(LittleFS, "/test.txt"); // Delete the file

    Serial.println( "Test complete" );
}

void loop(){}

```

How the Does the Code Work?

First, you need to include the following libraries: FS.h to handle files, and LittleFS.h to create and access the filesystem.

```
#include "FS.h"
#include <LittleFS.h>
```

The first time you use LittleFS on the ESP32, you need to format it so that it creates a partition dedicated to that filesystem. To do that, we have the following Boolean variable to control whether we want to format the filesystem or not.

```
#define FORMAT_LITTLEFS_IF_FAILED true
```

The example provides several functions to handle files on the LittleFS filesystem. Let's take a look at them.

List a directory

The `listDir()` function lists the directories on the filesystem. This function accepts as arguments the filesystem (`LittleFS`), the main directory's name, and the levels to go into the directory.

```
void listDir(fs::FS &fs, const char * dirname, uint8_t levels){
    Serial.printf("Listing directory: %s\r\n", dirname);

    File root = fs.open(dirname);
    if(!root){
        Serial.println("- failed to open directory");
        return;
    }
    if(!root.isDirectory()){
        Serial.println(" - not a directory");
        return;
    }

    File file = root.openNextFile();
    while(file){
        if(file.isDirectory()){
            Serial.print(" DIR : ");
            Serial.println(file.name());
            if(levels){
                listDir(fs, file.path(), levels -1);
            }
        } else {
            Serial.print(" FILE: ");
            Serial.print(file.name());
            Serial.print("\tSIZE: ");
            Serial.println(file.size());
        }
        file = root.openNextFile();
    }
}
```

Here's an example of how to call this function. The `/` corresponds to the root directory. The following command will list all the directories up to one level starting at the root directory.

```
// List the directories up to one level beginning at the root directory
listDir(LittleFS, "/", 1);
```

Create a Directory

The `createDir()` function creates a new directory. Pass as an argument the `LittleFS` filesystem and the directory name path.

```
void createDir(fs::FS &fs, const char * path){
    Serial.printf("Creating Dir: %s\n", path);
    if(fs.mkdir(path)){
        Serial.println("Dir created");
    } else {
        Serial.println("mkdir failed");
    }
}
```

For example, the following command creates a new directory (folder) on the root called `mydir`.

```
createDir(LittleFS, "/mydir"); // Create a mydir folder
```

Remove a Directory

To remove a directory from the filesystem, use the `removeDir()` function and pass as an argument the `LittleFS` filesystem and the directory name path.

```
void removeDir(fs::FS &fs, const char * path){
    Serial.printf("Removing Dir: %s\n", path);
    if(fs.rmdir(path)){
        Serial.println("Dir removed");
    } else {
        Serial.println("rmdir failed");
    }
}
```

Here is an example that deletes the `mydir` folder.

```
removeDir(LittleFS, "/mydir");
```

The `readFile()` function reads the content of a file and prints the content in the Serial Monitor. As with previous functions, pass as an argument the `LittleFS` filesystem and the file path.

```
void readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return;
    }

    Serial.println("- read from file:");
    while(file.available()){
        Serial.write(file.read());
    }
    file.close();
}
```

For example, the following line reads the content of the `hello.txt` file.

```
readFile(LittleFS, "/hello.txt"); // Read the complete file
```

Write Content to a File

To write content to a file, you can use the `writeFile()` function. Pass as an argument, the `LittleFS` filesystem, the file path, and the message (as a `const char` variable).

```
void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
    file.close();
}
```

The following line writes Hello in the `hello.txt` file.

```
writeFile(LittleFS, "/hello.txt", "Hello ");
```

Append Content to a File

Similarly, you can append content to a file (without overwriting previous content) using the `appendFile()` function.

```
void appendFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Appending to file: %s\r\n", path);

    File file = fs.open(path, FILE_APPEND);
    if(!file){
        Serial.println("- failed to open file for appending");
        return;
    }
    if(file.print(message)){
        Serial.println("- message appended");
    } else {
        Serial.println("- append failed");
    }
    file.close();
}
```

The following line appends the message `World!\r\n` in the `hello.txt` file. The `\r\n` means that the next time you write something to the file, it will be written in a new line.

```
appendFile(LittleFS, "/hello.txt", "World!\r\n");
```

Rename a File

You can rename a file using the `renameFile()` function. Pass as arguments the LittleFS filesystem, the original filename, and the new filename.

```
void renameFile(fs::FS &fs, const char * path1, const char * path2){
    Serial.printf("Renaming file %s to %s\r\n", path1, path2);
    if (fs.rename(path1, path2)) {
        Serial.println("- file renamed");
    } else {
        Serial.println("- rename failed");
    }
}
```

The following line renames the `hello.txt` file to `foo.txt`.

```
renameFile(LittleFS, "/hello.txt", "/foo.txt");
```

Delete a File

Use the `deleteFile()` function to delete a file. Pass as an argument the LittleFS filesystem and the file path of the file you want to delete.

```
void deleteFile(fs::FS &fs, const char * path){
    Serial.printf("Deleting file: %s\r\n", path);
    if(fs.remove(path)){
        Serial.println("- file deleted");
    } else {
        Serial.println("- delete failed");
    }
}
```

The following line deletes the `foo.txt` file from the filesystem.

```
deleteFile(LittleFS, "/foo.txt"); //Delete the file
```

Test a File

The `testFileIO()` function shows how long it takes to read the content of a file.

```
void testFileIO(fs::FS &fs, const char * path){
    Serial.printf("Testing file I/O with %s\r\n", path);

    static uint8_t buf[512];
```

```

size_t len = 0;
File file = fs.open(path, FILE_WRITE);
if(!file){
    Serial.println("- failed to open file for writing");
    return;
}

(...)

    file.close();
} else {
    Serial.println("- failed to open file for reading");
}
}

```

The following function tests the **test.txt** file.

```
testFileIO(LittleFS, "/test.txt");
```

Initialize the Filesystem

In the **setup()**, the following lines initialize the **LittleFS** filesystem.

```

if(!LittleFS.begin(FORMAT_LITTLEFS_IF_FAILED)){
    Serial.println("LittleFS Mount Failed");
    return;
}

```

The **LittleFS.begin()** function returns **true** if the filesystem is initialized successfully or **false** if it isn't.

You can pass **true** or **false** as an argument to the **begin()** method. If you pass **true** it will format the **LittleFS** filesystem if the initialization fails.

Because this is the first test we're running, we set the **FORMAT_LITTLEFS_IF_FAILED** variable to **true**.

Testing the Filesystem

The following lines call all the functions we've seen previously.

```

// Create a mydir folder
createDir(LittleFS, "/mydir");
// Create a hello1.txt file with the content "Hello1"
writeFile(LittleFS, "/mydir/hello1.txt", "Hello1");
// List the directories up to one level beginning at the root directory
listDir(LittleFS, "/", 1);
// Delete the previously created file
deleteFile(LittleFS, "/mydir/hello1.txt");
// Delete the previously created folder
removeDir(LittleFS, "/mydir");
// list all directories to make sure they were deleted
listDir(LittleFS, "/", 1);

```

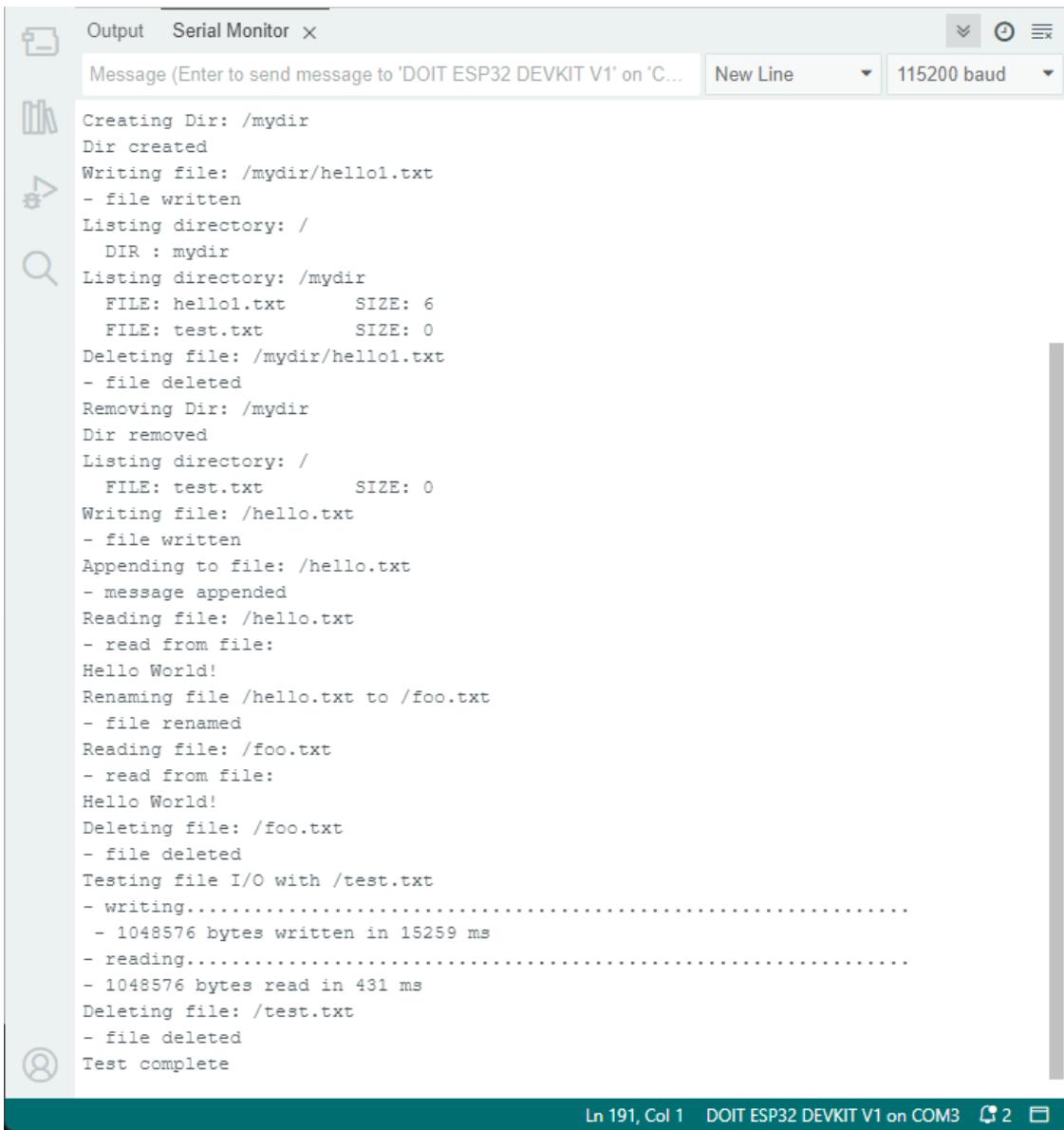
```

// Create and write a new file in the root directory
writeFile(LittleFS, "/hello.txt", "Hello ");
// Append some text to the previous file
appendFile(LittleFS, "/hello.txt", "World!\r\n");
readFile(LittleFS, "/hello.txt"); // Read the complete file
renameFile(LittleFS, "/hello.txt", "/foo.txt"); // Rename the previous file
readFile(LittleFS, "/foo.txt"); // Read the file with the new name
deleteFile(LittleFS, "/foo.txt"); // Delete the file
testFileIO(LittleFS, "/test.txt"); // Testin
deleteFile(LittleFS, "/test.txt"); // Delete the file

```

Demonstration

Upload the previous sketch to your ESP32 board. After that, open the Serial Monitor and press the ESP32 on-board RST button. If the initialization succeeds, you'll get similar messages on the Serial Monitor.



The screenshot shows the Arduino Serial Monitor window. The title bar says "Serial Monitor". The message area displays the following log output:

```

Creating Dir: /mydir
Dir created
Writing file: /mydir/hello1.txt
- file written
Listing directory: /
DIR : mydir
Listing directory: /mydir
FILE: hello1.txt      SIZE: 6
FILE: test.txt        SIZE: 0
Deleting file: /mydir/hello1.txt
- file deleted
Removing Dir: /mydir
Dir removed
Listing directory: /
FILE: test.txt        SIZE: 0
Writing file: /hello.txt
- file written
Appending to file: /hello.txt
- message appended
Reading file: /hello.txt
- read from file:
Hello World!
Renaming file /hello.txt to /foo.txt
- file renamed
Reading file: /foo.txt
- read from file:
Hello World!
Deleting file: /foo.txt
- file deleted
Testing file I/O with /test.txt
- Writing.....
- 1048576 bytes written in 15259 ms
- reading.....
- 1048576 bytes read in 431 ms
Deleting file: /test.txt
- file deleted
Test complete

```

The status bar at the bottom shows "Ln 191, Col 1 DOIT ESP32 DEVKIT V1 on COM3" and icons for volume, signal strength, and battery.

Now that you are more familiar with the ESP32 LittleFS filesystem, you can proceed to the next unit to learn how to save variable's values in a file.

3.3 - LittleFS: Save Variables' Values in a File

The previous unit illustrated almost all the operations you might need to do when dealing with files and folders on the filesystem. In this Unit, we'll take a look at a simpler and more specific example: how to save the content of a variable to the filesystem.

When you save the content of a variable to a file in the ESP32 filesystem, its value remains saved even after the ESP32 loses power or between resets. So, it's a great way to save variable's values permanently or for datalogging projects.

Save Variable's Values to a File - Code

In this example, we'll continuously save the value of a variable to the filesystem. For demonstration purposes, we'll save a random number, but this can be easily adjusted to save sensor readings, for example.

- [Click here to download the code.](#)

```
#include <Arduino.h>
#include "FS.h"
#include <LittleFS.h>

#define FORMAT_LITTLEFS_IF_FAILED true

int mydata;

void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
    file.close();
}

void appendFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Appending to file: %s\r\n", path);
```

```

File file = fs.open(path, FILE_APPEND);
if(!file){
    Serial.println("- failed to open file for appending");
    return;
}
if(file.print(message)){
    Serial.println("- message appended");
} else {
    Serial.println("- append failed");
}
file.close();
}

void readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return;
    }

    Serial.println("- read from file:");
    while(file.available()){
        Serial.write(file.read());
    }
    file.close();
}

void setup() {
    Serial.begin(115200);
    if(!LittleFS.begin(FORMAT_LITTLEFS_IF_FAILED)){
        Serial.println("LittleFS Mount Failed");
        return;
    }
    else{
        Serial.println("Little FS Mounted Successfully");
    }
    writeFile(LittleFS, "/data.txt", "MY ESP32 DATA \r\n");
}

void loop() {
    mydata = random (0, 1000);
    // Append data to the file
    appendFile(LittleFS, "/data.txt", (String(mydata)+"\r\n").c_str());
    readFile(LittleFS, "/data.txt"); // Read the contents of the file
    delay(30000);
}

```

How Does the Code Work?

We start by creating a variable that will hold a random number called `mydata`.

```
int mydata;
```

For this particular example, we just need to use the `writeFile()`, `appendFile()`, and `readFile()` functions. So, we have those functions defined before the `setup()`:

```
void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
    file.close();
}

void appendFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Appending to file: %s\r\n", path);

    File file = fs.open(path, FILE_APPEND);
    if(!file){
        Serial.println("- failed to open file for appending");
        return;
    }
    if(file.print(message)){
        Serial.println("- message appended");
    } else {
        Serial.println("- append failed");
    }
    file.close();
}

void readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return;
    }

    Serial.println("- read from file:");
    while(file.available()){
        Serial.write(file.read());
    }
    file.close();
}
```

In the `setup()`, we initialize the Serial Monitor for debugging purposes.

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

And we initialize the LittleFS filesystem:

```
if(!LittleFS.begin(FORMAT_LITTLEFS_IF_FAILED)){
    Serial.println("LittleFS Mount Failed");
    return;
}
else{
    Serial.println("Little FS Mounted Successfully");
}
```

Then, we create a file called **data.txt** with the following text inside MY ESP32 DATA:

```
writeFile(LittleFS, "/data.txt", "MY ESP32 DATA \r\n");
```

Something important to notice about the `writeFile()` function: it creates a file (if it doesn't exist) called **data.txt** with the text we define inside.

If that file already exists, the `writeFile()` function will overwrite any existing contents inside that file. So, if you want to continuously add new data without replacing it, you should use the `appendFile()` function after creating the file. If you want to replace the content of the file, you should use the `writeFile()`.

In the `loop()`, we start by attributing a random value between 0 and 1000 to the `mydata` variable.

```
mydata = random (0, 1000);
```

Then, we append data to the file by calling the `appendFile()` function.

```
// Append data to the file
appendFile(LittleFS, "/data.txt", (String(mydata)+ "\r\n").c_str());
```

Notice that we concatenate the `mydata` variable with "`\r\n`" so that subsequent data is written on the next line. Because our variable is of `int` type, we need to convert it to a `String` before concatenating.

```
String(mydata)
```

Additionally, then, we need to convert it to a `const char` using the `c_str()` method:

```
(String(mydata)+ "\r\n").c_str())
```

After appending data to the file, we'll read its content by calling the `readFile()` function.

```
readFile(LittleFS, "/data.txt"); // Read the contents of the file
```

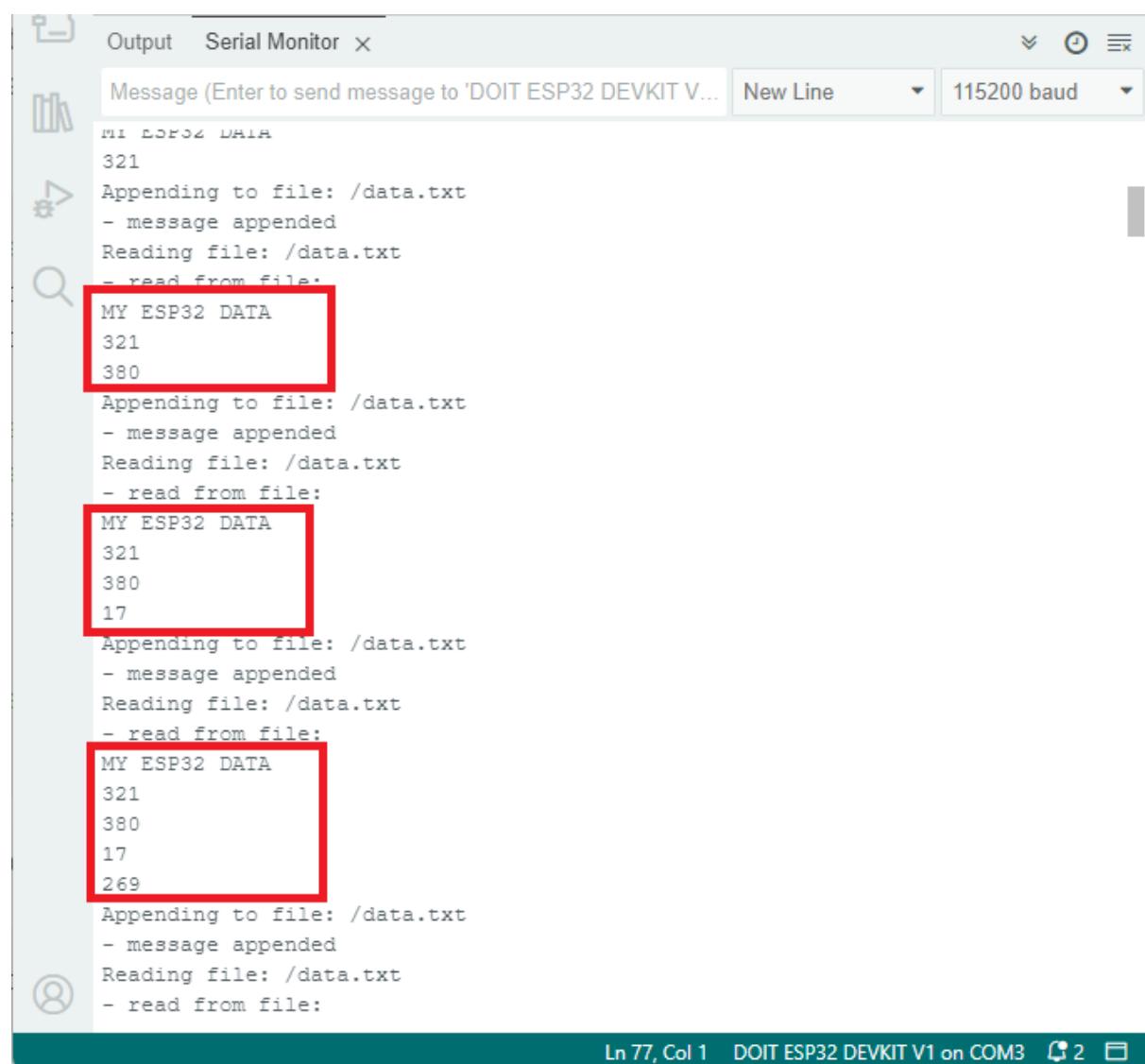
New random values are generated and added to the file every 30 seconds.

```
delay(30000);
```

Demonstration

Upload the code to your ESP32 board. Open the Serial Monitor at a baud rate of 115200.

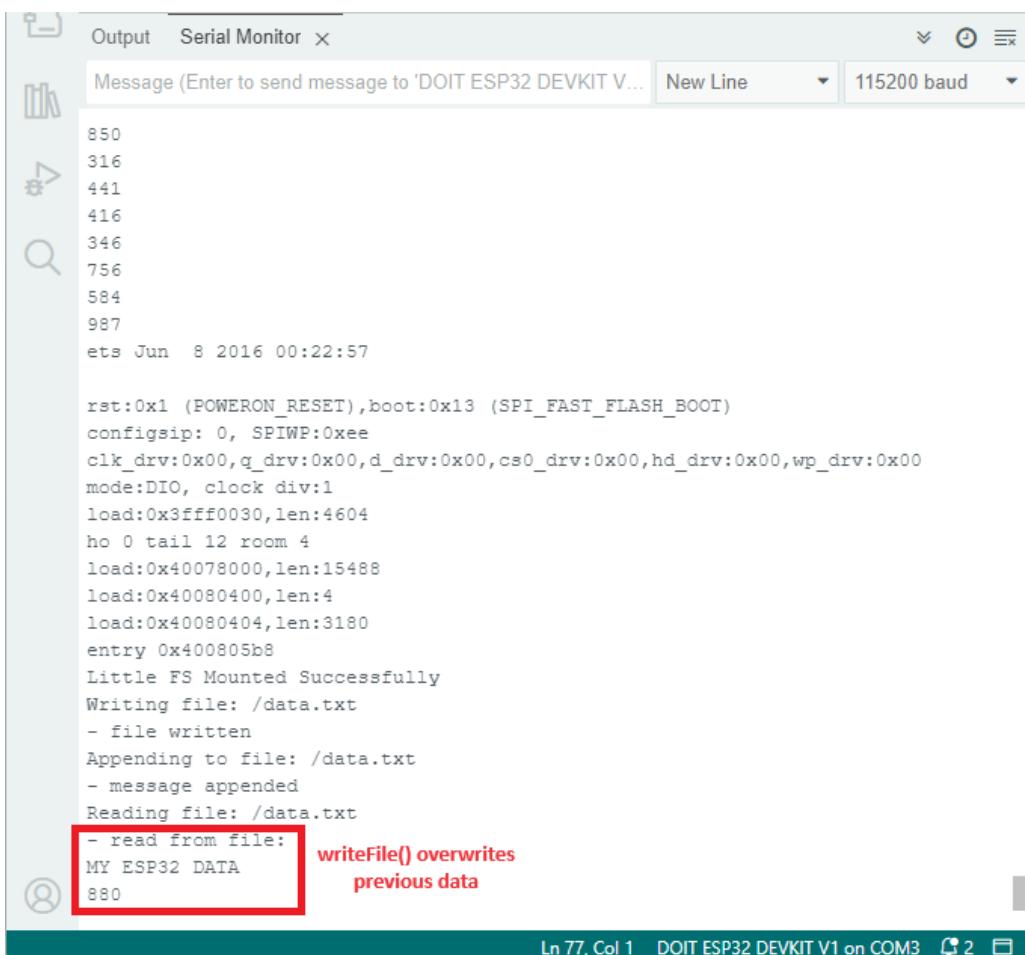
It should initialize the filesystem, create the file and start adding a new random number to the file every 30 seconds.



```
Output Serial Monitor x
Message (Enter to send message to 'DOIT ESP32 DEVKIT V...' New Line 115200 baud
mi ESP32 DATA
321
Appending to file: /data.txt
- message appended
Reading file: /data.txt
- read from file:
MY ESP32 DATA
321
380
Appending to file: /data.txt
- message appended
Reading file: /data.txt
- read from file:
MY ESP32 DATA
321
380
17
Appending to file: /data.txt
- message appended
Reading file: /data.txt
- read from file:
MY ESP32 DATA
321
380
17
269
Appending to file: /data.txt
- message appended
Reading file: /data.txt
- read from file:
```

Ln 77, Col 1 DOIT ESP32 DEVKIT V1 on COM3 2 8

Notice that if you restart your board, you'll lose all your previous data. Why is that happening?



```
850
316
441
416
346
756
584
987
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Little FS Mounted Successfully
Writing file: /data.txt
- file written
Appending to file: /data.txt
- message appended
Reading file: /data.txt
- read from file:
MY ESP32 DATA
880
```

Ln 77, Col 1 DOIT ESP32 DEVKIT V1 on COM3 2

That happens because we're calling the `writeFile()` function in the `setup()`. As we've explained previously, it will create a new file if it doesn't exist, or overwrite an already existing file with the same name. To prevent that, we can add some lines to the `setup()` to check whether the file already exists.

Checking if a file already exists

To check if a file already exists in the filesystem, we can use the `exists()` method and pass as an argument the file path. You can add the following lines to the `setup()` to prevent overwriting when the ESP32 restarts:

```
bool fileexists = LittleFS.exists("/data.txt");
Serial.print(fileexists);
if(!fileexists) {
    Serial.println("File doesn't exist");
    Serial.println("Creating file...");
    // Create File and add header
    writeFile(LittleFS, "/data.txt", "MY ESP32 DATA \r\n");
```

```
    }
} else {
    Serial.println("File already exists");
}
```

It uses the `exists()` method to check if the file already exists:

```
bool fileexists = LittleFS.exists("/data.txt");
```

It will return `true` if the file already exists or `false` if it doesn't.

If it doesn't exist, it will create the file with the content we define.

```
if(!fileexists) {
    Serial.println("File doesn't exist");
    Serial.println("Creating file...");
    // Create File and add header
    writeFile(LittleFS, "/data.txt", "MY ESP32 DATA \r\n");
}
```

If it already exists, it simply writes File already exists in the Serial Monitor.

```
else {
    Serial.println("File already exists");
}
```

Here's the complete example that checks if the file already exists.

If you test this example, you'll see that the file keeps all data even after a restart.

- [Click here to download the code.](#)

The screenshot shows the Serial Monitor window of the Arduino IDE. The text output is as follows:

```
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Little FS Mounted Successfully
1File already exists
Appending to file: /data.txt
- message appended
Reading file: /data.txt
- read from file:
MY ESP32 DATA
880
369
992
811
712
293
510
48
```

A red box highlights the section where the file is appended:

RESTART
Doesn't overwrite
previous data

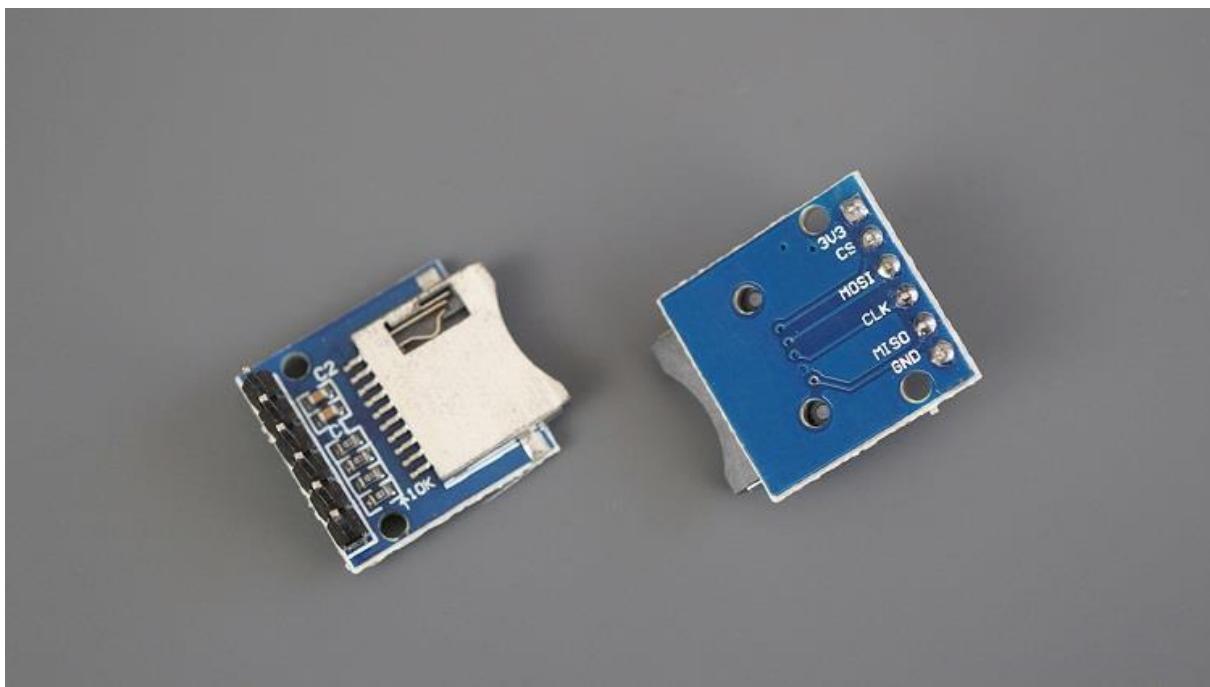
At the bottom of the window, status information is displayed: Ln 76, Col 29 DOIT ESP32 DEVKIT V1 on COM3 2 48

3.4 - Saving Data to a microSD Card

Instead of creating files on the ESP32 filesystem to save data, you can use a microSD card. The process to save data on files on the microSD card is similar, but you'll need to add the steps to initialize the microSD card. Using a microSD card with the ESP32 is especially useful for data logging or storing files that don't fit in the LittleFS filesystem.

MicroSD Card Module

To interface the microSD card with the ESP32 board, we'll use a microSD card module. There are different microSD card modules compatible with the ESP32. We're using the microSD card module shown in the following figure—it communicates using SPI communication protocol. You can use any other microSD card module with an SPI interface.



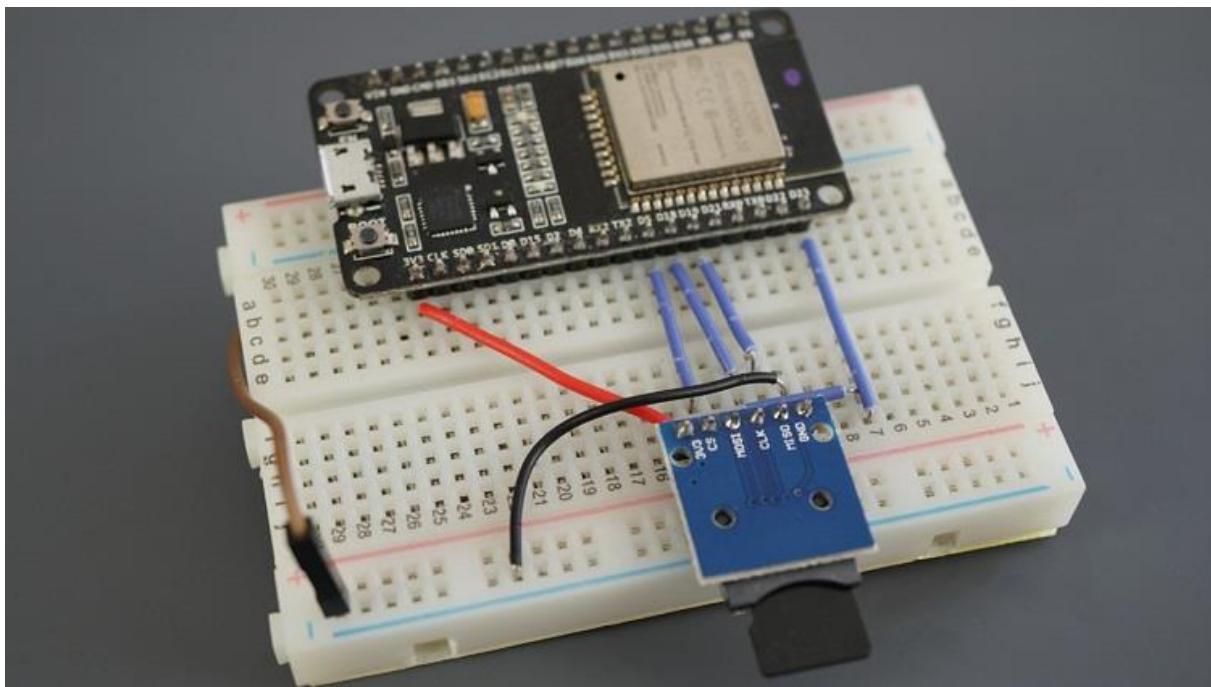
MicroSD Card Module Pinout – SPI

The microSD card module communicates using SPI communication protocol. You can connect it to the ESP32 using the default SPI pins. These are the pins if your board uses the ESP-WROOM-32 chip:

- 3V3 → 3V3*
- CS → GPIO 5
- MOSI → GPIO 23
- CLK → GPIO 18
- MISO → GPIO 19
- GND → GND

* Some microSD card modules require 5V instead of 3V3 to operate.

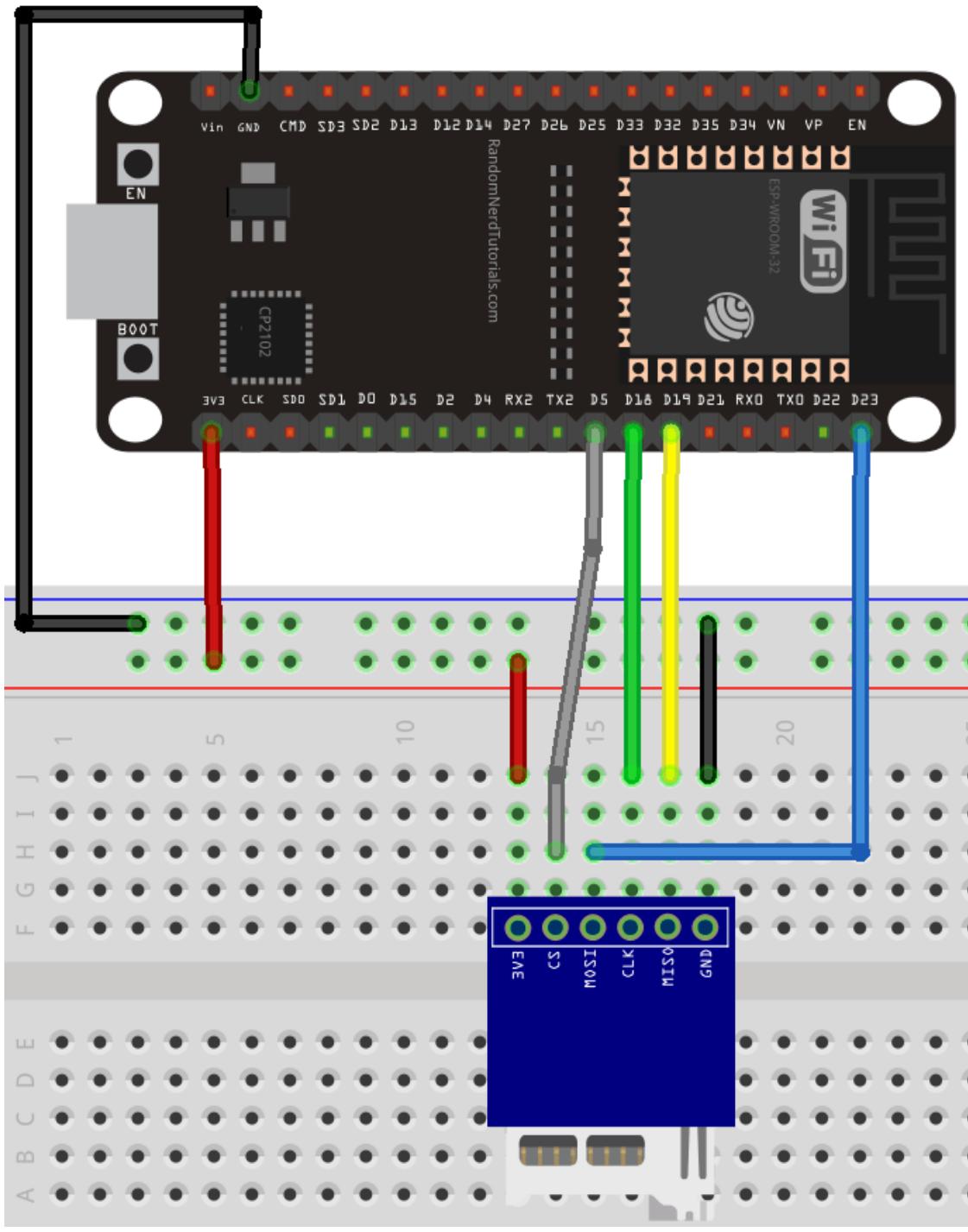
Wiring the Circuit



For this tutorial, you need the following parts:

- [ESP32 development board](#) (read: [Best ESP32 development boards](#))
- [MicroSD Card Module](#)
- [MicroSD Card](#)
- [Jumper Wires](#)
- [Breadboard](#)

To wire the microSD card module to the ESP32 board, you can follow the next diagram (for the default ESP32 SPI pins):



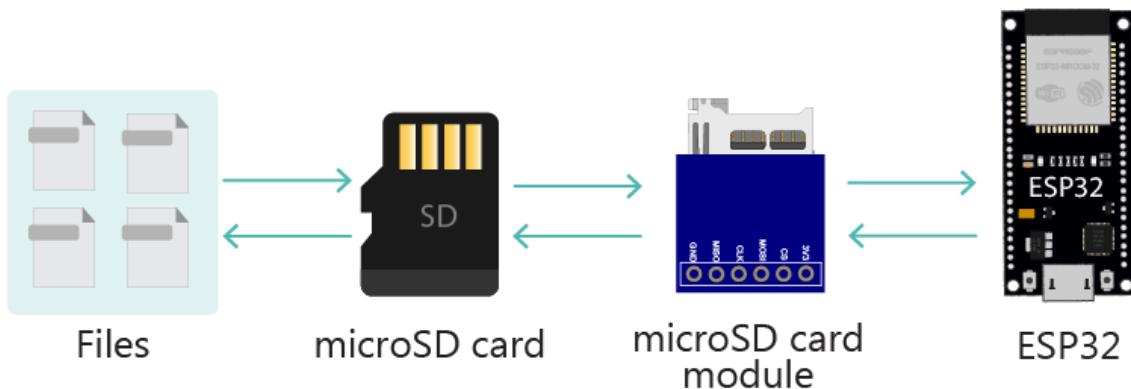
Preparing the microSD Card

Before proceeding with the tutorial, make sure you format your microSD card as FAT32. You can use a software tool like [SD Card Formatter](#) (compatible with Windows and Mac OS).

Handling Files with a MicroSD Card Module

There are two different libraries for the ESP32 (included in the Arduino core for the ESP32): the [SD library](#) and the [SDD_MMC.h library](#).

If you use the SD library, you're using the SPI controller. If you use the SDD_MMC library you're using the ESP32 SD/SDIO/MMC controller. We'll not cover the SD/SDIO/MMC controller here, but you can learn more about the [ESP32 SD/SDIO/MMC driver here](#).



There are several examples in Arduino IDE that show how to handle files on the microSD card using the ESP32. In the Arduino IDE, go to **File > Examples > SD(esp32) > SD_Test**, or copy the following code.

- [Click here to download the code.](#)

```
#include "FS.h"
#include "SD.h"
#include "SPI.h"

/*
Uncomment and set up if you want to use custom pins for the SPI communication
#define REASSIGN_PINS
int sck = -1;
int miso = -1;
int mosi = -1;
int cs = -1;
*/

void listDir(fs::FS &fs, const char *dirname, uint8_t levels) {
    Serial.printf("Listing directory: %s\n", dirname);

    File root = fs.open(dirname);
    if (!root) {
        Serial.println("Failed to open directory");
        return;
    }

    while (true) {
        File file = root.openNextFile();
        if (!file) break;

        Serial.print(" - ");
        Serial.print(file.name());
        Serial.print(" (");
        Serial.print(file.size());
        Serial.print(" bytes)");
        Serial.println((file.isDirectory()) ? " (dir)" : "(file)");

        if (levels)
            listDir(fs, file.path(), levels-1);
    }
}
```

```

if (!root.isDirectory()) {
    Serial.println("Not a directory");
    return;
}

File file = root.openNextFile();
while (file) {
    if (file.isDirectory()) {
        Serial.print(" DIR : ");
        Serial.println(file.name());
        if (levels) {
            listDir(fs, file.path(), levels - 1);
        }
    } else {
        Serial.print(" FILE: ");
        Serial.print(file.name());
        Serial.print(" SIZE: ");
        Serial.println(file.size());
    }
    file = root.openNextFile();
}
}

void createDir(fs::FS &fs, const char *path) {
    Serial.printf("Creating Dir: %s\n", path);
    if (fs.mkdir(path)) {
        Serial.println("Dir created");
    } else {
        Serial.println("mkdir failed");
    }
}

void removeDir(fs::FS &fs, const char *path) {
    Serial.printf("Removing Dir: %s\n", path);
    if (fs.rmdir(path)) {
        Serial.println("Dir removed");
    } else {
        Serial.println("rmdir failed");
    }
}

void readFile(fs::FS &fs, const char *path) {
    Serial.printf("Reading file: %s\n", path);

    File file = fs.open(path);
    if (!file) {
        Serial.println("Failed to open file for reading");
        return;
    }

    Serial.print("Read from file: ");
    while (file.available()) {
        Serial.write(file.read());
    }
    file.close();
}

void writeFile(fs::FS &fs, const char *path, const char *message) {
    Serial.printf("Writing file: %s\n", path);

    File file = fs.open(path, FILE_WRITE);

```

```

if (!file) {
    Serial.println("Failed to open file for writing");
    return;
}
if (file.print(message)) {
    Serial.println("File written");
} else {
    Serial.println("Write failed");
}
file.close();
}

void appendFile(fs::FS &fs, const char *path, const char *message) {
    Serial.printf("Appending to file: %s\n", path);

    File file = fs.open(path, FILE_APPEND);
    if (!file) {
        Serial.println("Failed to open file for appending");
        return;
    }
    if (file.print(message)) {
        Serial.println("Message appended");
    } else {
        Serial.println("Append failed");
    }
    file.close();
}

void renameFile(fs::FS &fs, const char *path1, const char *path2) {
    Serial.printf("Renaming file %s to %s\n", path1, path2);
    if (fs.rename(path1, path2)) {
        Serial.println("File renamed");
    } else {
        Serial.println("Rename failed");
    }
}

void deleteFile(fs::FS &fs, const char *path) {
    Serial.printf("Deleting file: %s\n", path);
    if (fs.remove(path)) {
        Serial.println("File deleted");
    } else {
        Serial.println("Delete failed");
    }
}

void testFileIO(fs::FS &fs, const char *path) {
    File file = fs.open(path);
    static uint8_t buf[512];
    size_t len = 0;
    uint32_t start = millis();
    uint32_t end = start;
    if (file) {
        len = file.size();
        size_t flen = len;
        start = millis();
        while (len) {
            size_t toRead = len;
            if (toRead > 512) {
                toRead = 512;
            }

```

```

        file.read(buf, toRead);
        len -= toRead;
    }
    end = millis() - start;
    Serial.printf("%u bytes read for %lu ms\n",flen, end);
    file.close();
} else {
    Serial.println("Failed to open file for reading");
}

file = fs.open(path, FILE_WRITE);
if (!file) {
    Serial.println("Failed to open file for writing");
    return;
}

size_t i;
start = millis();
for (i = 0; i < 2048; i++) {
    file.write(buf, 512);
}
end = millis() - start;
Serial.printf("%u bytes written for %lu ms\n", 2048 * 512, end);
file.close();
}

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

#ifndef REASSIGN_PINS
    SPI.begin(sck, miso, mosi, cs);
    if (!SD.begin(cs)) {
#else
    if (!SD.begin()) {
#endif
        Serial.println("Card Mount Failed");
        return;
    }
    uint8_t cardType = SD.cardType();

    if (cardType == CARD_NONE) {
        Serial.println("No SD card attached");
        return;
    }

    Serial.print("SD Card Type: ");
    if (cardType == CARD_MMC) {
        Serial.println("MMC");
    } else if (cardType == CARD_SD) {
        Serial.println("SDSC");
    } else if (cardType == CARD_SDHC) {
        Serial.println("SDHC");
    } else {
        Serial.println("UNKNOWN");
    }

    uint64_t cardSize = SD.cardSize() / (1024 * 1024);
    Serial.printf("SD Card Size: %lluMB\n", cardSize);
}

```

```

listDir(SD, "/");
createDir(SD, "/mydir");
listDir(SD, "/");
removeDir(SD, "/mydir");
listDir(SD, "/", 2);
writeFile(SD, "/hello.txt", "Hello ");
appendFile(SD, "/hello.txt", "World!\n");
readFile(SD, "/hello.txt");
deleteFile(SD, "/foo.txt");
renameFile(SD, "/hello.txt", "/foo.txt");
readFile(SD, "/foo.txt");
testFileIO(SD, "/test.txt");
Serial.printf("Total space: %lluMB\n", SD.totalBytes() / (1024 * 1024));
Serial.printf("Used space: %lluMB\n", SD.usedBytes() / (1024 * 1024));
}

void loop() {
}

```

This example shows how to do almost any task you may need with the microSD card. You'll see that it works quite similar to the LittleFS example, but we'll use the SD filesystem instead.

Alternatively, you can use the SD_MMC examples – these are similar to the SD examples, but use the SDMMC driver. For the SDMMC driver, you need a compatible microSD card module. The module we're using in this tutorial doesn't support SDMMC.

How Does the Code Work?

First, you need to include the following libraries: `FS.h` to handle files, `SD.h` to interface with the microSD card and `SPI.h` to use SPI communication protocol.

```
#include "FS.h"
#include "SD.h"
#include "SPI.h"
```

The example provides several functions to handle files on the microSD card.

Using Other SPI Pins

If you want to use other SPI pins instead of the default ones, uncomment the following lines and replace the `-1` with the GPIOs you're using.

```
/* Uncomment and set up if you want to use custom pins for the SPI communication
```

```
#define REASSIGN_PINS  
int sck = -1;  
int miso = -1;  
int mosi = -1;  
int cs = -1; /*
```

List a directory

The `listDir()` function lists the directories on the SD card. This function accepts as arguments the filesystem (SD), the main directory's name, and the levels to go into the directory.

```
void listDir(fs::FS &fs, const char *dirname, uint8_t levels) {  
    Serial.printf("Listing directory: %s\n", dirname);  
  
    File root = fs.open(dirname);  
    if (!root) {  
        Serial.println("Failed to open directory");  
        return;  
    }  
    if (!root.isDirectory()) {  
        Serial.println("Not a directory");  
        return;  
    }  
  
    File file = root.openNextFile();  
    while (file) {  
        if (file.isDirectory()) {  
            Serial.print(" DIR : ");  
            Serial.println(file.name());  
            if (levels) {  
                listDir(fs, file.path(), levels - 1);  
            }  
        } else {  
            Serial.print(" FILE: ");  
            Serial.print(file.name());  
            Serial.print(" SIZE: ");  
            Serial.println(file.size());  
        }  
        file = root.openNextFile();  
    }  
}
```

Here's an example of how to call this function. The `/` corresponds to the microSD card root directory.

```
listDir(SD, "/", 0);
```

Create a Directory

The `createDir()` function creates a new directory. Pass as an argument the SD filesystem and the directory name path.

```
void createDir(fs::FS &fs, const char *path) {
```

```
Serial.printf("Creating Dir: %s\n", path);
if (fs.mkdir(path)) {
    Serial.println("Dir created");
} else {
    Serial.println("mkdir failed");
}
```

For example, the following command creates a new directory on the root called `mydir`.

```
createDir(SD, "/mydir");
```

Remove a Directory

To remove a directory from the microSD card, use the `removeDir()` function and pass as an argument the `SD` filesystem and the directory name `path`.

```
void removeDir(fs::FS &fs, const char *path) {
    Serial.printf("Removing Dir: %s\n", path);
    if (fs.rmdir(path)) {
        Serial.println("Dir removed");
    } else {
        Serial.println("rmdir failed");
    }
}
```

Here is an example:

```
removeDir(SD, "/mydir");
```

Read File Content

The `readFile()` function reads the content of a file and prints the content in the Serial Monitor. As with previous functions, pass as an argument the `SD` filesystem and the file path.

```
void readFile(fs::FS &fs, const char *path) {
    Serial.printf("Reading file: %s\n", path);

    File file = fs.open(path);
    if (!file) {
        Serial.println("Failed to open file for reading");
        return;
    }
    Serial.print("Read from file: ");
    while (file.available()) {
        Serial.write(file.read());
    }
    file.close();
}
```

For example, the following line reads the content of the `hello.txt` file.

```
readFile(SD, "/hello.txt");
```

Write Content to a File

To write content to a file, you can use the `writeFile()` function. Pass as an argument, the `SD` filesystem, the file path and the message.

```
void writeFile(fs::FS &fs, const char *path, const char *message) {
    Serial.printf("Writing file: %s\n", path);

    File file = fs.open(path, FILE_WRITE);
    if (!file) {
        Serial.println("Failed to open file for writing");
        return;
    }
    if (file.print(message)) {
        Serial.println("File written");
    } else {
        Serial.println("Write failed");
    }
    file.close();
}
```

The following line writes Hello in the `hello.txt` file.

```
writeFile(SD, "/hello.txt", "Hello ");
```

Append Content to a File

Similarly, you can append content to a file (without overwriting previous content) using the `appendFile()` function.

```
void appendFile(fs::FS &fs, const char *path, const char *message) {
    Serial.printf("Appending to file: %s\n", path);

    File file = fs.open(path, FILE_APPEND);
    if (!file) {
        Serial.println("Failed to open file for appending");
        return;
    }
    if (file.print(message)) {
        Serial.println("Message appended");
    } else {
        Serial.println("Append failed");
    }
    file.close();
}
```

The following line appends the message `World!\n` in the `hello.txt` file. The `\n` means that the next time you write something to the file, it will be written in a new line.

```
appendFile(SD, "/hello.txt", "World!\n");
```

Rename a File

You can rename a file using the `renameFile()` function. Pass as arguments the SD filesystem, the original filename, and the new filename.

```
void renameFile(fs::FS &fs, const char *path1, const char *path2) {
    Serial.printf("Renaming file %s to %s\n", path1, path2);
    if (fs.rename(path1, path2)) {
        Serial.println("File renamed");
    } else {
        Serial.println("Rename failed");
    }
}
```

The following line renames the `hello.txt` file to `foo.txt`.

```
renameFile(SD, "/hello.txt", "/foo.txt");
```

Delete a File

Use the `deleteFile()` function to delete a file. Pass as an argument the SD filesystem and the path of the file you want to delete.

```
void deleteFile(fs::FS &fs, const char *path) {
    Serial.printf("Deleting file: %s\n", path);
    if (fs.remove(path)) {
        Serial.println("File deleted");
    } else {
        Serial.println("Delete failed");
    }
}
```

The following line deletes the `foo.txt` file from the microSD card.

```
deleteFile(SD, "/foo.txt");
```

Test a File

The `testFileIO()` function shows how long it takes to read the content of a file.

```
void testFileIO(fs::FS &fs, const char *path) {
    File file = fs.open(path);
    static uint8_t buf[512];
    size_t len = 0;
    (...)
```

```
size_t flen = len;
start = millis();
while (len) {
    size_t toRead = len;
end = millis() - start;
Serial.printf("%u bytes written for %lu ms\n", 2048 * 512, end);
file.close();
}
```

The following function tests the **test.txt** file.

```
testFileIO(SD, "/test.txt");
```

Initialize the microSD Card

In the `setup()`, the following lines initialize the microSD card on the default SPI pins with `SD.begin()`.

```
if (!SD.begin()) {
```

If you want to use custom pins, you need to initialize SPI communication on your desired pins and then initialize the microSD card using the `begin()` method and pass the CS pin.

```
SPI.begin(sck, miso, mosi, cs);
if (!SD.begin(cs)) {
```

Get microSD Card Type

The following lines print the microSD card type on the Serial Monitor.

```
Serial.print("SD Card Type: ");
if (cardType == CARD_MMC) {
    Serial.println("MMC");
} else if (cardType == CARD_SD) {
    Serial.println("SDSC");
} else if (cardType == CARD_SDHC) {
    Serial.println("SDHC");
} else {
    Serial.println("UNKNOWN");
}
```

Get microSD Card Size

You can get the microSD card size by calling the `cardSize()` method:

```
uint64_t cardSize = SD.cardSize() / (1024 * 1024);
Serial.printf("SD Card Size: %lluMB\n", cardSize);
```

Testing MicroSD Card Functions

The following lines call the functions we've seen previously.

```
listDir(SD, "/");
createDir(SD, "/mydir");
listDir(SD, "/");
removeDir(SD, "/mydir");
listDir(SD, "/");
writeFile(SD, "/hello.txt", "Hello ");
appendFile(SD, "/hello.txt", "World!\n");
readFile(SD, "/hello.txt");
deleteFile(SD, "/foo.txt");
renameFile(SD, "/hello.txt", "/foo.txt");
readFile(SD, "/foo.txt");
testFileIO(SD, "/test.txt");
Serial.printf("Total space: %lluMB\n", SD.totalBytes() / (1024 * 1024));
Serial.printf("Used space: %lluMB\n", SD.usedBytes() / (1024 * 1024));
```

As you can see most functions are similar to writing to the LittleFS filesystem.

Demonstration

Upload the previous sketch to your ESP32 board. After that, open the Serial Monitor and press the ESP32 on-board RST button. If the initialization succeeds, you'll get similar messages on the Serial Monitor.



The screenshot shows a Serial Monitor window with a light gray background. On the left side, there are three icons: a blue folder icon, a green arrow icon pointing right, and a magnifying glass icon. The main area displays the output of a C program. The output shows the following sequence of events:

- SD Card Type: SDHC
- SD Card Size: 7580MB
- Listing directory: /
- DIR : System Volume Information
- FILE: test.txt SIZE: 1048576
- FILE: foo.txt SIZE: 13
- Creating Dir: /mydir
- Dir created
- Listing directory: /
- DIR : System Volume Information
- FILE: test.txt SIZE: 1048576
- FILE: foo.txt SIZE: 13
- DIR : mydir
- Removing Dir: /mydir
- Dir removed
- Listing directory: /
- DIR : System Volume Information
- Listing directory: /System Volume Information
- FILE: WPSettings.dat SIZE: 12
- FILE: IndexerVolumeGuid SIZE: 76
- FILE: test.txt SIZE: 1048576
- FILE: foo.txt SIZE: 13

```
DIR : mydir
Removing Dir: /mydir
Dir removed
Listing directory: /
  DIR : System Volume Information
  Listing directory: /System Volume Information
    FILE: WPSettings.dat  SIZE: 12
    FILE: IndexerVolumeGuid  SIZE: 76
    FILE: test.txt  SIZE: 1048576
    FILE: foo.txt  SIZE: 13
Writing file: /hello.txt
File written
Appending to file: /hello.txt
Message appended
Reading file: /hello.txt
Read from file: Hello World!
Deleting file: /foo.txt
File deleted
Renaming file /hello.txt to /foo.txt
File renamed
Reading file: /foo.txt
Read from file: Hello World!
1048576 bytes read for 2381 ms
1048576 bytes written for 2483 ms
Total space: 7572MB
Used space: 1MB
```



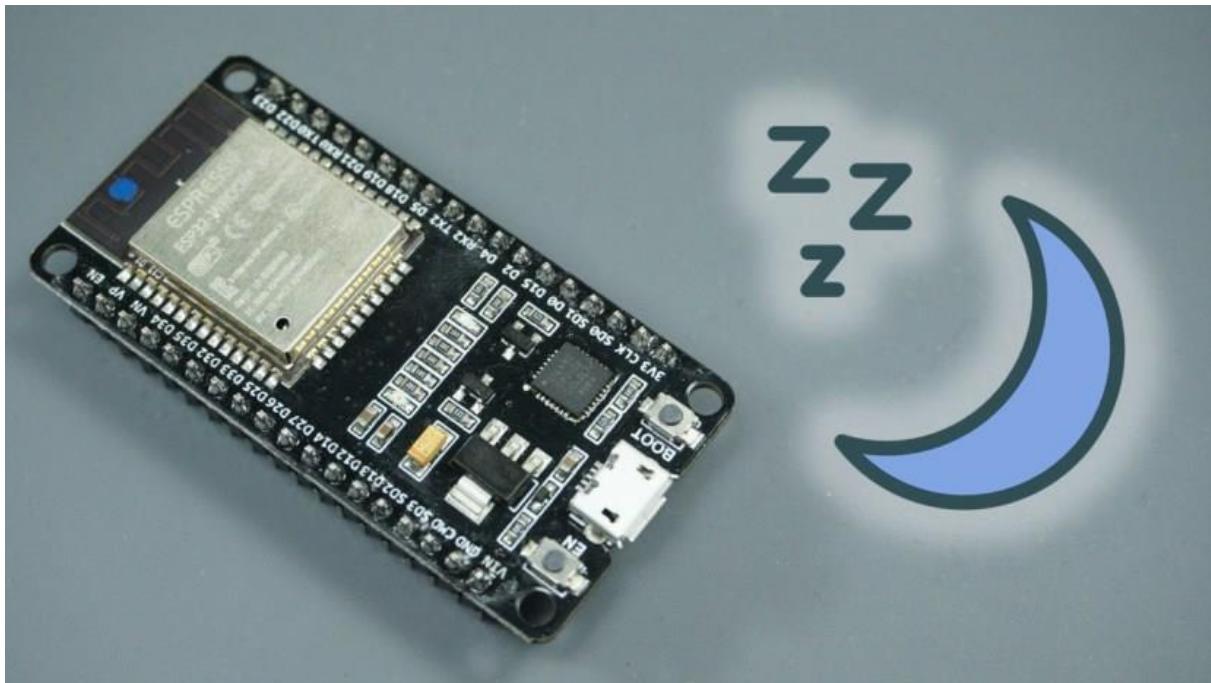
Ln 255, Col 44 DOIT ESP32 DEVKIT V1 on COM3 6 2

MODULE 4

ESP32 Deep Sleep

4.1 - ESP32 Deep Sleep Mode

In this Module, you're going to learn what deep sleep is, how to put the ESP32 into deep sleep mode, and different methods you can use to wake up your ESP32.



Note: this Unit is an introduction to the deep sleep mode. For ESP32 deep sleep examples, go to the subsequent units in this Module.

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 in 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

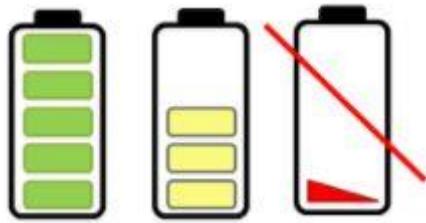
And here's also Table 10 to compare the power consumption in active mode:

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 is not ideal since batteries' power 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 leaving just enough activity to wake up the processor when something interesting happens.

In deep sleep mode, neither CPU nor 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.

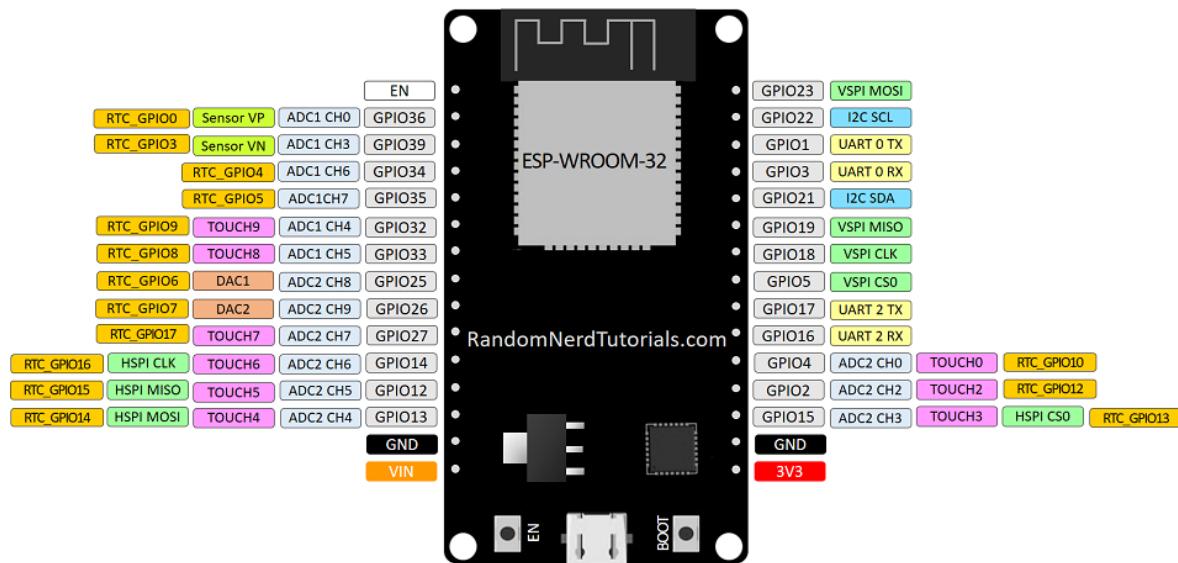
RTC_GPIO Pins

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](#) on page 14.

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.

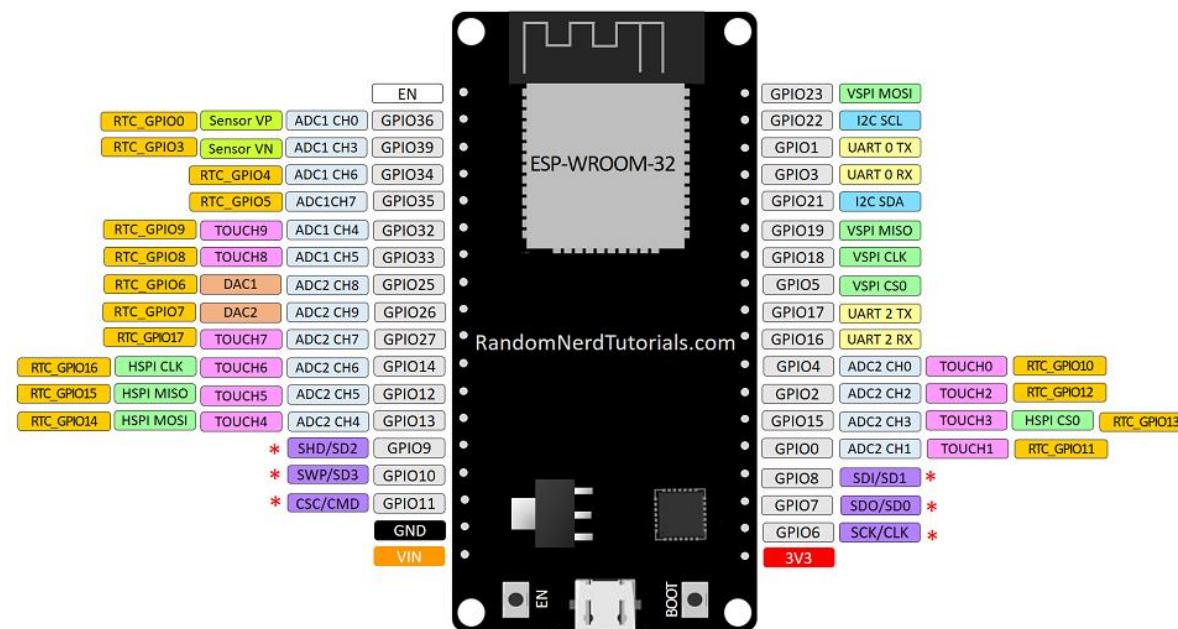
ESP32 DEVKIT V1 – DOIT

version with 30 GPIOs



ESP32 DEVKIT V1 – DOIT

version with 36 GPIOs



* Pins SCK/CLK, SDO/SD0, SDI/SD1, SHD/SD2, SWP/SD3 and CSC/CMD, namely, GPIO6 to GPIO11 are connected to the integrated SPI flash integrated on ESP-WROOM-32 and are not recommended for other uses.

Wake-Up Sources

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

- You can use the **timer**, waking up your ESP32 at predefined periods of time;
- You can use two possibilities of **external wake-up**: you can use either one external wake-up or several different external wake-up sources;
- You can use the **touch pins**;
- You can use the **ULP co-processor** to wake up.

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 configuring 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.

Note: the following units in this module will show you how to use the different wake-up sources.

Additional Resources

Alongside this deep sleep module, we recommend taking a look at ESP32 Espressif datasheet, and at the deep sleep API documentation at readthedocs.io website:

- [ESP32 Espressif Datasheet \(EN\)](#)
- [ESP32 Deep Sleep API Documentation](#)

4.2 - Deep Sleep – Timer Wake Up

Your ESP32 can go into deep sleep mode and then wake up at predefined periods of time. This feature is handy 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. In this Unit, we're going to show you how you can do that using the Arduino IDE.



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. Or you can click the link below to get the code.

- [Click here to download the code.](#)

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

RTC_DATA_ATTR int bootCount = 0;

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

```

wakeup_reason = esp_sleep_get_wakeup_cause();

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

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

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

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

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

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

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

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

```

Let's take a look at this code.

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

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 source that caused the wake-up from deep sleep.

```
void print_wakeup_reason() {
    esp_sleep_wakeup_cause_t wakeup_reason;

    wakeup_reason = esp_sleep_get_wakeup_cause();

    switch (wakeup_reason) {
        case ESP_SLEEP_WAKEUP_EXT0:
            Serial.println("Wakeup caused by external signal using RTC_IO");
            break;
        case ESP_SLEEP_WAKEUP_EXT1:
            Serial.println("Wakeup caused by external signal using RTC_CNTL");
            break;
    }
}
```

```
case ESP_SLEEP_WAKEUP_TIMER:  
    Serial.println("Wakeup caused by timer");  
    break;  
case ESP_SLEEP_WAKEUP_TOUCHPAD:  
    Serial.println("Wakeup caused by touchpad");  
    break;  
case ESP_SLEEP_WAKEUP_ULP:  
    Serial.println("Wakeup caused by ULP program");  
    break;  
default:  
    Serial.printf("Wakeup was not caused by deep sleep: %d\n", wakeup_reason);  
    break;  
}  
}
```

This function will be called later in the `setup()` every time the ESP32 reboots or wakes up from sleep. This way, we can check what caused the wake-up.

setup()

In the `setup()` is where you should put your code. You need to write all the instructions before calling the `esp_deep_sleep_start()` function.

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.

```
//Increment boot number and print it every reboot  
++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 an 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 ESP32 goes to sleep by calling the following function:

```
esp_deep_sleep_start();
```

As soon as you call the `esp_deep_sleep_start()` function, the ESP32 will go to sleep and will not run any code written after this function. When it wakes up from deep sleep, it will run the code from the beginning.

loop()

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

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

Testing the Example

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

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.



```
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Boot number: 109
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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Boot number: 110
Wakeup caused by timer
Setup ESP32 to sleep for every 5 Seconds
Going to sleep now
```

Ln 87, Col 1 DOIT ESP32 DEVKIT V1 on COM3

However, notice that if you press the RESET/EN button on the ESP32 board, it resets the `bootCount` to 1 again. That's because we're saving the `bootCount` variable on the RTC memory that is erased if you click on the RESET/EN button.

Wrapping Up

In summary, we've shown you how to use the timer wake-up source to wake-up the ESP32 in this unit.

- 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.

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 for performing periodic tasks with the ESP32, like daily tasks, without draining much power.

4.3 - Deep Sleep with Touch Wake Up

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



Enable Touch Wake Up

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

```
touchSleepWakeUpEnable(TOUCH_PIN, THRESHOLD);
```

Code

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

- [Click here to download the code.](#)

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

RTC_DATA_ATTR int bootCount = 0;
touch_pad_t touchPin;

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

    wakeup_reason = esp_sleep_get_wakeup_cause();

    switch (wakeup_reason) {
        case ESP_SLEEP_WAKEUP_EXT0:
            Serial.println("Wakeup caused by external signal using RTC_IO");
            break;
        case ESP_SLEEP_WAKEUP_EXT1:
            Serial.println("Wakeup caused by external signal using RTC_CNTL");
            break;
    }
}
```

```

        case ESP_SLEEP_WAKEUP_TIMER:
            Serial.println("Wakeup caused by timer");
            break;
        case ESP_SLEEP_WAKEUP_TOUCHPAD:
            Serial.println("Wakeup caused by touchpad");
            break;
        case ESP_SLEEP_WAKEUP_ULP:
            Serial.println("Wakeup caused by ULP program");
            break;
        default:
            Serial.printf("Wakeup was not caused by deep sleep: %d\n", wakeup_reason);
            break;
    }
}

// Method to print the touchpad by which ESP32 has been awoken from sleep
void print_wakeup_touchpad() {
    touchPin = esp_sleep_get_touchpad_wakeup_status();

#if CONFIG_IDF_TARGET_ESP32
    switch (touchPin) {
        case 0: Serial.println("Touch detected on GPIO 4"); break;
        case 1: Serial.println("Touch detected on GPIO 0"); break;
        case 2: Serial.println("Touch detected on GPIO 2"); break;
        case 3: Serial.println("Touch detected on GPIO 15"); break;
        case 4: Serial.println("Touch detected on GPIO 13"); break;
        case 5: Serial.println("Touch detected on GPIO 12"); break;
        case 6: Serial.println("Touch detected on GPIO 14"); break;
        case 7: Serial.println("Touch detected on GPIO 27"); break;
        case 8: Serial.println("Touch detected on GPIO 33"); break;
        case 9: Serial.println("Touch detected on GPIO 32"); break;
        default: Serial.println("Wakeup not by touchpad"); break;
    }
#else
    if (touchPin < TOUCH_PAD_MAX) {
        Serial.printf("Touch detected on GPIO %d\n", touchPin);
    } else {
        Serial.println("Wakeup not by touchpad");
    }
#endif
}

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

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

    // Print the wakeup reason for ESP32 and touchpad too
    print_wakeup_reason();
    print_wakeup_touchpad();

#if CONFIG_IDF_TARGET_ESP32
    // Setup sleep wakeup on Touch Pad 3 + 7 (GPIO15 + GPIO 27)
    touchSleepWakeUpEnable(T3, THRESHOLD);
    touchSleepWakeUpEnable(T7, THRESHOLD);

```

```

#else // ESP32-S2 + ESP32-S3
    // Setup sleep wakeup on Touch Pad 3 (GPIO3)

```

```

touchSleepWakeUpEnable(T3, THRESHOLD);

#endif

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

void loop() {
    // This will never be reached
}

```

Setting the Threshold

Note: we recommend reading the [“ESP32 Touch Sensor” Unit in “Exploring the ESP32 GPIOs” Module](#), if you haven’t already, to learn more about the touch pins.

The first thing you need to do is set a threshold value for the touch pins.

```
#define THRESHOLD 40 // Greater the value, more the sensitivity
```

The values read by a touch pin decrease when you touch it. The threshold value means that the ESP32 will wake up when the value read on the touch pin is below 40. You can adjust that value depending on the desired sensitivity.

However, if you’re using an ESP32-S2 or ESP32-S3 models, things work a little differently. That’s why there’s a different section in the code defining a different threshold for those boards. In this case, the lower the value, the more the sensitivity.

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

Setting Touch Pins as a Wake-up Source

To set a touch pin as a wake-up source, you can use the `touchSleepWakeUpEnable()` function that accepts as arguments the touch pin and the threshold value that will wake up the board.

In this example, it sets GPIO 15 (T3) and GPIO 27 (T7) as wake-up sources with the same threshold value.

```
#if CONFIG_IDF_TARGET_ESP32
    // Setup sleep wakeup on Touch Pad 3 + 7 (GPIO15 + GPIO 27)
    touchSleepWakeUpEnable(T3, THRESHOLD);
    touchSleepWakeUpEnable(T7, THRESHOLD);
```

If you're using an ESP32-S2 or S3 model, the example just defines wake-up on GPIO 3 (T3). Note that the touch pin numbering might be different depending on the board model you're using.

```
#else // ESP32-S2 + ESP32-S3
    // Setup sleep wakeup on Touch Pad 3 (GPIO3)
    touchSleepWakeUpEnable(T3, THRESHOLD);
```

Finally, call the `esp_deep_sleep_start()` to put the ESP32 in deep sleep mode.

```
esp_deep_sleep_start();
```

These were the general instructions to set up touch pins as a wake-up source. Now, let's take a look at the other sections of the code.

setup()

When the ESP32 wakes up from sleep, it will run the code from the start until it finds the `esp_deep_sleep_start()` function.

In this particular example, we have a control variable called `bootCount` that will be increased between each sleep cycle so that we have an idea of how many times the ESP32 woke up.

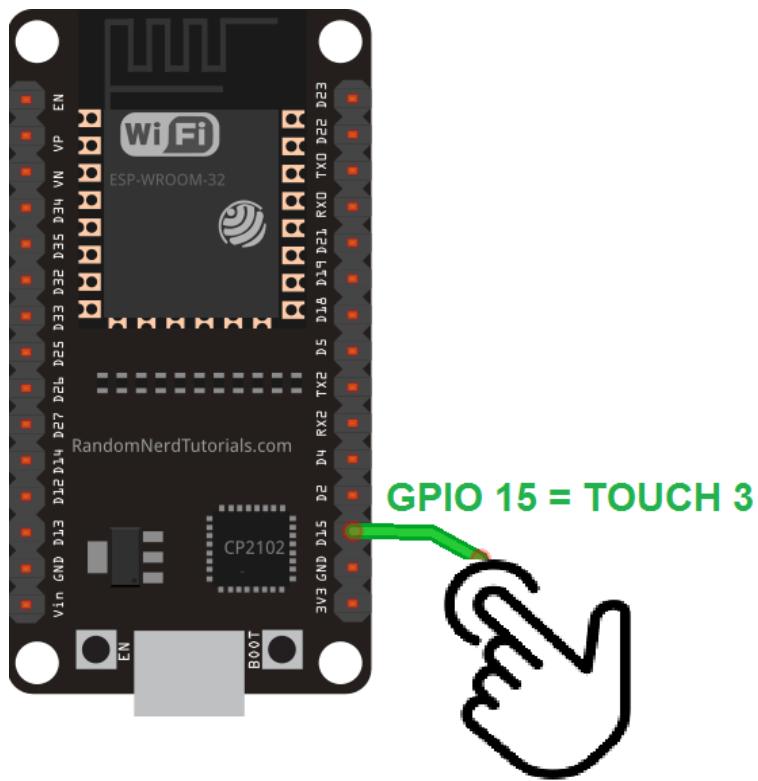
```
// Increment boot number and print it every reboot
++bootCount;
Serial.println("Boot number: " + String(bootCount));
```

We also call the `print_wakeup_reason()` and `print_wakeup_touchpad()` functions defined earlier in the code to print the wake-up reason and the pin that caused the wake-up.

```
// Print the wakeup reason for ESP32 and touchpad too
print_wakeup_reason();
print_wakeup_touchpad();
```

Wiring the Circuit

To test this example, wire a jumper wire to GPIO 15 and/or GPIO 27, as shown in the schematic below.



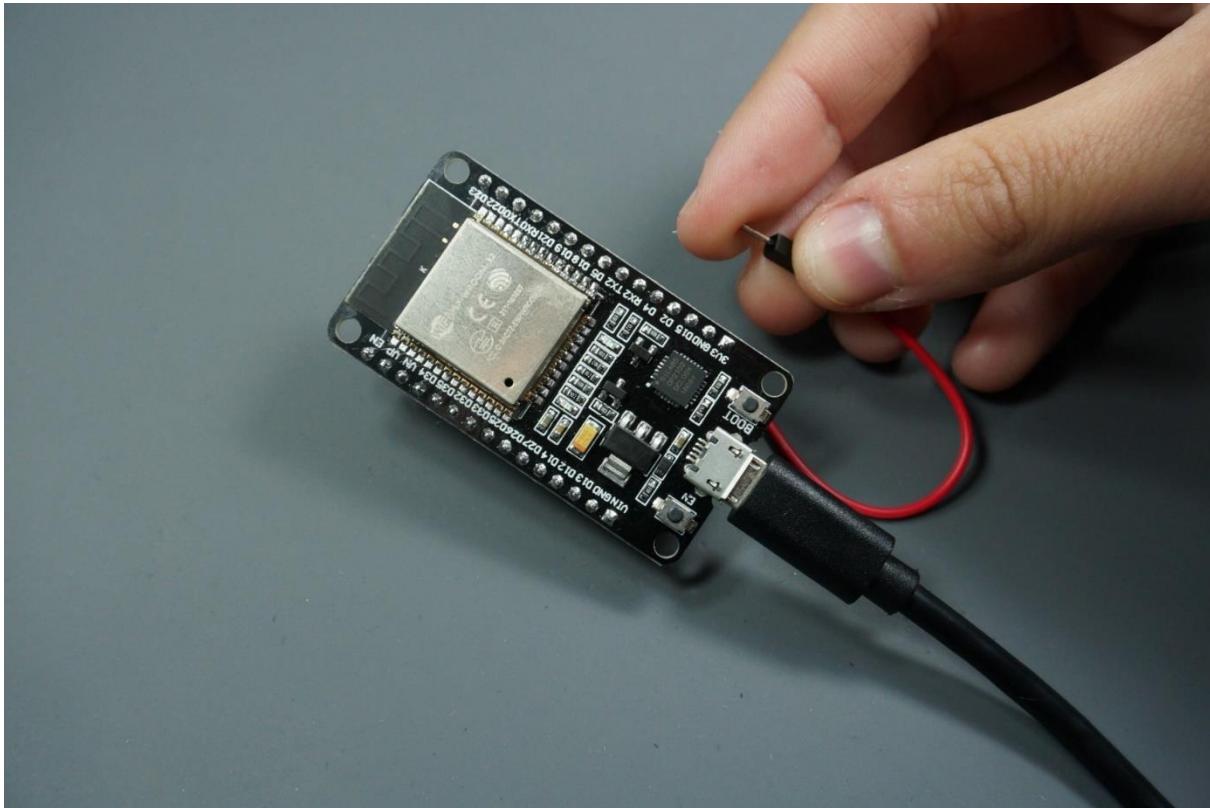
If you're using an ESP32-S2 or ESP32-S3 model, please check the location of your touch pins.

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 the following on the Serial Monitor: the boot number, the wake-up cause, and in which GPIO touch was detected.

Output Serial Monitor ▼ ⏱ ⌂

Message (Enter to send message to 'DOIT ESP32 DEVKIT V1'...) New Line 115200 baud

```
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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Boot number: 4
Wakeup caused by touchpad
Touch detected on GPIO 15
Going to sleep now
```

Ln 81, Col 1 DOIT ESP32 DEVKIT V1 on COM3 ↳ 2 ☰

Wrapping Up

This unit showed you how to wake up the ESP32 using the touch pins. In summary:

- First, define the wake-up source by calling the `touchSleepWakeUpEnable()` and passing as arguments the touch pin and the threshold value.
- Then, use the `esp_deep_sleep_start()` function to put the ESP32 in deep sleep mode.

4.4 - Deep Sleep External Wake Up

The ESP32 can awake from sleep with an external wake-up source.



An external wake-up source can be the press of a pushbutton, the detection of movement, or other scenarios that involve changing the value of a signal: from HIGH to LOW or LOW to HIGH.

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 are 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 the 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: you can only use pins that are **RTC GPIOs** with this wake-up source. Here's a list of the RTC GPIOs for different ESP32 chip models:

- **ESP32:** 0, 2, 4, 12-15, 25-27, 32-39;
- **ESP32-S2:** 0-21;
- **ESP32-S3:** 0-21.

External Wake Up (ext1)

This wake-up source allows you to use multiple RTC GPIOs. 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_io(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 one of the following options:
 - `ESP_EXT1_WAKEUP_ALL_LOW`: wake up when all GPIOs go low;
 - `ESP_EXT1_WAKEUP_ANY_HIGH`: wake up when any of the GPIOs go high.

If you're using an ESP32-S2, ESP32-S3, ESP32-C6 or ESP32-H2, these are the available modes:

- `ESP_EXT1_WAKEUP_ANY_LOW`: wake up when any of the selected GPIOs is low
- `ESP_EXT1_WAKEUP_ANY_HIGH`: wake up when any of the selected GPIOs is high

Note: you can only use pins that are RTC GPIOs.

For all the details about the ext1 deep sleep wake-up source, take a look at the [Espressif deep sleep documentation](#).

Code: External Wake-Up Example

The following example demonstrates how to use the external wake-up source. You can find it at **File > Examples > ESP32 > Deep Sleep > ExternalWakeUp**, or you can download it at the link below.

- [Click here to download the code.](#)

```
#include "driver/rtc_io.h"

#define BUTTON_PIN_BITMASK(GPIO) (1ULL << GPIO) // 2 ^ GPIO_NUMBER in hex
#define USE_EXT0_WAKEUP 1 // 1 = EXT0 wakeup, 0 = EXT1 wakeup
#define WAKEUP_GPIO GPIO_NUM_33 // Only RTC IO are allowed - ESP32 Pin example
RTC_DATA_ATTR int bootCount = 0;

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

    wakeup_reason = esp_sleep_get_wakeup_cause();

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

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

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

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

    /*First we configure the wake up source
     * We set our ESP32 to wake up for an external trigger.
```

```

There are two types for ESP32, ext0 and ext1.
ext0 uses RTC_IO to wakeup thus requires RTC peripherals
to be on while ext1 uses RTC Controller so does not need
peripherals to be powered on.
Note that using internal pullups/pulldowns also requires
RTC peripherals to be turned on. */

#ifndef USE_EXT0_WAKEUP
    esp_sleep_enable_ext0_wakeup(WAKEUP_GPIO, 1); //1 = High, 0 = Low
    // Configure pullup/downs via RTCIO to tie wakeup pins to inactive level during deepsleep
    // EXT0 resides in the same power domain (RTC_PERIPH) as the RTC IO pullup/downs
    // No need to keep that power domain explicitly, unlike EXT1.
    rtc_gpio_pullup_dis(WAKEUP_GPIO);
    rtc_gpio_pulldown_en(WAKEUP_GPIO);

#else // EXT1 WAKEUP
    // If you were to use ext1, you would use it like
    esp_sleep_enable_ext1_wakeup_io(BUTTON_PIN_BITMASK(WAKEUP_GPIO),
                                    ESP_EXT1_WAKEUP_ANY_HIGH);

    /* If there are no external pull-up/downs, tie wakeup pins to inactive level with
    internal pull-up/downs via RTC IO during deepsleep. However, RTC IO relies on the
    RTC_PERIPH power domain. Keeping this power domain on will increase some power
    consumption. However, if we turn off the RTC_PERIPH domain or if certain chips lack
    the RTC_PERIPH domain, we will use the HOLD feature to maintain the pull-up and
    pull-down on the pins during sleep. */
    // GPIO33 is tie to GND in order to wake up in HIGH
    rtc_gpio_pulldown_en(WAKEUP_GPIO);
    // Disable PULL_UP in order to allow it to wakeup on HIGH
    rtc_gpio_pullup_dis(WAKEUP_GPIO);
#endif
    // Go to sleep now
    Serial.println("Going to sleep now");
    esp_deep_sleep_start();
    Serial.println("This will never be printed");
}

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

```

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.

How Does the Code Work?

Let's take a look at the code.

This code is very similar to the previous ones in this module. 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 on the serial monitor.

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

After, you print the wake-up reason by calling the `print_wake_up_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 goes to HIGH:

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

Instead of GPIO 33, you can use any other RTC GPIO pin. Just define it on the `WAKEUP_GPIO` variable at the beginning of the code.

```
#define WAKEUP_GPIO GPIO_NUM_33 // Only RTC IO are allowed
```

Internal Pull-Up and Pull-Down Resistors

We also call the following two lines:

```
rtc_gpio_pullup_dis(WAKEUP_GPIO);  
rtc_gpio_pulldown_en(WAKEUP_GPIO);
```

The `rtc_gpio_pullup_dis()` function disables any internal pull-up resistors on the wake-up GPIO—it ensures that the pin is not unintentionally held high. This is important because a pull-up resistor would keep the pin high, potentially causing unintended wakeups.

The `rtc_gpio_pulldown_en()` function enables the internal pull-down resistor on the wake-up GPIO—it ensures the pin is held low until a valid wakeup signal (HIGH) is received. By configuring the pin with a pull-down resistor, we guarantee that it

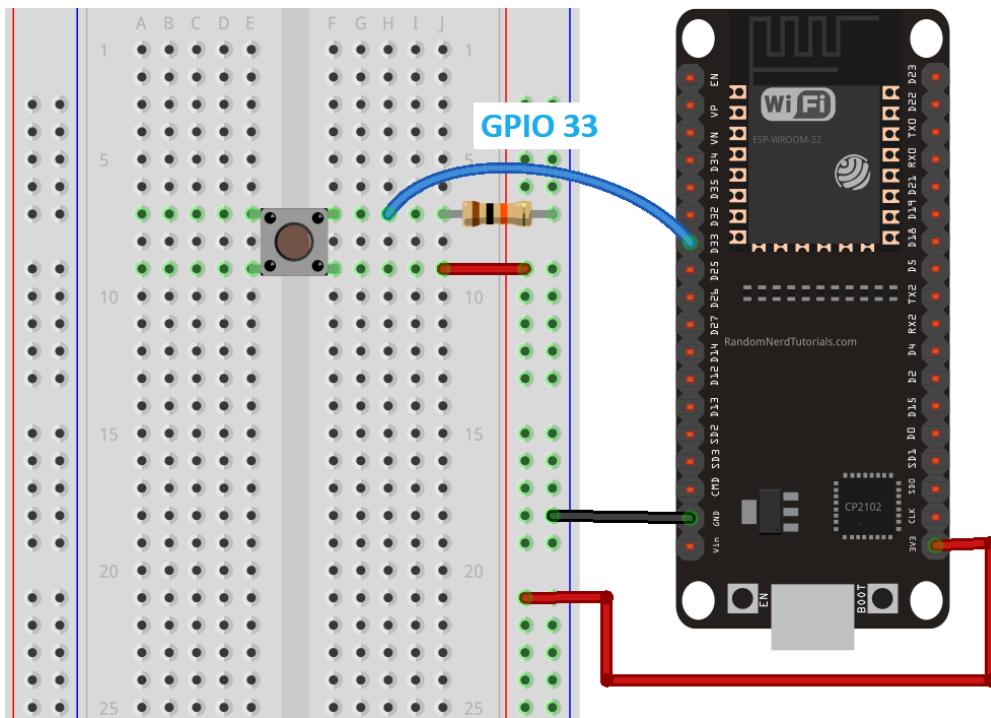
remains in a stable low state during deep sleep. This stability ensures that the ESP32 wakes up only when the specified GPIO pin receives an external high signal, matching the wakeup condition set by the `esp_sleep_enable_ext0_wakeup(WAKEUP_GPIO, 1)`.

Wiring the Circuit

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

- ESP32 DOIT DEVKIT V1 Board
 - Pushbutton
 - 10k Ohm resistor
 - Breadboard
 - Jumper wires

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.

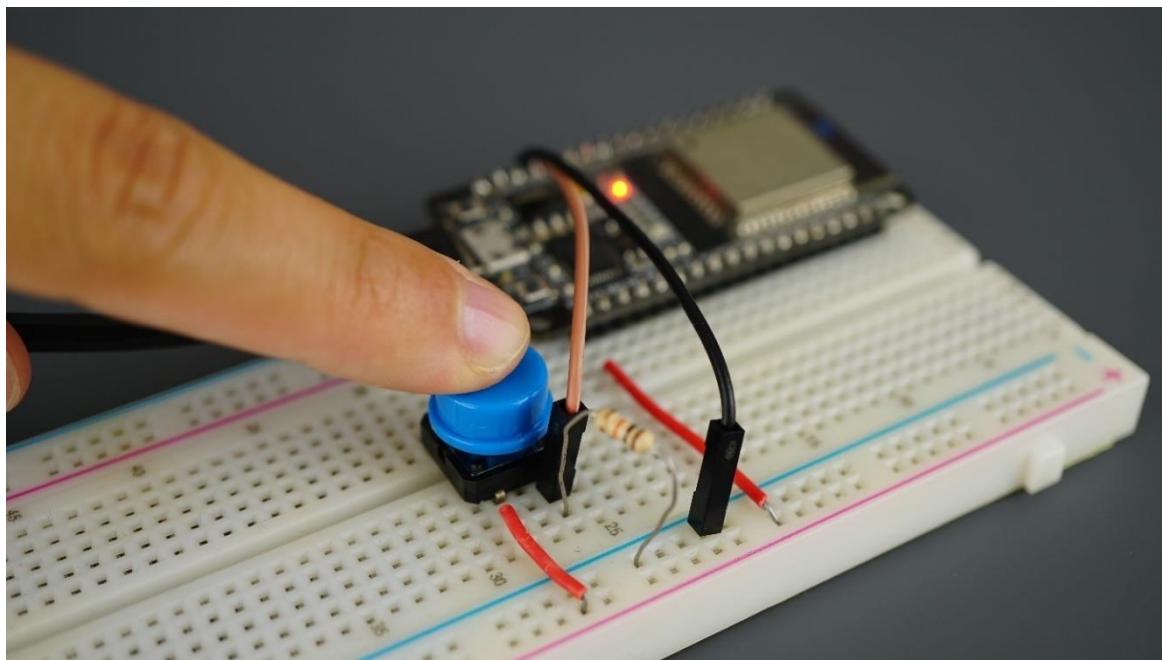


Note that only RTC GPIOs can be used as a wake-up source—you can check the pinout of your board to identify the RTC GPIOs. Instead of GPIO 33, you can use any RTC GPIO pins to connect your button.

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.

The screenshot shows a serial monitor window with two distinct sections of text, each highlighted with a red box. The top section starts with "Boot number: 2" and ends with "ets Jun 8 2016 00:22:57". The bottom section starts with "Boot number: 3" and ends with "Going to sleep now". Both sections include the message "Wakeup caused by external signal using RTC_IO". The rest of the text in the window is standard boot log information.

```
Output Serial Monitor X
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1') New Line 115200 baud
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Boot number: 2
Wakeup caused by external signal using RTC_IO
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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Boot number: 3
Wakeup caused by external signal using RTC_IO
Going to sleep now
```

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

What if you want different buttons, all wake up the ESP32, but perform different tasks? For that, you need to use the `ext1` method.

ext1

The `ext1` wake-up source allows you to wake up the ESP32 using different GPIOs and perform different tasks depending on the GPIO that caused the wake-up.

Instead of using the `esp_sleep_enable_ext0_wakeup()` function, you use the `esp_sleep_enable_ext1_wakeup_io` function.

```
esp_sleep_enable_ext1_wakeup_io(BUTTON_PIN_BITMASK(WAKEUP_GPIO),  
ESP_EXT1_WAKEUP_ANY_HIGH);
```

In this particular example, to run that part of the code, make sure you set the `USE_EXT0_WAKEUP` variable to `0` at the beginning of the code like so:

```
#define USE_EXT0_WAKEUP 0 // 1 = EXT0 wakeup, 0 = EXT1 wakeup
```

The `esp_sleep_enable_ext1_wakeup_io()` function

The first argument of the `esp_sleep_enable_ext1_wakeup_io()` 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.

GPIOs Bitmask

To define the GPIOs bitmask, we can use the `BUTTON_PIN_BITMASK()` macro defined at the beginning of the code.

```
#define BUTTON_PIN_BITMASK(GPIO) (1ULL << GPIO) // 2 ^ GPIO_NUMBER in hex
```

`BUTTON_PIN_BITMASK(GPIO)` is a macro that creates a bitmask for a specific GPIO. It is not a function but behaves somewhat similarly by accepting an argument and returning a value.

So, if you want to return the bitmask for a specific GPIO, you just need to call it like so:

```
BUTTON_PIN_BITMASK(WAKEUP_GPIO)
```

So, the `esp_sleep_enable_ext1_wakeup_io()` will look like this:

```
esp_sleep_enable_ext1_wakeup_io(BUTTON_PIN_BITMASK(WAKEUP_GPIO),  
ESP_EXT1_WAKEUP_ANY_HIGH);
```

Bitmask for Multiple GPIOs

To create a bitmask for multiple GPIOs using the `BUTTON_PIN_BITMASK(GPIO)` macro, you can use bitwise OR (`|`) to combine the individual bitmasks for each GPIO pin. Here's how you can do it:

Suppose you want to create a bitmask for GPIO pins 2 and 15. You can do this by combining the individual bitmasks for each pin like this:

```
uint64_t bitmask = BUTTON_PIN_BITMASK(GPIO_NUM_2) | BUTTON_PIN_BITMASK(GPIO_NUM_15);
```

We'll see an example using multiple GPIOs as a wake-up source next.

Wake-Up Mode

The second argument of the `esp_sleep_enable_ext1_wakeup_io()` function is the wake-up mode. As we've seen previously, these are your options:

- `ESP_EXT1_WAKEUP_ALL_LOW`: wake up when all GPIOs go low;
- `ESP_EXT1_WAKEUP_ANY_HIGH`: wake up when any of the GPIOs go high.

If you're using an ESP32-S2, ESP32-S3, ESP32-C6 or ESP32-H2, these are the available modes:

- `ESP_EXT1_WAKEUP_ANY_LOW`: wake up when any of the selected GPIOs is low
- `ESP_EXT1_WAKEUP_ANY_HIGH`: wake up when any of the selected GPIOs is high

In our particular example, we're using `ESP_EXT1_WAKEUP_ANY_HIGH`. So, the ESP32 will wake-up when any of the GPIOs from the bitmask are high. Like with the `ext0`,

we disable pull-up internal resistors and enable pull-down resistors to ensure stability of the reading of the wake-up GPIO.

External Wake Up — Multiple GPIOs

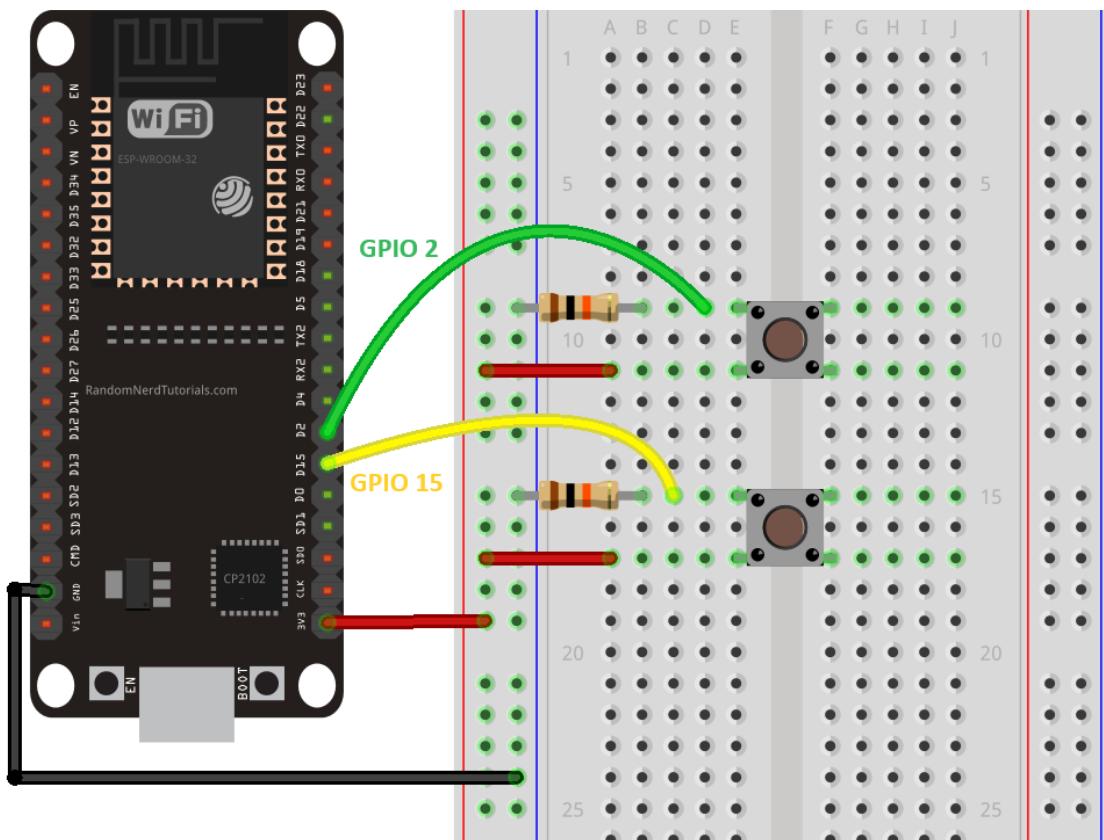
Now, let's see an example to wake up the ESP32 using different buttons (GPIOs) and identify which button caused the wake-up. In this example, we'll use GPIO 2 and GPIO 15 as a wake-up source.

Wiring the Circuit

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [2x pushbuttons](#)
- [2x 10k Ohm resistors](#)
- [Breadboard](#)
- [Jumper wires](#)

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:

- 1) create a bitmask to use GPIO 15 and GPIO 2. We've shown you how to do this before;
- 2) enable ext1 as a wake-up source;
- 3) create a function that identifies the GPIO that caused the wake-up.

The next sketch has all those changes implemented.

- [Click here to download the code.](#)

```
// Deep Sleep with External Wake Up (Multiple Wake-up GPIOs)

#include "driver/rtc_io.h"

#define BUTTON_PIN_BITMASK(GPIO) (1ULL << GPIO) // 2 ^ GPIO_NUMBER in hex
#define WAKEUP_GPIO_2 GPIO_NUM_2 // Only RTC IO are allowed - ESP32 Pin example
#define WAKEUP_GPIO_15 GPIO_NUM_15 // Only RTC IO are allowed - ESP32 Pin example

// Define bitmask for multiple GPIOs
uint64_t bitmask = BUTTON_PIN_BITMASK(GPIO_NUM_2) |
                   BUTTON_PIN_BITMASK(GPIO_NUM_15);

RTC_DATA_ATTR int bootCount = 0;

// Method to print the GPIO that triggered the wakeup
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);
}

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

    wakeup_reason = esp_sleep_get_wakeup_cause();

    switch (wakeup_reason) {
        case ESP_SLEEP_WAKEUP_EXT0:
            Serial.println("Wakeup caused by external signal using RTC_IO");
            break;
        case ESP_SLEEP_WAKEUP_EXT1:
            Serial.println("Wakeup caused by external signal using RTC_CNTL");
            print_GPIO_wake_up();
            break;
        case ESP_SLEEP_WAKEUP_TIMER:
            Serial.println("Wakeup caused by timer");
            break;
        case ESP_SLEEP_WAKEUP_TOUCHPAD:
            Serial.println("Wakeup caused by touchpad");
            break;
        case ESP_SLEEP_WAKEUP_ULP:
            break;
    }
}
```

```

        Serial.println("Wakeup caused by ULP program");
        break;
    default:
        Serial.printf("Wakeup was not caused by deep sleep: %d\n", wakeup_reason);
        break;
    }
}

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

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

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

    // Use ext1 as a wake-up source
    esp_sleep_enable_ext1_wakeup_io(bitmask, ESP_EXT1_WAKEUP_ANY_HIGH);
    // enable pull-down resistors and disable pull-up resistors
    rtc_gpio_pulldown_en(WAKEUP_GPIO_2);
    rtc_gpio_pullup_dis(WAKEUP_GPIO_2);
    rtc_gpio_pulldown_en(WAKEUP_GPIO_15);
    rtc_gpio_pullup_dis(WAKEUP_GPIO_15);

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

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

```

How Does the Code Work?

Let's take a quick look at how the code works.

GPIOs Bitmask

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

```

#define BUTTON_PIN_BITMASK(GPIO) (1ULL << GPIO) // 2 ^ GPIO_NUMBER in hex
#define WAKEUP_GPIO_2             GPIO_NUM_2      // Only RTC IO are allowed -
#define WAKEUP_GPIO_15            GPIO_NUM_15     // Only RTC IO are allowed

// Define bitmask for multiple GPIOs
uint64_t bitmask = BUTTON_PIN_BITMASK(WAKEUP_GPIO_2) | BUTTON_PIN_BITMASK(WAKEUP_GPIO_15);

```

Identify the GPIO that Caused the Wake-up

We create a function called `print_GPIO_wake_up()` that prints 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);
}
```

The `esp_sleep_get_ext1_wakeup_status()` function returns a bitmask with the GPIO that caused the wake-up. We can get the GPIO number as follows:

```
log(GPIO_reason))/log(2)
```

Print the Wake-Up Reason

We modified the `print_wakeup_reason()` function to print the GPIO that caused the wake-up when the wake-up source is ext1:

```
case ESP_SLEEP_WAKEUP_EXT1:
    Serial.println("Wakeup caused by external signal using RTC_CNTL");
    print_GPIO_wake_up();
    break;
```

Enable ext1 as a Wake-Up Source

Enable `ext1` as a wake-up source by passing the GPIOs bitmask and wake-up mode.

```
esp_sleep_enable_ext1_wakeup_io(bitmask, ESP_EXT1_WAKEUP_ANY_HIGH);
```

Don't forget to disable any internal pull-up resistors and enable pull-down resistors on the GPIOs used as a wake-up source.

```
rtc_gpio_pulldown_en(WAKEUP_GPIO_2);
rtc_gpio_pullup_dis(WAKEUP_GPIO_2);
rtc_gpio_pulldown_en(WAKEUP_GPIO_15);
rtc_gpio_pullup_dis(WAKEUP_GPIO_15);
```

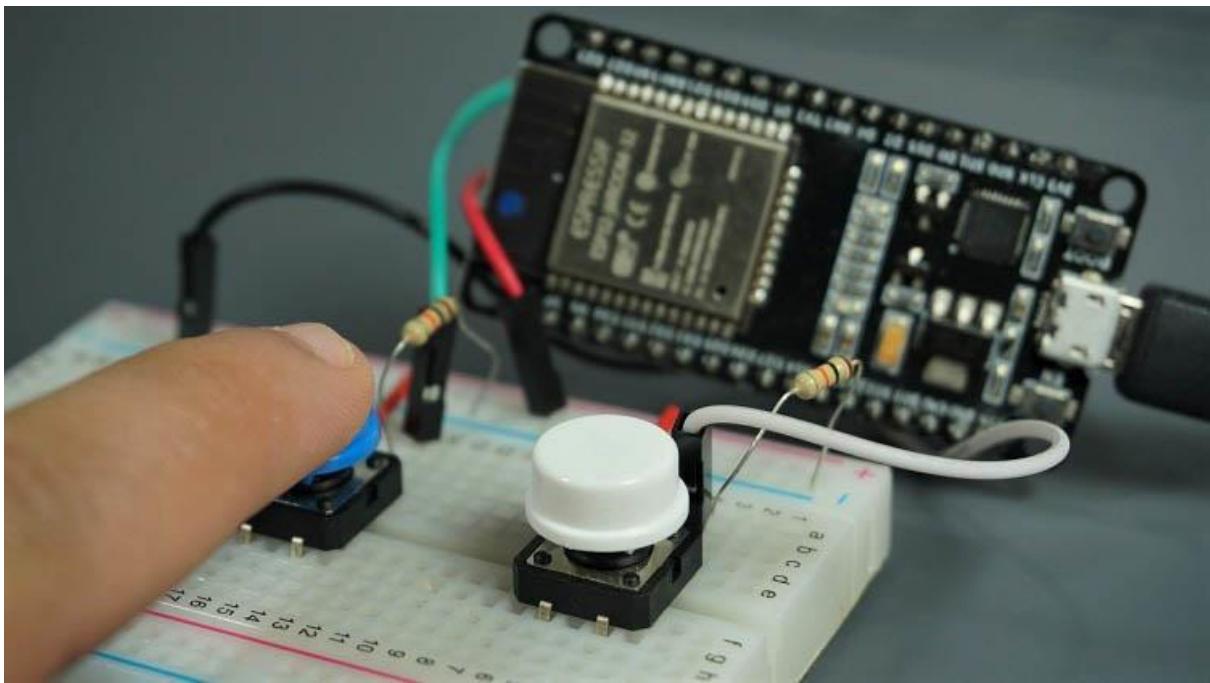
Enable Deep Sleep

Finally, call `esp_deep_sleep_start()` to put the ESP32 in deep sleep mode.

```
esp_deep_sleep_start();
```

Testing the Example

After uploading the code to the board, the ESP32 will be in deep sleep mode. 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. On the Serial Monitor, you should get the wake-up reason and the GPIO that caused the wake-up.

```
Output Serial Monitor ×
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'C') New Line 115200 baud
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Boot number: 2
Wakeup caused by external signal using RTC_CNTL
GPIO that triggered the wake up: GPIO 15
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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Boot number: 3
Wakeup caused by external signal using RTC_CNTL
GPIO that triggered the wake up: GPIO 2
Going to sleep now

Ln 54, Col 6 DOIT ESP32 DEVKIT V1 on COM3 2 2
```

Wrapping Up

In this unit, you learned to wake up the ESP32 using an external wake-up. This means that you can wake up the ESP32 by changing the state of a GPIO pin.

In summary:

- 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 multiple GPIO pins.

MODULE 5

Introducing Wi-Fi

5.1 - Introducing Wi-Fi

The ESP32 supports Bluetooth and 2.4 GHz Wi-Fi capabilities, which are both useful in IoT and home automation. In this Unit, you'll get started with using Wi-Fi on the ESP32. Learn the difference between Wi-Fi station and access point, how to scan for nearby Wi-Fi networks, connect your ESP32 to your local network, get information about its IP address, and other useful Wi-Fi functions.

The Wi-Fi Library

The first thing you need to do to use the ESP32 Wi-Fi functionalities is to include the `WiFi.h` library in your code, as follows:

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

This library is automatically “installed” when you install the ESP32 add-on in your Arduino IDE. So, you don't need to install it.

ESP32 Wi-Fi Modes

The ESP32 board can act as Wi-Fi Station, Access Point or both. To set the Wi-Fi mode, use `WiFi.mode()` and set the desired mode as argument:

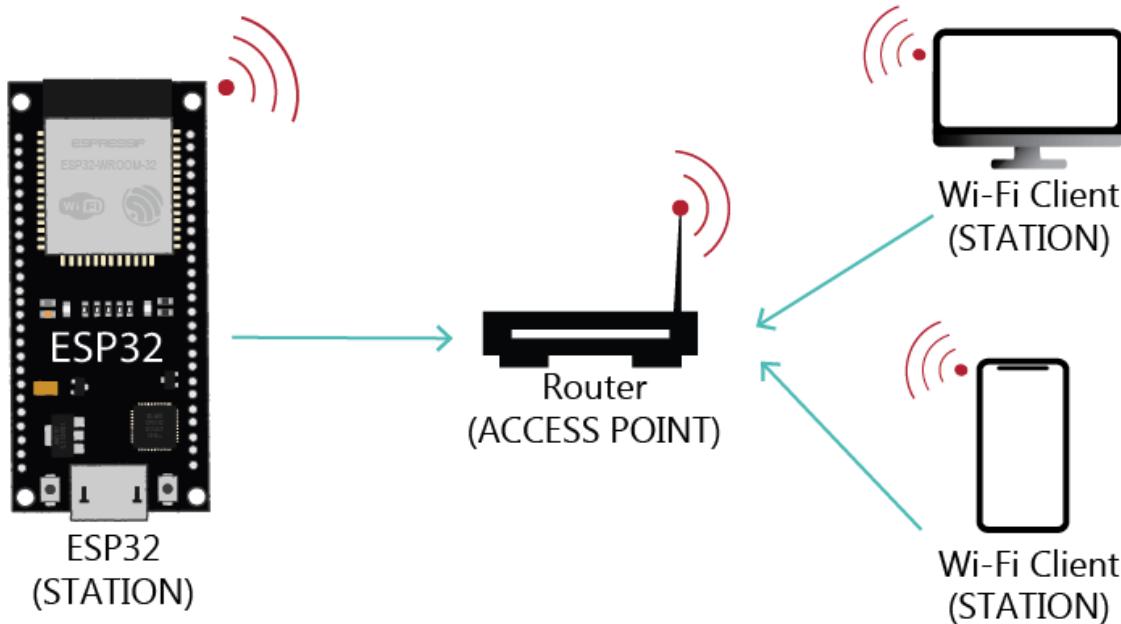
<code>WiFi.mode(WIFI_STA)</code>	Station mode: the ESP32 connects to an Access Point
<code>WiFi.mode(WIFI_AP)</code>	Access Point mode: Stations can connect to the ESP32
<code>WiFi.mode(WIFI_AP_STA)</code>	Access Point and a Station connected to another Access Point

Wi-Fi Station and Wi-Fi Access Point

The ESP32/ESP8266 board can act as Wi-Fi Station, Access Point or both.

Wi-Fi Station

When the ESP32 or ESP8266 is set as a Wi-Fi station, it can connect to other networks (like connecting to your router—local network). In this scenario, the router assigns a unique IP address to your ESP board. You can communicate with the ESP using other devices (stations) that are also connected to the same network by referring to the ESP32 unique IP address.

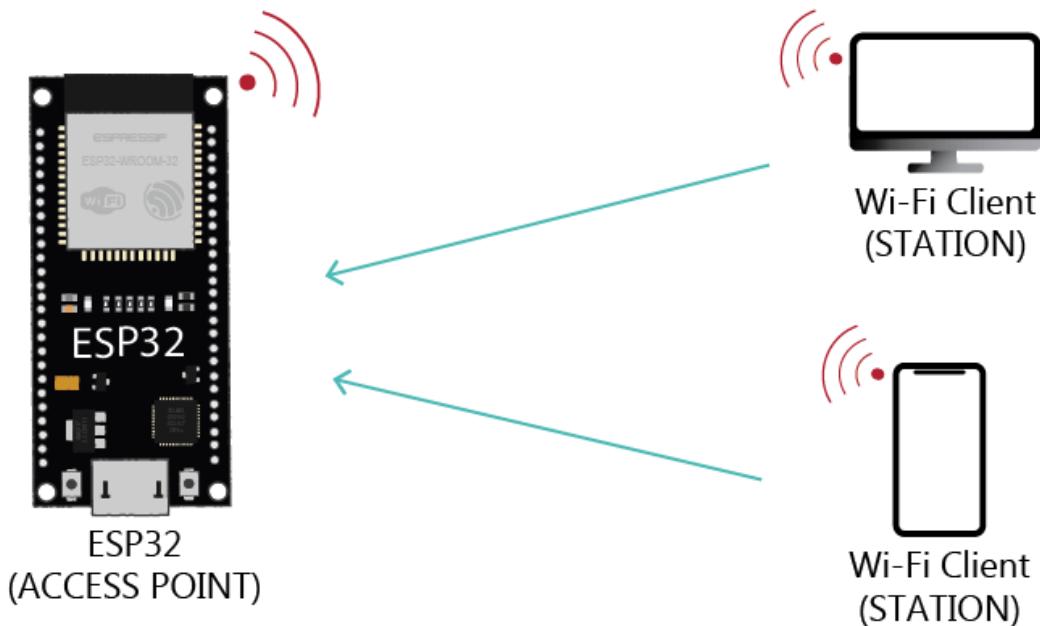


The router is connected to the internet, so we can request information from the internet using the ESP32 board like data from APIs (weather data, for example), publish data to online platforms, use icons and images from the internet or include JavaScript libraries to build web server pages, and more.

In some cases, this might not be the best configuration – when you don't have a network nearby and want you still want to connect to the ESP via Wi-Fi to control it. In this scenario, you must set your ESP32 board as an access point.

Access Point

When you set your ESP32 board as an access point, you can be connected using any device with Wi-Fi capabilities without connecting to your router. When you set the ESP32 as an access point, you create its own Wi-Fi network, and nearby Wi-Fi devices (stations) can connect to it, like your smartphone or computer. So, you don't need to be connected to a router to control it.



Unlike a router, an ESP Access Point doesn't connect further to a wired network or the Internet. This means that if you try to load libraries or use firmware from the internet, it will not work. It also doesn't work if you make HTTP requests to services on the internet to publish sensor readings to the cloud or use services on the internet (like sending an email, for example).

Set the ESP32 as an Access Point

To set the ESP32 as an access point, set the Wi-Fi mode to access point:

```
WiFi.mode(WIFI_AP);
```

And then, use the `softAP()` method as follows:

```
WiFi.softAP(ssid, password);
```

`ssid` is the name you want to give to the ESP32 access point, and the `password` variable is the password for the access point. If you don't want to set a password, set it to `NULL`.

There are also other optional parameters you can pass to the `softAP()` method.

Here are all the parameters:

```
WiFi.softAP(const char* ssid, const char* password, int channel,
           int ssid_hidden, int max_connection);
```

- `ssid`: name for the access point – maximum of 63 characters;
- `password`: minimum of 8 characters; set to NULL if you want the access point to be open;
- `channel`: Wi-Fi channel number (1-13)
- `ssid_hidden`: (0 = broadcast SSID, 1 = hide SSID)
- `max_connection`: maximum simultaneous connected clients (1-4)

Wi-Fi Station + Access Point

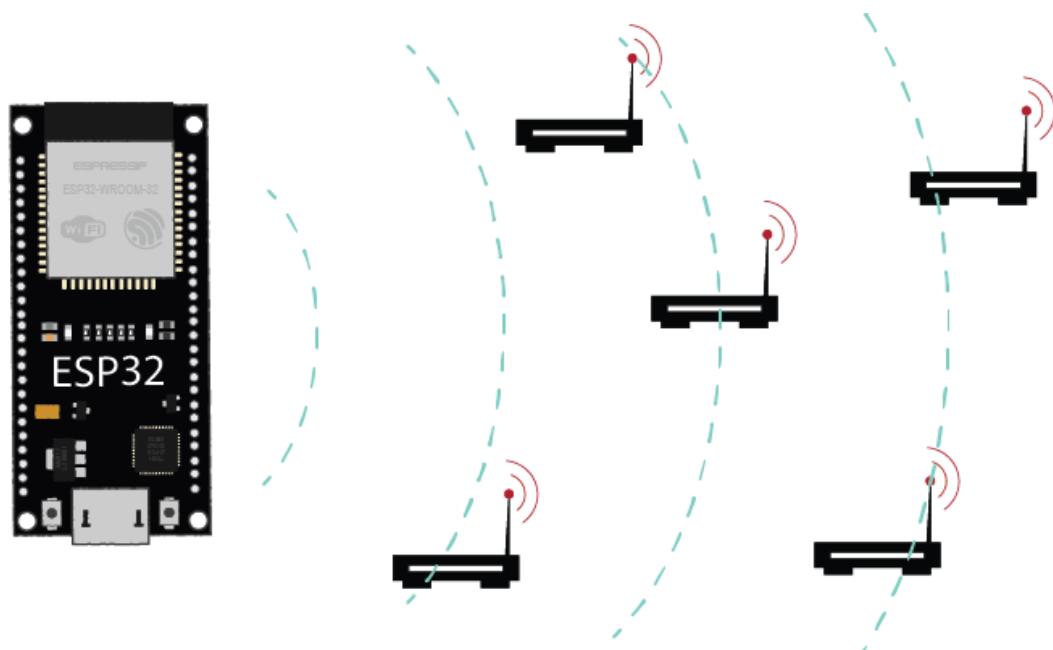
The ESP32 can be set as a Wi-Fi station and access point simultaneously. Set its mode to `WIFI_AP_STA`.

```
WiFi.mode(WIFI_AP_STA);
```

Scan Wi-Fi Networks

The ESP32 can scan nearby Wi-Fi networks within its Wi-Fi range. In the Arduino IDE, 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.



Alternatively, you can download the code using the link below:

- [Click here to download the code.](#)

```

#include "WiFi.h"

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

  // Set WiFi to station mode and disconnect from an AP if it was previously connected
  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(100);

  Serial.println("Setup done");
}

void loop() {
  Serial.println("Scan start");

  // WiFi.scanNetworks will return the number of networks found.
  int n = WiFi.scanNetworks();
  Serial.println("Scan done");
  if (n == 0) {
    Serial.println("no networks found");
  } else {
    Serial.print(n);
    Serial.println(" networks found");
    Serial.println("Nr | SSID | RSSI | CH | Encryption");
    for (int i = 0; i < n; ++i) {
      // Print SSID and RSSI for each network found
      Serial.printf("%2d", i + 1);
      Serial.print(" | ");
      Serial.printf("%-32.32s", WiFi.SSID(i).c_str());
      Serial.print(" | ");
      Serial.printf("%4ld", WiFi.RSSI(i));
      Serial.print(" | ");
      Serial.printf("%2ld", WiFi.channel(i));
      Serial.print(" | ");
      switch (WiFi.encryptionType(i)) {
        case WIFI_AUTH_OPEN:           Serial.print("open"); break;
        case WIFI_AUTH_WEP:            Serial.print("WEP"); break;
        case WIFI_AUTH_WPA_PSK:        Serial.print("WPA"); break;
        case WIFI_AUTH_WPA2_PSK:       Serial.print("WPA2"); break;
        case WIFI_AUTH_WPA_WPA2_PSK:   Serial.print("WPA+WPA2"); break;
        case WIFI_AUTH_WPA2_ENTERPRISE:Serial.print("WPA2-EAP"); break;
        case WIFI_AUTH_WPA3_PSK:       Serial.print("WPA3"); break;
        case WIFI_AUTH_WPA2_WPA3_PSK:  Serial.print("WPA2+WPA3"); break;
        case WIFI_AUTH_WAPI_PSK:       Serial.print("WAPI"); break;
        default:                      Serial.print("unknown");
      }
      Serial.println();
      delay(10);
    }
  }
  Serial.println("");

  // Delete the scan result to free memory for code below.
  WiFi.scanDelete();

  // Wait a bit before scanning again.
  delay(5000);
}

```

This can be useful for verifying whether the Wi-Fi network you're trying to connect to is within range of your board or other devices. Often, Wi-Fi projects fail because the connection to the router is too weak due to insufficient signal strength.

You can upload the previous sketch to your board and check the available networks as well as the RSSI (received signal strength indicator).

Let's take a look at the main functions used in this code.

`WiFi.scanNetworks()` returns the number of networks found.

```
int n = WiFi.scanNetworks();
```

After the scanning, you can access the parameters of each network.

`WiFi.SSID()` gets the SSID for a specific network. The following line in the code prints the SSID of the network.

```
Serial.printf("%-32.32s", WiFi.SSID(i).c_str());
```

`WiFi.RSSI()` returns the RSSI of that network. RSSI stands for Received Signal Strength Indicator. It is an estimated measure of the power level that an RF client device is receiving from an access point or router.

```
Serial.printf("%4ld", WiFi.RSSI(i));
```

We can get the Wi-Fi channel the board is using by calling `WiFi.channel(i)`.

```
Serial.printf("%2ld", WiFi.channel(i));
```

Finally, `WiFi.encryptionType()` returns the network encryption type. The encryption type can return all the scenarios specified in the following lines:

```
switch (WiFi.encryptionType(i)) {
    case WIFI_AUTH_OPEN:           Serial.print("open"); break;
    case WIFI_AUTH_WEP:            Serial.print("WEP"); break;
    case WIFI_AUTH_WPA_PSK:        Serial.print("WPA"); break;
    case WIFI_AUTH_WPA2_PSK:       Serial.print("WPA2"); break;
    case WIFI_AUTH_WPA_WPA2_PSK:   Serial.print("WPA+WPA2"); break;
    case WIFI_AUTH_WPA2_ENTERPRISE: Serial.print("WPA2-EAP"); break;
    case WIFI_AUTH_WPA3_PSK:       Serial.print("WPA3"); break;
    case WIFI_AUTH_WPA2_WPA3_PSK:  Serial.print("WPA2+WPA3"); break;
    case WIFI_AUTH_WAPI_PSK:       Serial.print("WAPI"); break;
    default:                      Serial.print("unknown");
}
```

After running the previous example on your board, you should get something similar to the following picture.



The screenshot shows the Arduino Serial Monitor window. The title bar says "Output Serial Monitor". The message area displays the output of a Wi-Fi scan:

```
Scan done
10 networks found
Nr | SSID | RSSI | CH | Encryption
1 | MEO-D' .. | -67 | 12 | WPA2
2 | MEO-WiFi | -67 | 12 | open
3 | MEO----- ..-N | -67 | 12 | WPA2
4 | MEO-00000000 | -81 | 6 | WPA2
5 | MEO-WiFi | -81 | 6 | open
6 | MEO-0L.. | -85 | 6 | WPA+WPA2
7 | MEO-WiFi | -85 | 6 | open
8 | MEO-6.. _7 | -86 | 6 | WPA+WPA2
9 | MEO-WiFi | -88 | 6 | open
10 | MEO-2D' 0 | -93 | 6 | WPA+WPA2
```

The status bar at the bottom shows "Ln 51, Col 65 DOIT ESP32 DEVKIT V1 on COM3" and icons for serial port and file.

Connect to a Wi-Fi Network

To connect the ESP32 to a specific Wi-Fi network, you must know its SSID and password. Additionally, that network must be within the ESP32 Wi-Fi range (to check that, you can use the previous example to scan Wi-Fi networks).

You can use the following function to connect the ESP32 to a Wi-Fi network `initWiFi()`:

```
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}
```

The `ssid` and `password` variables hold the SSID and password of the network you want to connect to. You can define them in your code like this.

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

Then, you simply need to call the `initWiFi()` function in your `setup()`.

How Does it Work?

Let's take a quick look at how this function works.

First, set the Wi-Fi mode. If the ESP32 will connect to another network (access point/hotspot) it must be in station mode.

```
WiFi.mode(WIFI_STA);
```

Then, use `WiFi.begin()` to connect to a network. You must pass as arguments the network SSID and its password:

```
WiFi.begin(ssid, password);
```

Connecting to a Wi-Fi network can take a while, so we usually add a `while` loop that keeps checking if the connection was already established using `WiFi.status()`. When the connection is successfully established, it returns `WL_CONNECTED`.

```
while (WiFi.status() != WL_CONNECTED) {
```

We'll see a complete example of connecting your ESP32 to a network soon.

Get the Wi-Fi Connection Status

To get the status of the Wi-Fi connection, you can use `WiFi.status()`. This returns one of the following values that correspond to the constants on the table:

Value	Constant	Meaning
0	WL_IDLE_STATUS	temporary status assigned when <code>WiFi.begin()</code> is called
1	WL_NO_SSID_AVAIL	when no SSID are available
2	WL_SCAN_COMPLETED	scan networks are completed
3	WL_CONNECTED	when connected to a Wi-Fi network
4	WL_CONNECT_FAILED	when the connection fails for all the attempts
5	WL_CONNECTION_LOST	when the connection is lost
6	WL_DISCONNECTED	when disconnected from a network

Get Wi-Fi Connection Strength

To get the Wi-Fi connection strength, you can simply call `WiFi.RSSI()` after a successful Wi-Fi connection.

Get ESP32 IP Address

When the ESP32 is set as a Wi-Fi station, it can connect to other networks (like your router). In this scenario, the router assigns a unique IP address to your ESP32 board. To get your board's IP address, you need to call `WiFi.localIP()` after establishing a connection with your network.

```
Serial.println(WiFi.localIP());
```

Code Example: Connect to a Network, Get the IP Address and Connection Strength

The following code shows an example of how to connect your board to a nearby network, get its IP address, and the strength of the network.

- [Click here to download the code.](#)

```
#include <WiFi.h>

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

void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi ..");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }
  Serial.println(WiFi.localIP());
}

void setup() {
  Serial.begin(115200);
  initWiFi();
  Serial.print("RSSI: ");
  Serial.println(WiFi.RSSI());
}

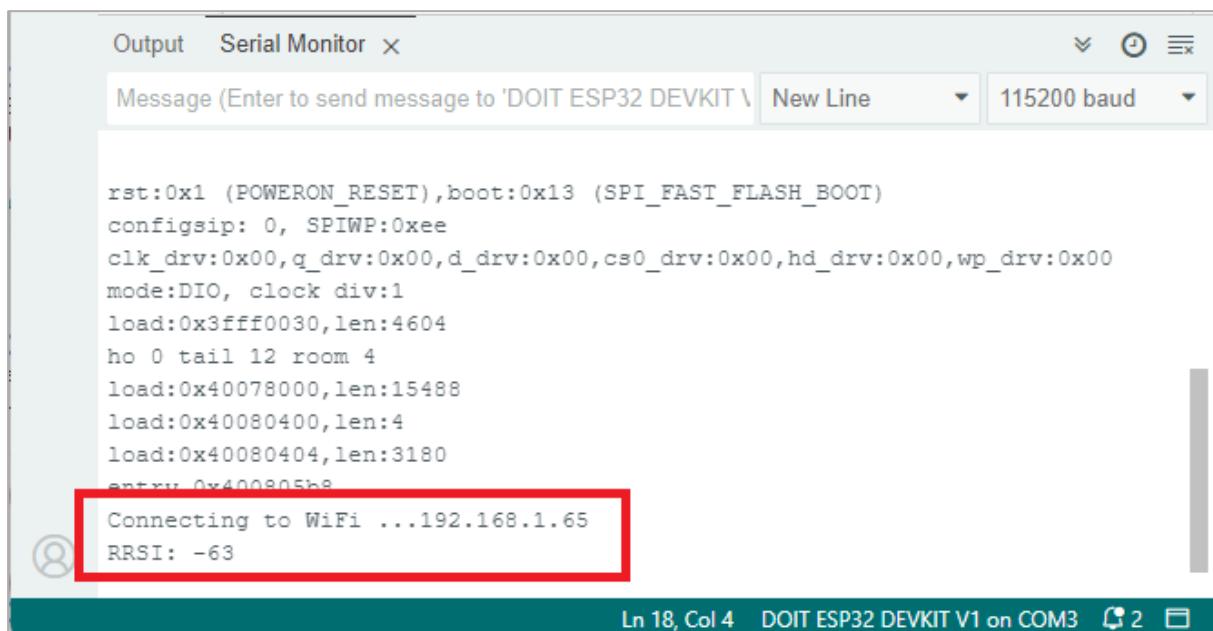
void loop() {
```

```
// put your main code here, to run repeatedly:  
}
```

Insert the network credentials on the following lines and then upload the code to your board.

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

After uploading the code, open the Serial Monitor and press the ESP32 on-board RST button. It will connect to your network, and print the ESP32 IP address and the RSSI (received signal strength indicator) of the connection. In our case, the IP address assigned to the ESP32 board is 192.168.1.65. In the case of the RSSI, a lower absolute value means a strongest Wi-Fi connection.



Set a Static ESP32 IP Address

Instead of getting a randomly assigned IP address, you can set an available IP address of your preference to the ESP32 using the `WiFi.config()` method. Why is that useful?

In the case of my network, even if I restart the ESP32 board several times, my router will always assign the same IP address to that device. However, that's not the case for some other networks. In that scenario, it's nice to have a way to set a static IP address to your board, so that it doesn't change every time it resets.

Additionally, it might also be useful for projects that are not connected to your computer or don't have a way to display the IP address.

The `WiFi.config()` method accepts the following parameters as arguments:

```
WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS);
```

The first argument corresponds to the IP address you want to attribute. When choosing the static IP address for your ESP32 board, make sure that the address is not already being used by another device. To do that, you can use a software like *Angry IP Scanner* (compatible with Windows and Mac OS). Or you can log in into your router dashboard and check which IP addresses are already assigned.

Then, you need to set the gateway, subnet, primary DNS, and secondary DNS.

How to find those values?

There are different ways to get information about the subnet mask and the gateway IP. The easiest way is to open a Terminal window on your computer and run the following command:

On Windows:

```
ipconfig
```

On MacOS or Linux:

```
ifconfig
```



It will return the information you're looking for: the gateway IP (the IP address of your router) and the subnet mask.

```
Windows PowerShell      X + - X
Connection-specific DNS Suffix . :
Wireless LAN adapter Wi-Fi:

Connection-specific DNS Suffix . : Home
IPv6 Address . . . . . : 2001:8a0::3000:80ff:fe27:1a03
Temporary IPv6 Address . . . . . : 2001:ca0::a307:7609:d8e6:1a5c:e5b2:f6a3
Link-local IPv6 Address . . . . . : fe80::a307:7609:d8e6:1a5c%17
IPv4 Address . . . . . : 192.168.1.66
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.1.254
Ethernet adapter Bluetooth Network Connection:
```

In my case, the subnet mask is 255.255.255.0 and the gateway IP address is 192.168.1.254.

For the primary DNS server, you can always use 8.8.8.8, which is Google's public DNS server. For secondary DNS, you can use 8.8.4.4.

After having all the required parameters, here's how to set a static IP address.

Outside the `setup()` and `loop()` functions, define the following variables with your own static IP address and corresponding gateway IP address, and the primary and secondary DNS. For example:

```
// Set your Static IP address
IPAddress local_IP(192, 168, 1, 184);
// Set your Gateway IP address
IPAddress gateway(192, 168, 1, 1);

IPAddress subnet(255, 255, 255, 0);
IPAddress primaryDNS(8, 8, 8, 8);
IPAddress secondaryDNS(8, 8, 4, 4);
```

Then, in the `setup()` you need to call the `WiFi.config()` method before connecting to a network.

```
if (!WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS)) {
    Serial.println("STA Failed to configure");
}
```

Here's a complete example sketch that sets a static IP address to the ESP32.

- [Click here to download the code.](#)

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

```

// Set your Static IP address
IPAddress local_IP(192, 168, 1, 184);
// Set your Gateway IP address
IPAddress gateway(192, 168, 1, 1);

IPAddress subnet(255, 255, 255, 0);
IPAddress primaryDNS(8, 8, 8, 8);
IPAddress secondaryDNS(8, 8, 4, 4);

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

void initWiFi() {
    WiFi.mode(WIFI_STA);

    // Set static IP address
    if (!WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS)) {
        Serial.println("STA Failed to configure");
    }

    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void setup() {
    Serial.begin(115200);
    initWiFi();
    Serial.print("RSSI: ");
    Serial.println(WiFi.RSSI());
}

void loop() {
    // put your main code here, to run repeatedly:
}

```

Disconnect from Wi-Fi Network

To disconnect from a previously connected Wi-Fi network, use `WiFi.disconnect()`:

```
WiFi.disconnect();
```

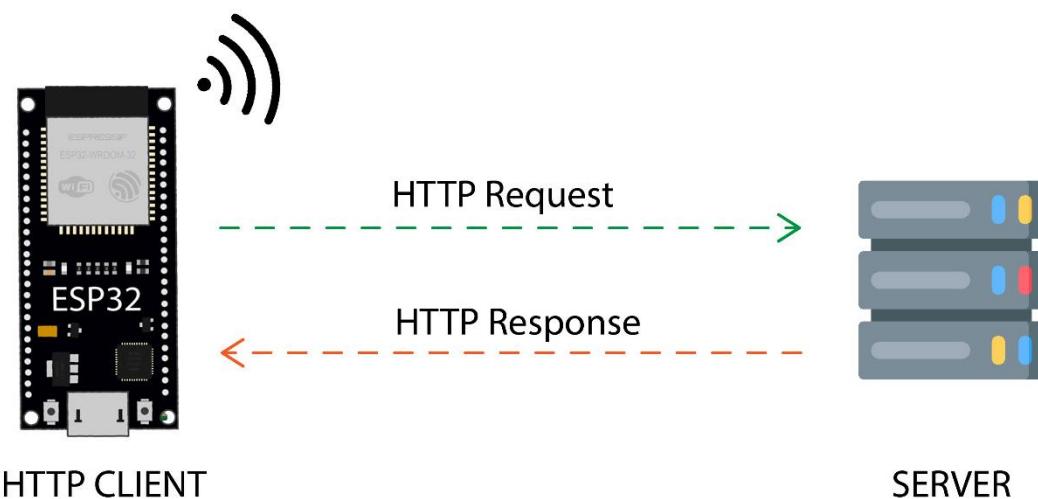
5.2 - Getting Started with HTTP Requests

In this Unit, you'll learn the basics of HTTP requests. When it comes to IoT projects, HTTP requests are fundamental to interact with web services to retrieve data from external sources, integrate with Web APIs to access web services, or even remote control your ESP32 using external third-party platforms.

What are HTTP Requests?

The most common way to exchange data with another computer (server) on the internet is using the HTTP protocol.

HTTP (Hypertext Transfer Protocol) works through a client-server model. In this model, the client communicates with a server using HTTP requests. An HTTP request is a message sent by the client to a server, typically to request a specific action or retrieve information. The server sends a response back to the client also through HTTP.



- **Client:** initiates communication through a request.
- **Server:** receives and processes the request and sends a response in return.

The ESP32 can either be a client or a server. When it is a client, it sends requests to servers. When it is a server, it handles the client's requests. In this Unit, we'll take a look at the ESP32 as an HTTP client that makes requests to a server.

If you're a bit confused about why HTTP requests are important, here are some examples of what you can do using HTTP requests with the ESP32:

- Get data from the internet: for example, time, current weather, stock prices, traffic updates, and much more...
- Datalogging: send data to the cloud to save your data online;
- Remote control of your ESP32: by interacting with IoT platforms like Adafruit IO, Node-RED, and others—you can interact with your board remotely by exchanging data via HTTP requests;
- Interact with third-party services to send notifications: SMS, emails, notifications, and much more...

Technical Overview of HTTP Requests

Let's take a quick look at the technical aspects of HTTP requests. If this is your first time dealing with HTTP requests in a technical manner and it seems confusing, that's perfectly normal. But, don't worry, there are libraries that abstract most of the technical stuff, making it a bit more straightforward to make and handle HTTP requests. Additionally, for more information about the technical aspects of HTTP requests, we recommend the following resources:

- [Anatomy of an http request & response](#)
- [The HTTP Protocol \(documentation by IBM\)](#)

HTTP Requests

HTTP request consists of several parts. The main elements are: request line, headers, and body.

Request Line

The request line specifies the HTTP method, the resource being requested (the URL), and the version of the HTTP protocol being used. Common HTTP methods include GET, POST, PUT, and DELETE. For example:

```
GET /path/to/resource HTTP/1.1
```

Headers

Headers provide information about the request. It can include the host to which the client is sending the request, the type of content being sent or accepted, the user-agent (identifies the client making the request), and more. For example:

```
Host: example.com  
User-Agent: ESP32  
Accept: application/json
```

Body

The body is an optional content, and it contains data or information that we want to send to the server. This is used in POST and PUT requests. For example:

```
POST /submit-form HTTP/1.1  
Content-Type: application/x-www-form-urlencoded  
username=johndoe&password=secretpassword
```

HTTP Request Methods

The choice of HTTP method in a request indicates the intended action to be performed on the server. Some commonly used methods include:

- **GET:** retrieve data from the server.
- **POST:** submit data to be processed to a specified resource.
- **PUT:** update a resource on the server.
- **DELETE:** remove a resource from the server.

We'll mainly use GET requests with the ESP32.

HTTP Status Codes

After receiving an HTTP request, the server responds with an HTTP status code indicating the outcome of the request. Status codes fall into categories such as 2xx (successful), 3xx (redirection), 4xx (client error), and 5xx (server error). Some of the most common:

- **200 OK:** the request was successful.
- **404 Not Found:** the requested resource could not be found.

HTTP Requests with the ESP32

One of the easiest and most common ways to make HTTP requests with the ESP32 is using the `HTTPClient.h` library.

The following code is a basic example of how you can make a simple HTTP request to Google Website using the `HTTPClient` library.

- [Click here to download the code.](#)

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

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

// The URL of the webpage/API/server
const char* url = "http://www.google.com";

void setup() {
    // Start the serial communication
    Serial.begin(115200);

    // Connect to WiFi
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi...");
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.print(".");
    }
    Serial.println("\nConnected to WiFi");

    // Create an HTTPClient object
    HTTPClient http;

    // Begin the HTTP request to the specified URL
    http.begin(url);

    // Send the HTTP GET request
    int httpResponseCode = http.GET();

    // Check the response code to see if the request was successful
    if (httpResponseCode > 0) {
        // If successful, print the HTTP response code and the response payload
        // (the content of the page)
        Serial.println("HTTP Response code: " + String(httpResponseCode));
        String payload = http.getString();
        Serial.println("Response:");
        Serial.println(payload);
    } else {
        // If there was an error, print the error code
        Serial.println("Error in HTTP request, code: " + String(httpResponseCode));
    }
}
```

```
// End the HTTP request
http.end();
}

void loop() {
```

How Does the Code Work?

Let's take a quick look at the code to see how it works.

Including Libraries

First, include the `WiFi.h` and `HTTPClient.h` libraries.

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

Network Credentials

Insert your network credentials on the following lines so that the ESP32 can connect to your local network.

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

Request URL

Create a variable to hold the URL where you want to make the request. In this case, we'll create a request to google webpage.

```
const char* url = "http://www.google.com";
```

Connecting to Wi-Fi

Then, in the `setup()`, connect to Wi-Fi.

```
// Connect to WiFi
WiFi.begin(ssid, password);
Serial.print("Connecting to WiFi...");
while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
}
Serial.println("\nConnected to WiFi");
```

HTTP Request

Create an `HTTPClient` object called `http`.

```
HTTPClient http;
```

The following line prepares the `HTTPClient` to send a request to the specified URL.

```
http.begin(url);
```

Finally, send a GET request by using the `GET()` method on the `http` client object.

```
int httpResponseCode = http.GET();
```

This will return an HTTP response code. If the code is positive, the request was successful (when it returns an error code, it is negative). We save the response code on the `httpResponseCode` variable.

If we have a successful request, we can read the response—`http.getString()` retrieves the content of the response and stores it in the `payload` variable.

```
// Check the response code to see if the request was successful
if (httpResponseCode == 200) {
    // If successful, print the HTTP response code and the response payload
    // the content of the page
    Serial.println("HTTP Response code: " + String(httpResponseCode));
    String payload = http.getString();
    Serial.println("Response:");
    Serial.println(payload);
```

If the request fails, it prints the error code:

```
Serial.println("Error in HTTP request, code: " + String(httpResponseCode));
```

Finally, `http.end()` closes the connection and frees up resources.

```
http.end();
```

Demonstration

After inserting your network credentials in the code, upload it to your board. After uploading, open the Serial Monitor at a baud rate of 115200.

If everything goes as expected, the response code should be 200, and it will print the content of the Google web page.

Output Serial Monitor x

Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'COM3') New Line 115200 baud

```
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Connecting to WiFi.....
Connected to WiFi
HTTP Response code: 200
Response:
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="pt-PT"><head><me
var h=this||self;function l(){return window.google!==void 0&&window.google.kOPI!==void 0&&win
function t(a,b,c,d,k){var e="";b.search("&ei")===-1&&(e="&ei="+p(d),b.search("&lei")===-1&
document.documentElement.addEventListener("submit",function(b){var a;if(a=b.target){var c=a.g
</style><style>body,td,a,p,.h{font-family:arial,sans-serif}body{margin:0;overflow-y:scroll}#g
var g=this||self;var k,l=(k=g.mei)!null?k:1,n,p=(n=g.sdo)!null?n:!0,q=0,r,t=google.er.d,v=t.
"&bver)+"+b(t.bv);t.dpf&&(c+="&dpf)+"+b(t.dpf));var f=a.lineNumber;f!==void 0&&(c+="&line="+f);
if (!iesg){document.f&&document.f.q.focus();document.gbqf&&document.gbqf.q.focus();}
}

})();</script><div id="mngb"><div id=gbar><nobr><b class=gb1>Pesquisa</b> <a class=gb1 href=
else top.location='/doodles/'};});});</script><input value="AL9hbhdgAAAAAZseCbKm1QcyY1Yd-ZRTGR
if(a&&b&&(a!=google.cdo.width||b!=google.cdo.height)){var e=google,f=e.log,g="/client_204?&t
var f=this||self,g=function(a){return a};var h;var k={};l=function(a){this.g=a};l.prototype.t
var n=/^s*(?!javascript:)(?:[^\w+-]+|:[^:/?#]*(?:[/?#]|$))/i;var p="alternate author bookmark
function z(){var a=[u];if(!google.dp){for(var b=0;b<a.length;b++){var c=r("LINK"),d=c,e=t[a[b
var e=this||self;var g,h;a:(for(var k=["CLOSURE_FLAGS"],l=e,n=0;n<k.length;n++)if(l==k[n]),l
a.closest(["data-ved"])))?D(f)||"";"":f=f||"";if(a.hasAttribute("jsname"))a=a.getAttribute("ja
```

Wrapping Up

In this Unit, you learned the basics of HTTP GET requests and we create a simple example to demonstrate how to program the ESP32 to make a simple HTTP GET request. It can be extended to access APIs or other web services, just by changing the URL and handling the response appropriately.

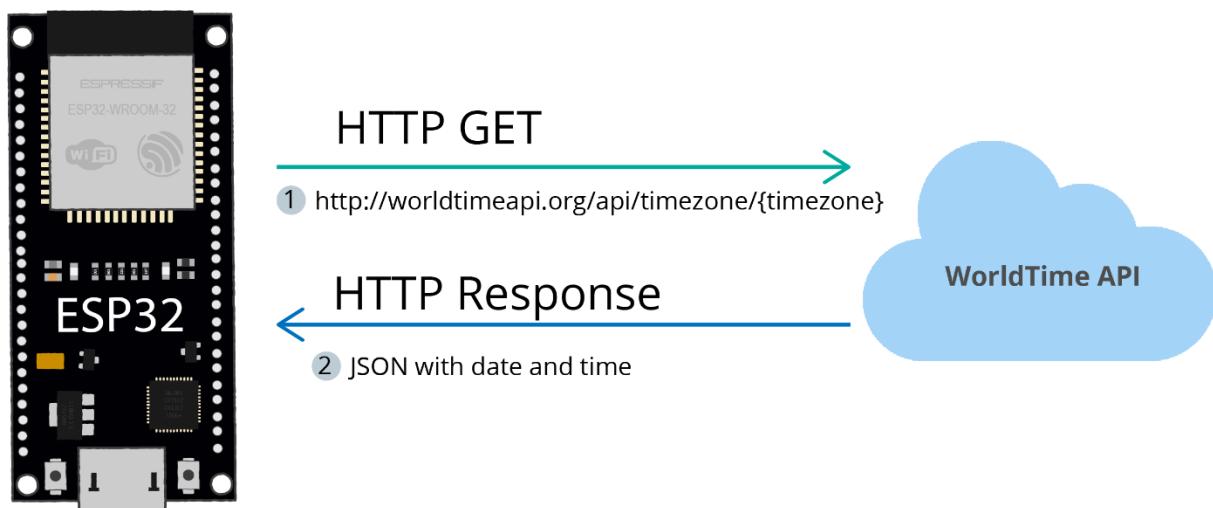
In the next Unit, we'll see how to request data from an API with the ESP32.

5.3 - Making HTTP Requests (WorldTime API and ThingSpeak)

In this Unit, you'll learn how to get the date and time for a specific time zone using the WorldTime API. This API retrieves the date and time adjusted for a particular time zone, including daylight saving time. With this example, you'll see another application of HTTP GET requests to get useful data. Later in this tutorial, we'll also take a look at how to send data to the ThingSpeak IoT platform also using HTTP GET requests.

The WorldTime API

The [WorldTime API](#) provides a simple HTTP interface to retrieve the current time, time zone information, and more.



Here's an example of how you can use the WorldTime API. Make a request to the following URL endpoint:

`http://worldtimeapi.org/api/timezone/{timezone}`

Replace **{timezone}** with the desired time zone. For example, to get the current time in UTC, you would make a request to:

`http://worldtimeapi.org/api/timezone/UTC`

You can also specify a city or region within a time zone, such as:

<http://worldtimeapi.org/api/timezone/Europe/Lisbon>

You can find the whole list of time zones here: <https://worldtimeapi.org/timezones>

After making the request, it will send a JSON with a response with the following information:

```
{  
    "utc_offset": "+01:00",  
    "timezone": "Europe/Lisbon",  
    "day_of_week": 5,  
    "day_of_year": 236,  
    "datetime": "2024-08-23T14:58:57.085854+01:00",  
    "utc_datetime": "2024-08-23T13:58:57.085854+00:00",  
    "unixtime": 1724421537,  
    "raw_offset": 0,  
    "week_number": 34,  
    "dst": true,  
    "abbreviation": "WEST",  
    "dst_offset": 3600,  
    "dst_from": "2024-03-31T01:00:00+00:00",  
    "dst_until": "2024-10-27T01:00:00+00:00",  
    "client_ip": "2001:8a0:e3e2:3500:31aa:cd21:10bb:a0f3"  
}
```

The meaning of each of those parameters can be found in the following table:

abbreviation	type: string	the abbreviated name of the time zone
datetime	type: string	an ISO8601-valid string representing the current, local date/time
day_of_week	type: integer	current day number of the week. Sunday is 0.
day_of_year	type: integer	ordinal date of the current year
dst	type: bool	flag indicating whether the local time is in daylight savings
dst_from	type: string	an ISO8601-valid string representing the datetime when daylight savings started for this time zone
dst_offset	type: integer	the difference in seconds between the current local time and daylight-saving time for the location

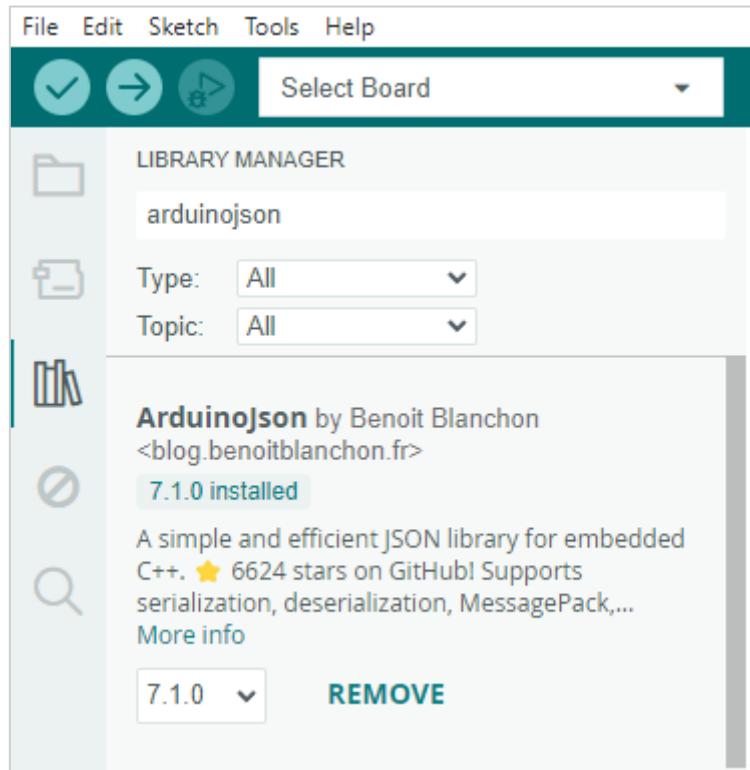
dst_until	type: string	an ISO8601-valid string representing the datetime when daylight savings will end for this time zone
raw_offset	type: integer	the difference in seconds between the current local time and the time in UTC, excluding any daylight-saving difference (see dst_offset)
timezone	type: string	time zone in Area/Location or Area/ Location /Region format
unixtime	type: integer	number of seconds since the Epoch
utc_datetime	type: string	an ISO8601-valid string representing the current date/time in UTC
utc_offset	type: string	an ISO8601-valid string representing the offset from UTC
week_number	type: integer	the current week number

<https://worldtimeapi.org/pages/schema>

Installing JSON Library

The response from the WorldTime API comes in JSON format. We'll use the ArduinoJson library to get the information we want from the JSON object.

To install the library, in the Arduino IDE go to **Sketch > Include Library > Manage Libraries** and search for ArduinoJson. Install the library by Benoit Blanchon. We're using library version 7.1.0.



Getting Date and Time - Code

The following code makes a request to the WorldTime API for a timezone of your location.

- [Click here to download the code.](#)

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

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

// Specify the timezone you want to get the time for:
// https://worldtimeapi.org/api/timezone/
const char* timezone = "Europe/Lisbon";
// API endpoint
String url = String("http://worldtimeapi.org/api/timezone/") + timezone;

// Store date and time
String currentDate;
String currentTime;

// Store hour, minute, second
int hour;
int minute;
int second;

// Store the result from the API request
String requestData = "failed";
```

```

String formatTime(int time) {
    return (time < 10) ? "0" + String(time) : String(time);
}

String getDate() {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;

        // Construct the request
        http.begin(url);
        // Make the GET request
        int httpCode = http.GET();
        if (httpCode > 0) {
            // Check for the response
            if (httpCode == HTTP_CODE_OK) {
                String payload = http.getString();
                Serial.println("Time information:");
                Serial.println(payload);
                // Parse the JSON to extract the time
                StaticJsonDocument<1024> doc;
                DeserializationError error = deserializeJson(doc, payload);
                if (!error) {
                    const char* datetime = doc["datetime"];
                    // Split the datetime into date and time
                    return String(datetime);
                } else {
                    Serial.print("deserializeJson() failed: ");
                    Serial.println(error.c_str());
                    return "failed";
                }
            }
        } else {
            Serial.printf("GET request failed, error: %s\n",
                          http.errorToString(httpCode).c_str());
            return "failed";
        }
        http.end(); // Close connection
    } else {
        Serial.println("Not connected to Wi-Fi");
        return "failed";
    }
}

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

    // Connect to Wi-Fi
    WiFi.begin(ssid, password);
    Serial.print("Connecting");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.print("\nConnected to Wi-Fi network with IP Address: ");
    Serial.println(WiFi.localIP());

    while(requestData == "failed") {
        requestData = getDate();
    }

    int splitIndex = requestData.indexOf('T');
}

```

```

// Extract date
currentDate = requestData.substring(0, splitIndex);
Serial.println("Current Date: " + currentDate);

// Extract time
currentTime = requestData.substring(splitIndex + 1, splitIndex + 9);
Serial.println("Current Time: " + currentTime);

// Extract hour, minute, second
hour = currentTime.substring(0, 2).toInt();
minute = currentTime.substring(3, 5).toInt();
second = currentTime.substring(6, 8).toInt();
Serial.println("Hour: " + String(formatTime(hour)));
Serial.println("Minute: " + String(formatTime(minute)));
Serial.println("Second: " + String(formatTime(second)));
}

void loop() {
}

```

How Does the Code Work?

First, include the required libraries.

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

Insert your network credentials in the following lines so that the ESP32 can connect to your network and get access to the internet.

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

Insert the timezone you want to get the data for.

```
const char* timezone = "Europe/Lisbon";
```

You can find a list of supported timezones in the link below:

- <https://worldtimeapi.org/timezones>

Then, create the API endpoint where you'll make the request taking into account the timezone.

```
String url = String("http://worldtimeapi.org/api/timezone/") + timezone;
```

Create String global variables to hold the date and time.

```
// Store date and time
```

```
String currentDate;
String currentTime;

// Store hour, minute, second
int hour;
int minute;
int second;
```

Create a variable to hold the response state. At first, we set it to `failed` because we haven't made a request yet.

```
String requestData = "failed";
```

The following function will format the time so that we add a zero at the left when the minutes, seconds or hours have just one digit.

```
String formatTime(int time) {
    return (time < 10) ? "0" + String(time) : String(time);
}
```

The `getDateTime()` function will return the date and time as a string.

```
String getDateTime() {
```

First, we check if we're connected to Wi-Fi before trying to make a request:

```
if (WiFi.status() == WL_CONNECTED) {
```

Then, we make the request to the URL specified previously.

```
HTTPClient http;

// Construct the request
http.begin(url);
// Make the GET request
int httpCode = http.GET();
```

If we get a good response, we'll deserialize the JSON object. The result is saved on the `doc` variable.

```
if (httpCode == HTTP_CODE_OK) {
    String payload = http.getString();
    Serial.println("Time information:");
    Serial.println(payload);
    // Parse the JSON to extract the time
    StaticJsonDocument<1024> doc;
    DeserializationError error = deserializeJson(doc, payload);
```

Then, it returns the date and time if everything goes as expected.

```
const char* datetime = doc["datetime"];
```

```
// Split the datetime into date and time
return String(datetime);
```

In the `setup()`, initialize the Serial Monitor and connect your board to Wi-Fi.

```
Serial.begin(115200);

// Connect to Wi-Fi
WiFi.begin(ssid, password);
Serial.print("Connecting");
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
}
Serial.print("\nConnected to Wi-Fi network with IP Address: ");
Serial.println(WiFi.localIP());
```

Then, call the `getDateTime()` function until we succeed.

```
while(requestData == "failed") {
    requestData = getDateTime();
}
```

The API returns the date and time in the following format.

```
2024-08-23T15:21:07.760694+01:00
```

So, we must split the String to get date and time separately on the T character.

```
int splitIndex = requestData.indexOf('T');
// Extract date
currentDate = requestData.substring(0, splitIndex);
Serial.println("Current Date: " + currentDate);

// Extract time
currentTime = requestData.substring(splitIndex + 1, splitIndex + 9);
Serial.println("Current Time: " + currentTime);
```

Then, we can continue splitting the time to get the hour minute and second on separate variables.

```
// Extract hour, minute, second
hour = currentTime.substring(0, 2).toInt();
minute = currentTime.substring(3, 5).toInt();
second = currentTime.substring(6, 8).toInt();
```

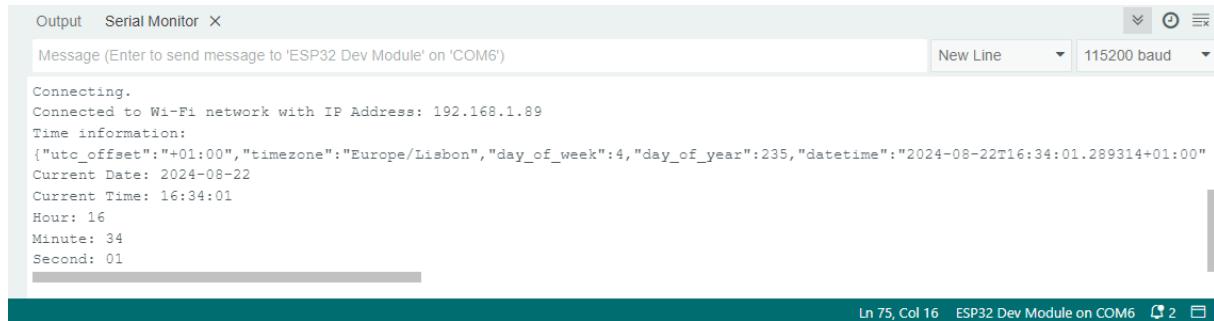
Finally, print the information to the Serial Monitor.

```
Serial.println("Hour: " + String(formatTime(hour)));
Serial.println("Minute: " + String(formatTime(minute)));
Serial.println("Second: " + String(formatTime(second)));
```

Testing the Code

After inserting your network credentials and desired timezone in the code, upload it to your board.

Open the Serial Monitor at a baud rate of 115200 and press the ESP32 RST button. It will display the JSON response and then the date and time in the desired format.



The screenshot shows the Arduino Serial Monitor window. The title bar says "Output Serial Monitor X". The message area contains the following text:

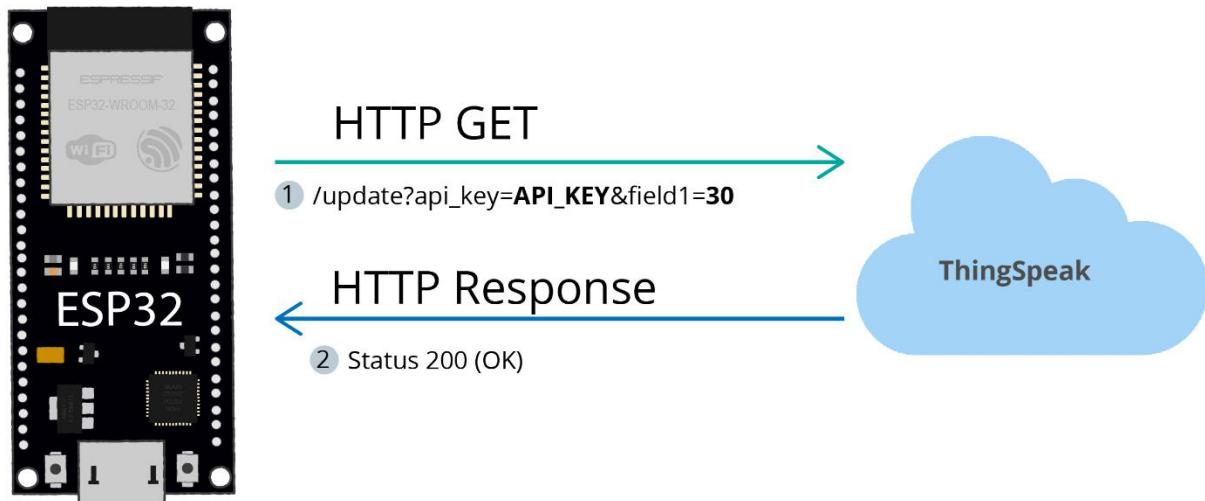
```
Message (Enter to send message to 'ESP32 Dev Module' on 'COM6')
Connecting.
Connected to Wi-Fi network with IP Address: 192.168.1.89
Time information:
{"utc_offset":"+01:00", "timezone":"Europe/Lisbon", "day_of_week":4, "day_of_year":235, "datetime":"2024-08-22T16:34:01.289314+01:00"}
Current Date: 2024-08-22
Current Time: 16:34:01
Hour: 16
Minute: 34
Second: 01
```

The status bar at the bottom right shows "Ln 75, Col 16 ESP32 Dev Module on COM6" and a small icon.

2. ESP32 HTTP GET: Update Value (ThingSpeak)

In this example, the ESP32 makes an HTTP GET request to send readings to ThingSpeak. This is an IoT platform that allows you to get real-time data collection and visualization from connected devices.

The data sent from the ESP32 can be visualized in charts on the ThingSpeak dashboard.



Using ThingSpeak API

ThingSpeak has a free API that allows you to store and retrieve data using HTTP. In this tutorial, you'll use the ThingSpeak API to publish and visualize data in charts from anywhere.

To use ThingSpeak with your ESP32, you need an API key. Follow the next steps:

- 1) Go to [ThingSpeak.com](https://thingspeak.com), create a free account, and sign in.
- 2) Then, open the [Channels](#) tab.
- 3) Create a New Channel.

The screenshot shows the 'My Channels' page of the ThingSpeak website. At the top, there's a navigation bar with links for 'Channels', 'Apps', and 'Support'. Below the navigation, the title 'My Channels' is displayed. To the right of the title is a search bar labeled 'Search by tag'. On the left, there's a green button with the text 'New Channel' enclosed in a red rectangular border. The main area below the title contains two columns: 'Name' and 'Created', which are currently empty.

- 4) Open your newly created channel and select the API Keys tab to copy your API Key.

The screenshot shows the 'Channel 220' page of the ThingSpeak website. The top navigation bar includes links for 'Channels', 'Apps', 'Support', 'Commercial Use', and 'How to Buy'. The 'API Keys' tab is highlighted with a red box. Below the navigation, the channel ID is listed as 'Channel ID: 220'. Under the 'API Keys' tab, there's a section titled 'Write API Key' with a text input field containing the value '7HQJM49R8JAPR'. A red box highlights this input field. Below the input field is a yellow button labeled 'Generate New Write API Key'. To the right of the 'Write API Key' section is a 'Help' section that explains what API keys are used for. Further down is an 'API Keys Settings' section with two bullet points: one about writing data and another about reading data.

Code ESP32 HTTP GET ThingSpeak

Copy the next sketch to your Arduino IDE. Don't forget to add your SSID, password, and API Key to the code:

- [Click here to download the code.](#)

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

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

// REPLACE WITH THINGSPEAK.COM API KEY
String serverName = "http://api.thingspeak.com/update?api_key=REPLACE_WITH_YOUR_API_KEY";
// EXAMPLE:
//String serverName = "http://api.thingspeak.com/update?api_key=7HQJM49R8JAP";

// THE DEFAULT TIMER IS SET TO 10 SECONDS FOR TESTING PURPOSES
// For a final application, check the API call limits per hour/minute
// to avoid getting blocked/banned
unsigned long lastTime = 0;
// Timer set to 10 minutes (600000)
//unsigned long timerDelay = 600000;
// Set timer to 10 seconds (10000)
unsigned long timerDelay = 10000;

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

    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: ");
    Serial.println(WiFi.localIP());

    Serial.println("Timer set to 10 seconds (timerDelay variable), it will take
10 seconds before publishing the first reading.");

    // Random seed is a number used to initialize a pseudorandom number generator
    randomSeed(analogRead(33));
}

void loop() {
    // Send an HTTP GET request
    if ((millis() - lastTime) > timerDelay) {
        // Check WiFi connection status
        if(WiFi.status()== WL_CONNECTED){
            WiFiClient client;
            HTTPClient http;

            String serverPath = serverName + "&field1=" + String(random(40));

            // Your Domain name with URL path or IP address with path
        }
    }
}
```

```
http.begin(client, serverPath.c_str());

// Send HTTP GET request
int httpResponseCode = http.GET();

if (httpResponseCode>0) {
    Serial.print("HTTP Response code: ");
    Serial.println(httpResponseCode);
    String payload = http.getString();
    Serial.println(payload);
}
else {
    Serial.print("Error code: ");
    Serial.println(httpResponseCode);
}
// Free resources
http.end();
}
else {
    Serial.println("WiFi Disconnected");
}
lastTime = millis();
}
```

Setting your network credentials

Modify the next lines with your network credentials: SSID and password.

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

Setting your serverName (API Key)

Modify the `serverName` variable to include your API key.

```
// REPLACE WITH THINGSPEAK.COM API KEY
String serverName = "http://api.thingspeak.com/update?api_key=REPLACE_WITH_YOUR_API_KEY";
```

Now, upload the code to your board and it should work straight away. Read the next section, if you want to learn how to make the HTTP GET request—it's quite similar to what we've seen in the previous examples.

HTTP GET Request

In the `loop()` is where you make the HTTP GET request every 10 seconds with random values:

```
String serverPath = serverName + "&field1=" + String(random(40));

// Your Domain name with URL path or IP address with path
http.begin(client, serverPath.c_str());
// Send HTTP GET request
int httpResponseCode = http.GET();
```

For example, to send a new value to ThingSpeak, the ESP32 needs to make a new request in the following URL—in this case, it updates the sensor **field1** with a new value (30).

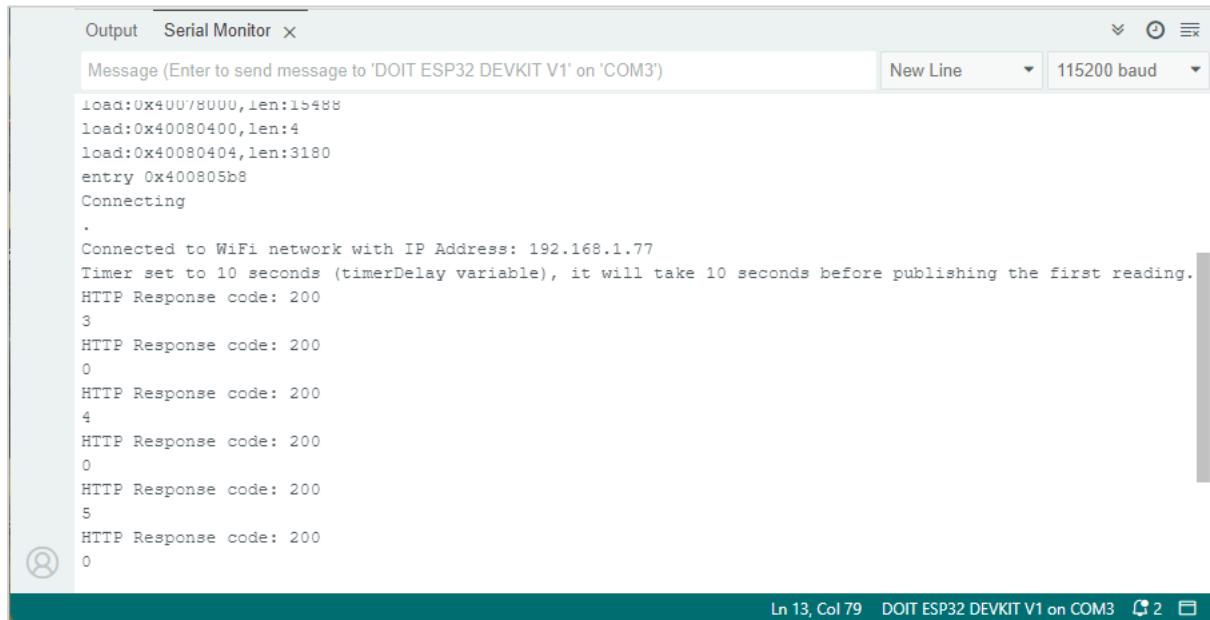
`http://api.thingspeak.com/update?api_key=REPLACE_WITH_YOUR_API_KEY&field1=30`

In our case, we're sending random values, but you can modify the code to send sensor readings instead.

Then, the following lines of code save and print the HTTP response from the server.

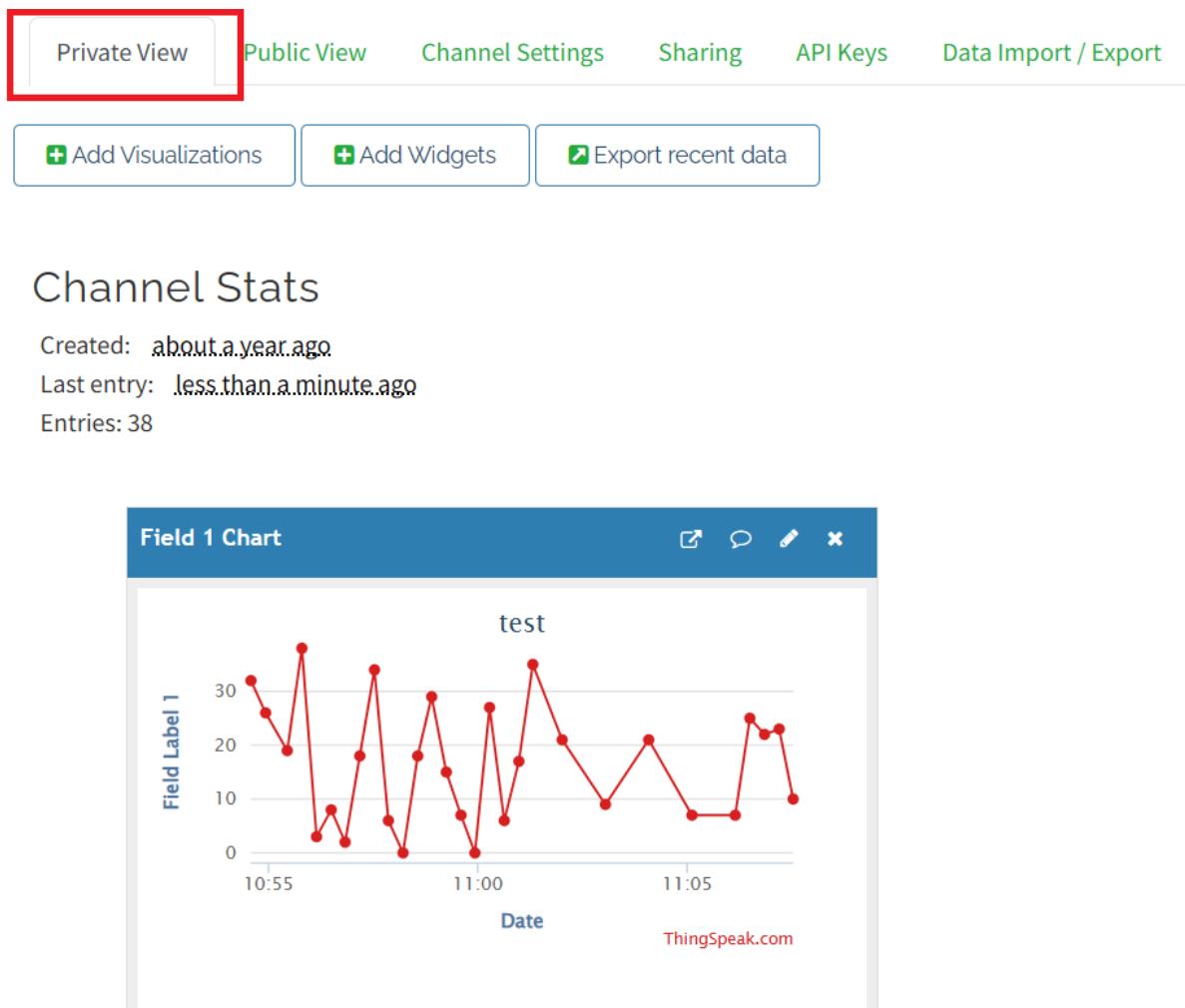
```
if (httpResponseCode>0) {
    Serial.print("HTTP Response code: ");
    Serial.println(httpResponseCode);
    String payload = http.getString();
    Serial.println(payload);
}
else {
    Serial.print("Error code: ");
    Serial.println(httpResponseCode);
}
```

In the Arduino IDE serial monitor, you should see an HTTP response code of 200 (this means that the request has succeeded).



In your ThingSpeak Channel, click on the **Private View** tab. Add a new Visualization. Select the Field 1 Chart.

You should see a chart with readings sent from the ESP32. New readings are added to the chart every second.

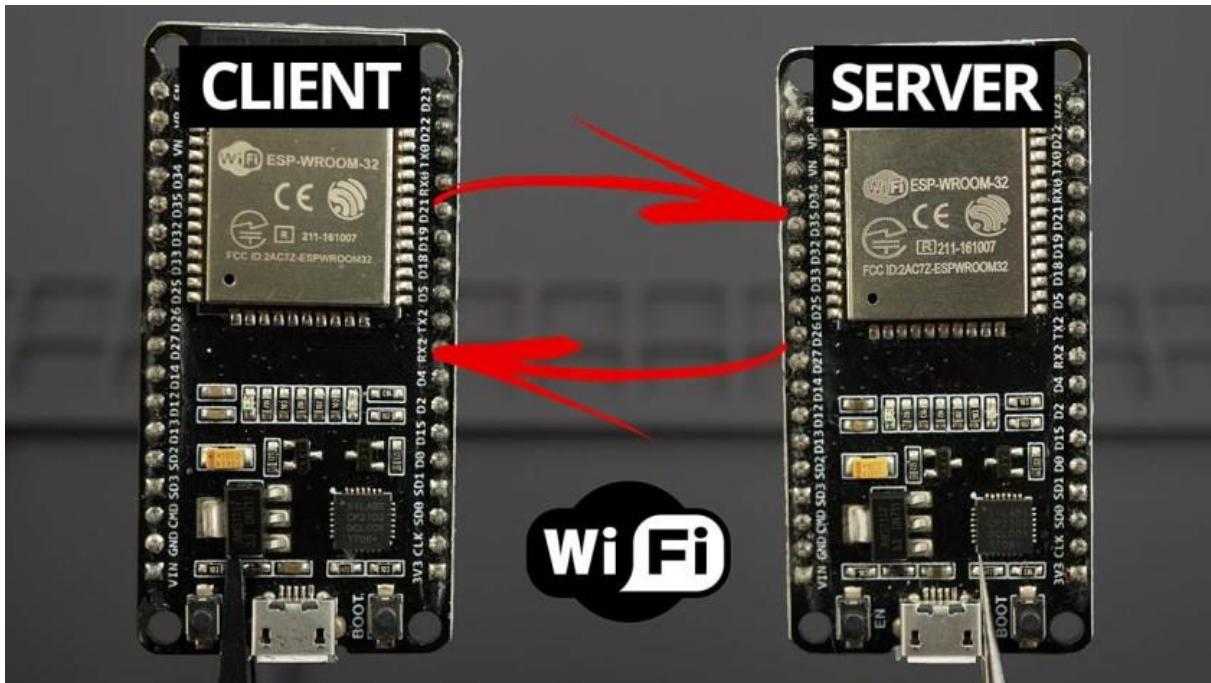


Now, you can modify this project to send sensor readings or other useful data that you want to see displayed on the chart.

For a final application, you might need to increase the timer or check the API call limits per hour/minute to avoid getting blocked/banned.

5.4 - ESP32 Client-Server Wi-Fi Communication Between Two Boards

In this unit, you'll learn how to set up 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.



Project Overview

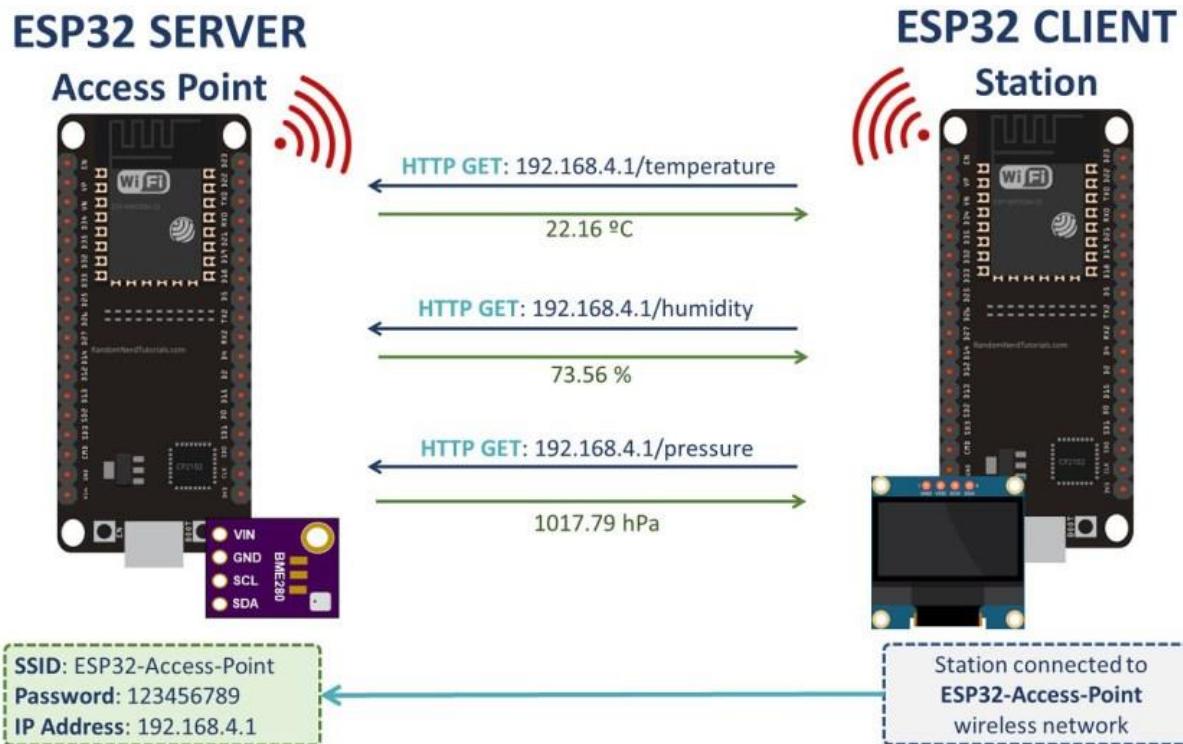
One ESP32 board will act as a server, and the other ESP32 board will act as a client.

Here's an overview of how things work:

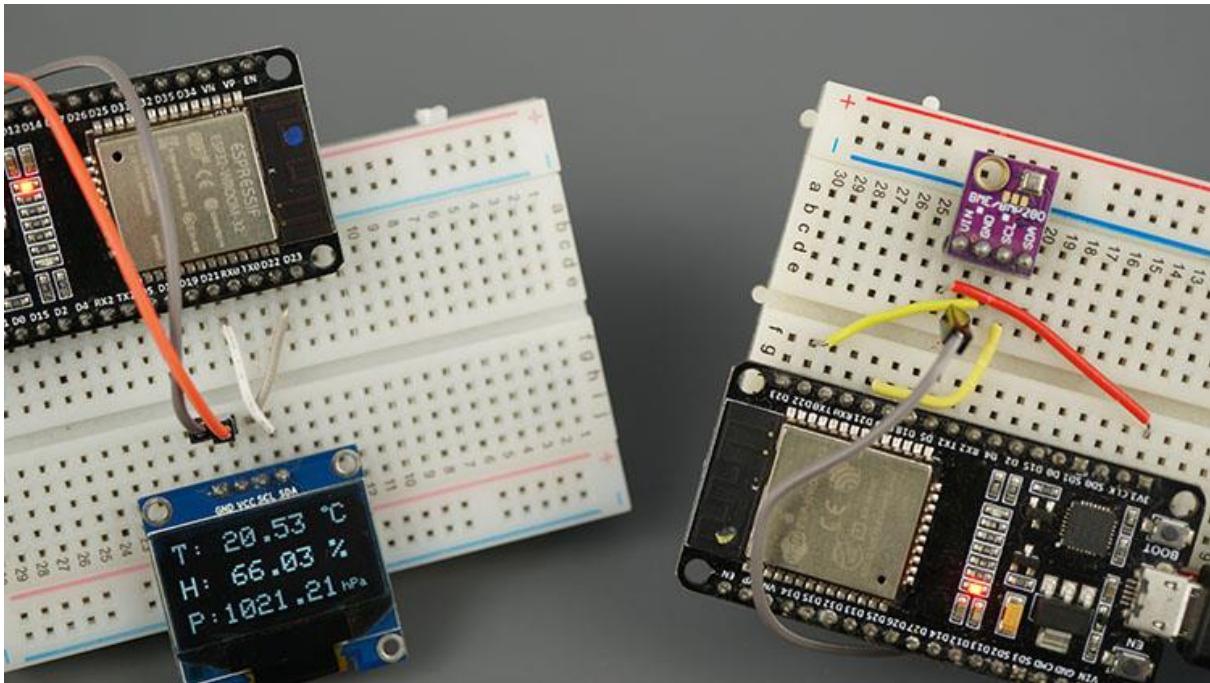
- The ESP32 server creates its own wireless network (ESP32 Soft-Access Point). So, other Wi-Fi devices can connect to that network to make requests (SSID: **ESP32-Access-Point**, Password: **123456789**).
- The ESP32 client is set as a station. So, it can connect to the ESP32 server wireless network.
- 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. In our example, it will respond to the following requests: /temperature, /humidity and /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](#)
- [BME280 sensor](#)
- [I2C SSD1306 OLED display](#)
- [Jumper Wires](#)
- [Breadboard](#)

Installing Libraries

For this tutorial, you need to install the following libraries:

BME280 Libraries

Install the following libraries to interface with the BME280 sensor. These libraries can be installed through the Arduino Library Manager. Go to **Sketch > Include Library > Manage Libraries** and search for the library name.

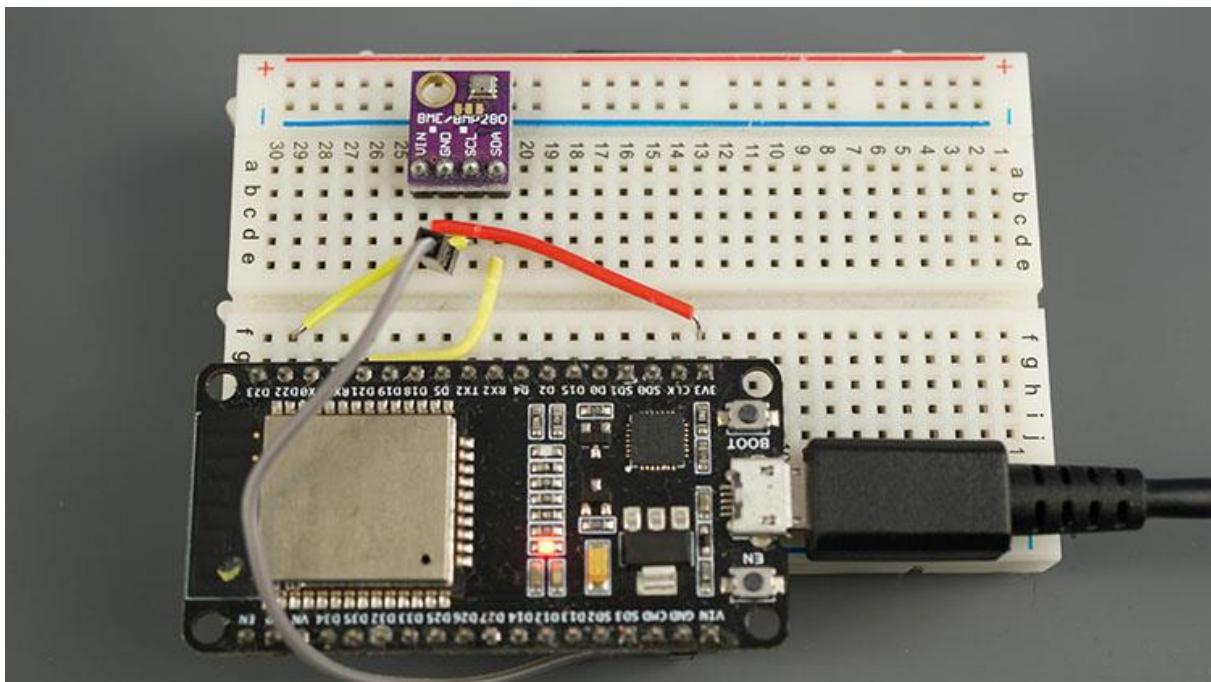
- [Adafruit_BME280 library](#)
- [Adafruit unified sensor library](#)

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](#)

#1 ESP32 Server (Access Point)

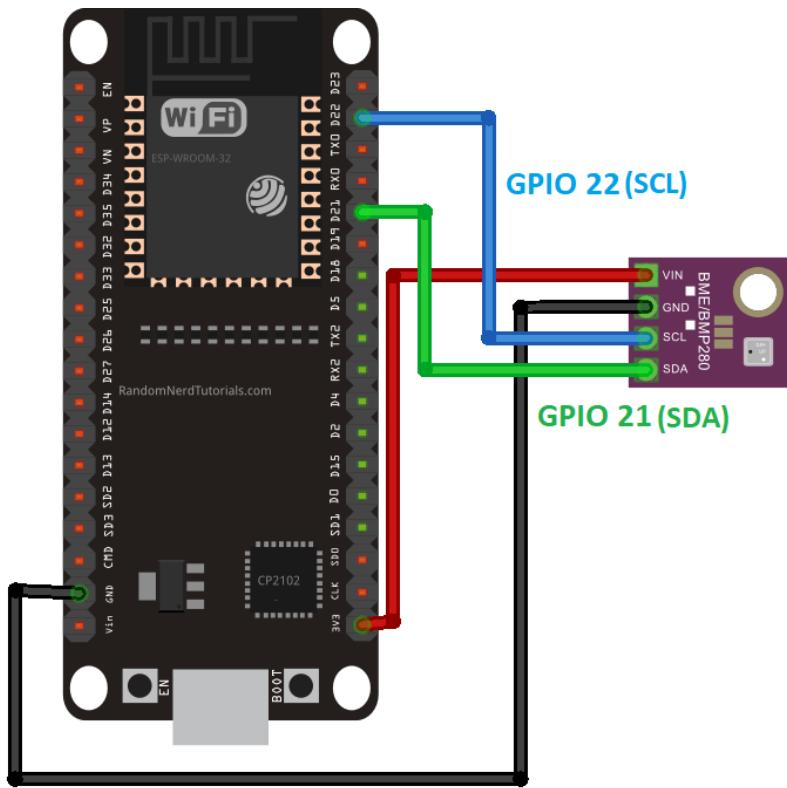


The ESP32 server is an Access Point (AP) that listens for requests on these paths `/temperature`, `/humidity` and `/pressure`. When it gets requests on those URLs, it sends the latest BME280 sensor readings.

For demonstration purposes, we're using a BME280 sensor, but you can use any other sensor by modifying a few lines of code.

Wiring the Circuit

Wire the ESP32 to the BME280 sensor as shown in the following schematic diagram.



You can use the following table as a reference when wiring the BME280 sensor.

BME280	ESP32
VIN	3.3V
GND	GND
SCL	GPIO 22
SDA	GPIO 21

ESP32 Server - Code

The following code sets up the ESP32 server with the BME280.

- [Click here to download the code.](#)

```
#include <WiFi.h>
#include <WebServer.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

// Access Point credentials
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";

Adafruit_BME280 bme; // I2C

String temperature;
String humidity;
String pressure;
```

```

// Create a web server object
WebServer server(80);

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);
}

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

    // Initialize BME280
    if (!bme.begin(0x76)) { // Change 0x76 to 0x77 if necessary
        Serial.println("Could not find BME280 sensor, check wiring!");
        while (1);
    }

    // Setting the ESP as an access point
    Serial.print("Setting AP (Access Point)...");
    WiFi.mode(WIFI_AP);
    // Remove the password parameter, if you want the AP (Access Point) to be open
    WiFi.softAP(ssid, password);
    IPAddress IP = WiFi.softAPIP();
    Serial.print("AP IP address: ");
    Serial.println(IP);

    // Set up the web server to respond with sensor data
    server.on("/temperature", HTTP_GET, []() {
        temperature = readTemp();
        String response = temperature;
        server.send(200, "text/plain", response);
    });
    server.on("/humidity", HTTP_GET, []() {
        humidity = readHumi();
        String response = humidity;
        server.send(200, "text/plain", response);
    });
    server.on("/pressure", HTTP_GET, []() {
        pressure = readPres();
        String response = pressure;
        server.send(200, "text/plain", response);
    });

    server.begin();
    Serial.println("HTTP server started");
}

void loop() {
    // Handle incoming client requests
    server.handleClient();
}

```

How Does the Code Works

Start by including the necessary libraries. Include the WiFi.h library and the WebServer.h library to set the ESP32 as a server and handle incoming HTTP requests. This WebServer.h library is included by default in the Arduino IDE when you install the ESP32 boards.

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

```
// Access Point 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`. This assumes we're connecting the BME280 to the ESP32 default I2C pins.

```
Adafruit_BME280 bme; // I2C
```

Create a web server on port 80.

```
// Create a web server object
WebServer 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);
```

Initialize the BME280 sensor:

```
// Initialize BME280
if (!bme.begin(0x76)) { // Change 0x76 to 0x77 if necessary
  Serial.println("Could not find BME280 sensor, check wiring!");
  while (1);
}
```

Set your ESP32 as an access point with the SSID name and password defined earlier.

```
// Setting the ESP as an access point
Serial.print("Setting AP (Access Point)...");
WiFi.mode(WIFI_AP);
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 response.

```
// Set up the web server to respond with sensor data
server.on("/temperature", HTTP_GET, []() {
  temperature = readTemp();
  String response = temperature;
  server.send(200, "text/plain", response);
});
```

The same happens when the ESP receives a request on the `/humidity` and `/pressure` URLs.

```
server.on("/humidity", HTTP_GET, []() {
  humidity = readHumi();
  String response = humidity;
  server.send(200, "text/plain", response);
});

server.on("/pressure", HTTP_GET, []() {
  pressure = readPres();
  String response = pressure;
  server.send(200, "text/plain", response);
});
```

Finally, start the server.

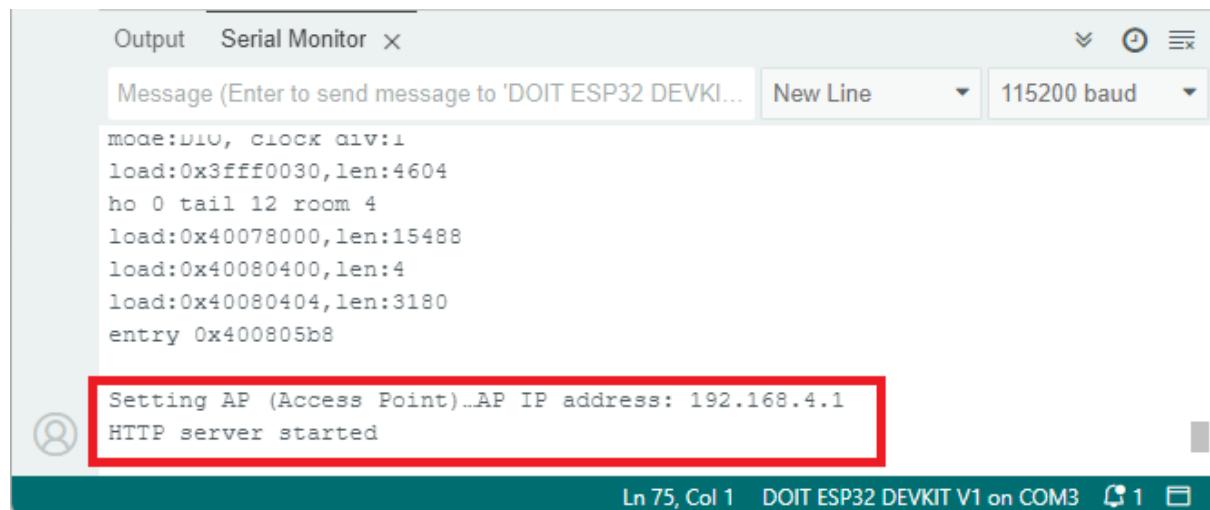
```
server.begin();
```

To keep the ESP32 listening and handling incoming requests, you need to add the following line to the `loop()`.

```
void loop() {
    // Handle incoming client requests
    server.handleClient();
}
```

Testing the ESP32 Server

Upload the code to your board and open the Serial Monitor. You should get something as follows:



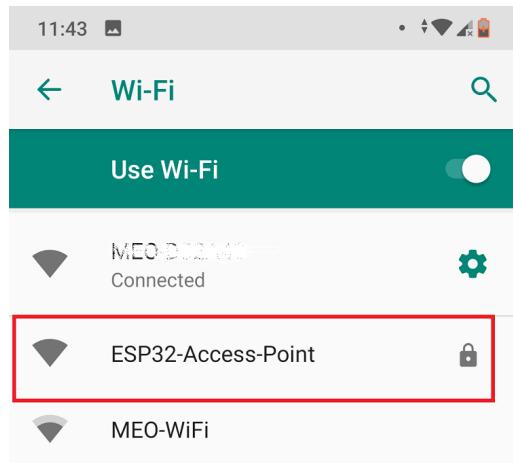
```
Output Serial Monitor X
Message (Enter to send message to 'DOIT ESP32 DEVKIT...' New Line 115200 baud
mode:DIO, clock div:1
load:0x3fff0030, len:4604
ho 0 tail 12 room 4
load:0x40078000, len:15488
load:0x40080400, len:4
load:0x40080404, len:3180
entry 0x400805b8

Setting AP (Access Point)...AP IP address: 192.168.4.1
HTTP server started
```

This means that the access point was set successfully.

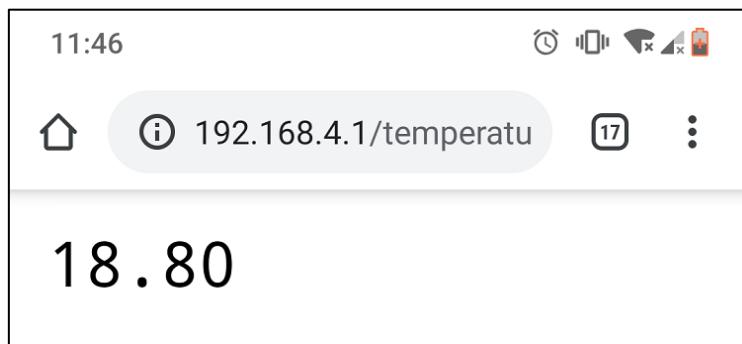
To make sure it is listening for temperature, humidity, and pressure requests, you need to connect to its network.

In your smartphone, go to the Wi-Fi settings and connect to the **ESP32-Access-Point**. The password is **123456789**.

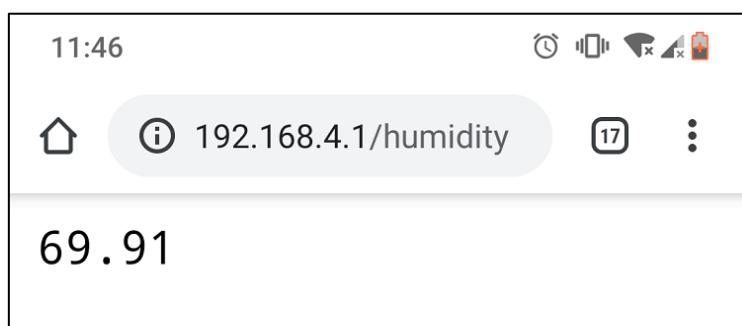


While connected to the access point, open your browser and type `192.168.4.1/temperature`.

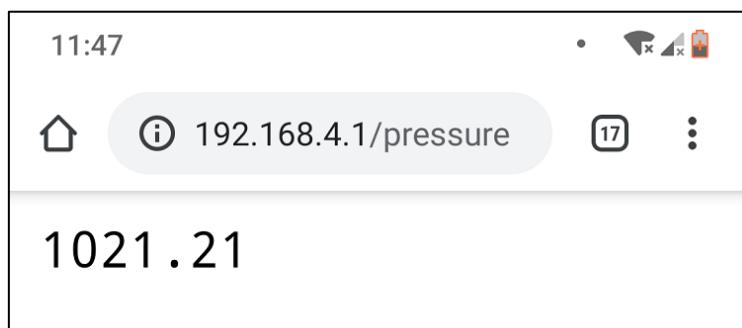
You should get the temperature value in your browser:



Try this URL path for the humidity `192.168.4.1/humidity`:

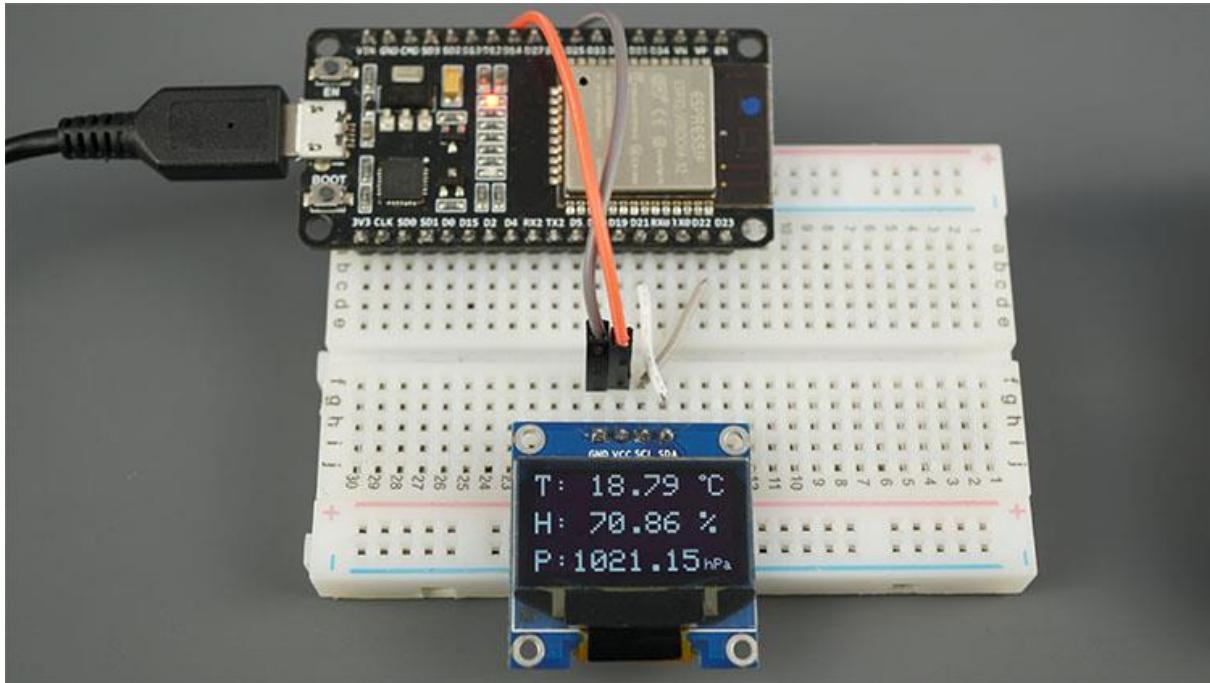


Finally, go to `192.168.4.1/pressure` URL:



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)

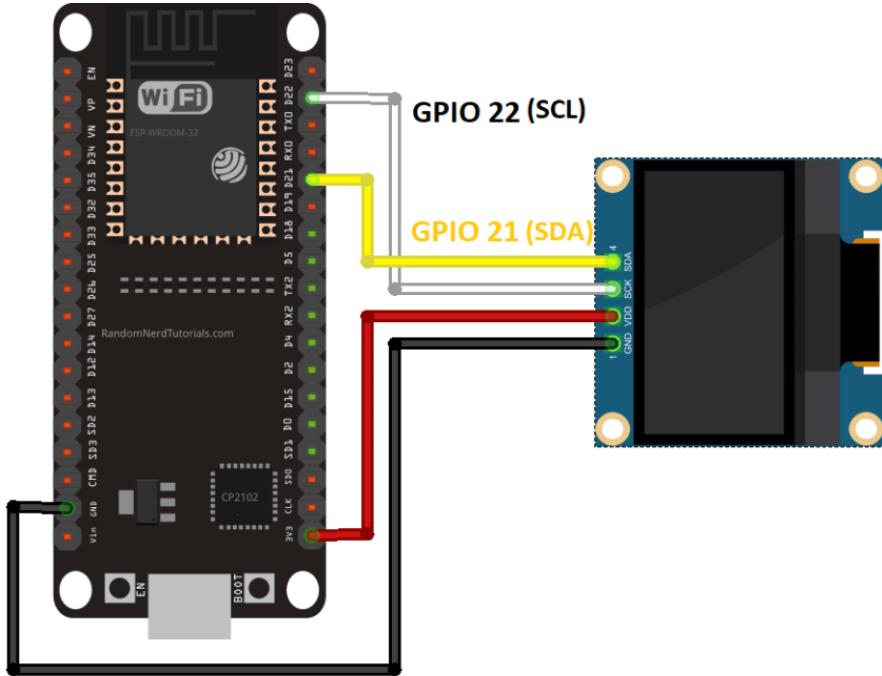


The ESP32 Client is a Wi-Fi station that is connected to the ESP32 Server. The client requests the temperature, humidity, and pressure from the server by making HTTP GET requests on the /temperature, /humidity, and /pressure URL routes. Then, it displays the readings on an OLED display.

Wiring the Circuit

Wire the ESP32 to the OLED display as shown in the following table or diagram.

OLED Display	ESP32
GND	GND
VCC	VIN
SCL	GPIO 22
SDA	GPIO 21



ESP32 Client - Code

Upload the following code to the other ESP32:

- [Click here to download the code.](#)

```
#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
#define SCREEN_HEIGHT 64 // OLED display height, in pixels

// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
#define OLED_RESET 4 // Reset pin # (or -1 if sharing Arduino reset pin)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

String temperature;
String humidity;
String pressure;

unsigned long previousMillis = 0;
const long interval = 5000;

void setup() {
```

```

Serial.begin(115200);

// Address 0x3C for 128x64, you might need to change this value
// use an I2C scanner
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println(F("SSD1306 allocation failed"));
    for(;;) // Don't proceed, loop forever
}
display.clearDisplay();
display.setTextColor(WHITE);

WiFi.begin(ssid, password);
Serial.println("Connecting");
while(WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
}
Serial.println("");
Serial.print("Connected to WiFi network.");
}

void loop() {
    unsigned long currentMillis = millis();

    if(currentMillis - previousMillis >= interval) {
        // Check WiFi connection status
        if(WiFi.status()== WL_CONNECTED ){
            temperature = httpGETRequest(serverNameTemp);
            humidity = httpGETRequest(serverNameHumi);
            pressure = httpGETRequest(serverNamePres);
            Serial.println("Temperature: " + temperature + " *C - Humidity: "
                           + humidity + " % - Pressure: " + pressure + " hPa");

            display.clearDisplay();

            // display temperature
            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");

            // display humidity
            display.setTextSize(2);
            display.setCursor(0, 25);
            display.print("H: ");
            display.print(humidity);
            display.print(" %");

            // display pressure
            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();

        // save the last HTTP GET Request
        previousMillis = currentMillis;
    }
    else {
        Serial.println("WiFi Disconnected");
    }
}

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;
}

```

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 <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_RESET);
```

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
// use an I2C scanner
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");
```

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: " + humidity +  
" % - Pressure: " + pressure + " hPa");
```

Then, display the temperature, humidity and pressure in the OLED display:

```
// display temperature  
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");  
  
// display humidity  
display.setTextSize(2);  
display.setCursor(0, 25);  
display.print("H: ");  
display.print(humidity);  
display.print(" %");  
  
// display pressure  
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 five seconds. That's why we have the `previousMillis`, `currentMillis` variables and use the `millis()` function.

Upload the sketch to #2 ESP32 (client) to test if everything is working properly.

Testing the ESP32 Client

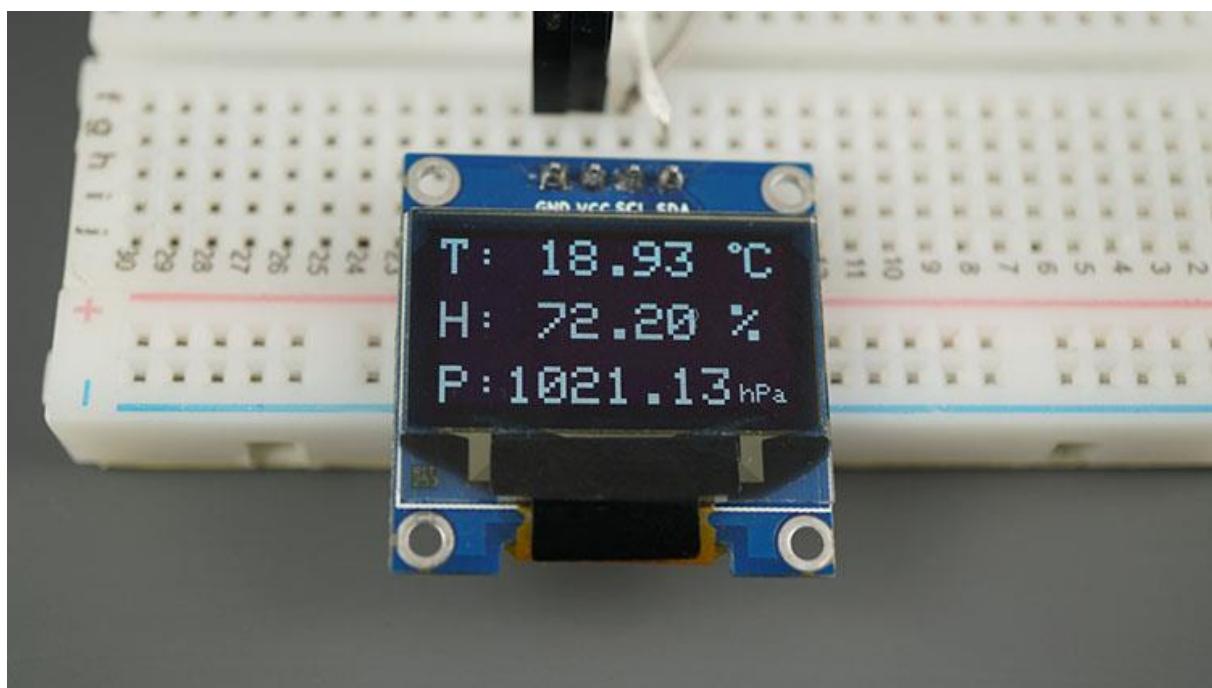
Having both boards fairly close and powered, you'll see that ESP32 client is making requests to the ESP32 server every five seconds. The ESP32 server sends the sensor readings as a response.

This is what you should see on the ESP32 Client Serial Monitor.

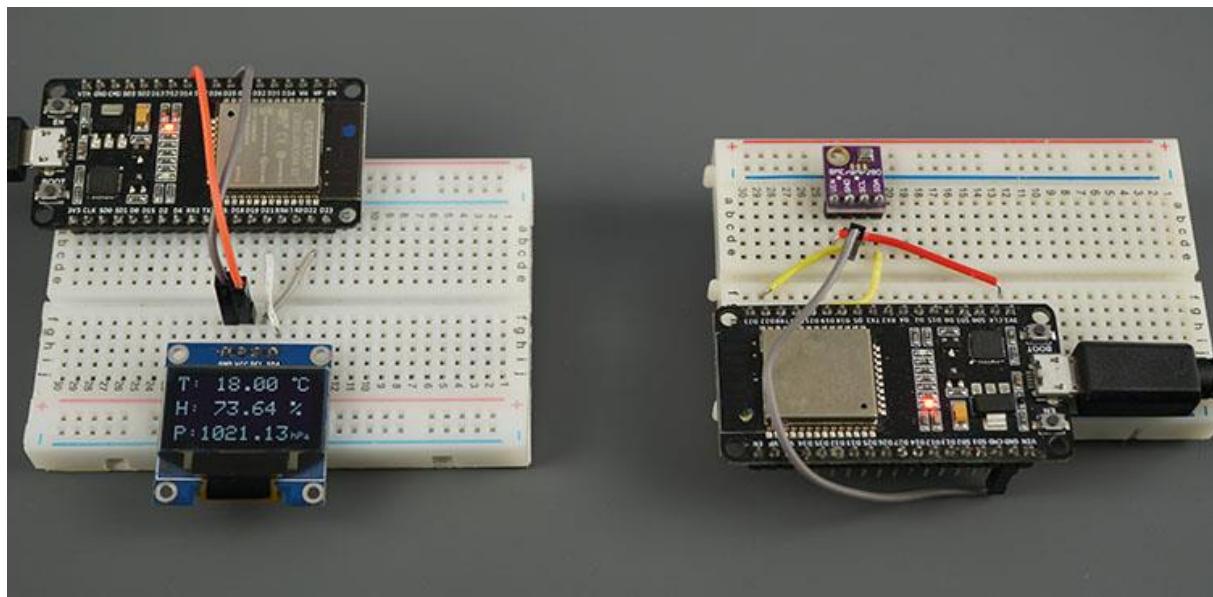
```
HTTP Response code: 200
Temperature: 35.52 °C - Humidity: 42.63 % - Pressure: 1003.65 hPa
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
Temperature: 35.51 °C - Humidity: 42.64 % - Pressure: 1003.69 hPa
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
Temperature: 35.52 °C - Humidity: 42.66 % - Pressure: 1003.68 hPa
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
Temperature: 35.50 °C - Humidity: 42.68 % - Pressure: 1003.69 hPa
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
Temperature: 35.49 °C - Humidity: 42.71 % - Pressure: 1003.68 hPa
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
```

Ln 98, Col 25 DOIT ESP32 DEVKIT V1 on COM5 [not connected]  

The sensor readings are also displayed in the OLED.



That's it! Your two boards are talking to each other via HTTP requests.



Wrapping Up

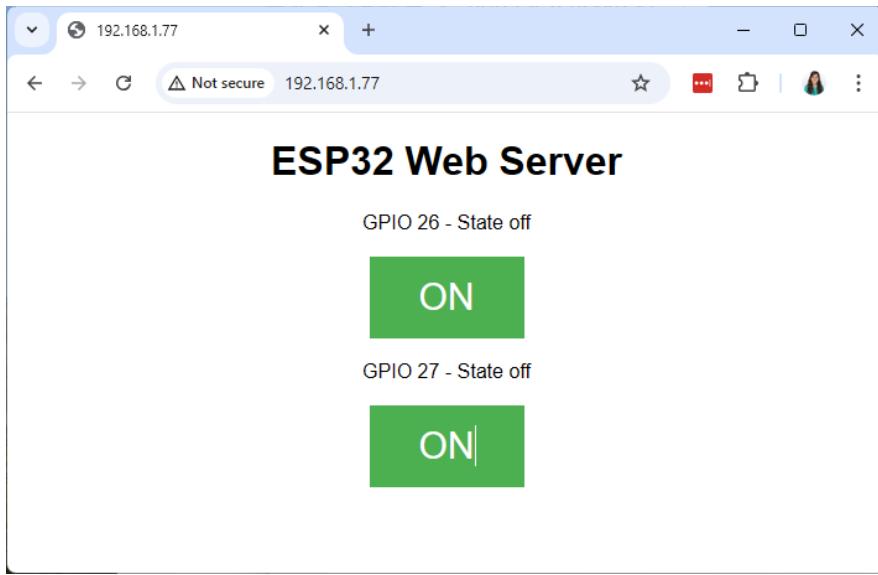
In this Unit, you've learned how to send data from one ESP32 to another via Wi-Fi using HTTP requests without the need to connect to the internet. For demonstration purposes, we've shown how to send BME280 sensor readings, but you can use any other sensor or send any other data.

MODULE 6

ESP32 Web Servers

6.1 - Web Server Introduction

In this section, we're going to take a look at the ESP32 as a web server. We'll create a web server to control outputs through the web, and we'll also show you how you can display sensor readings on a web page.



After creating the web server, we'll show you how you can add more outputs and sensor readings and how you can customize your web page using HTML and CSS. Finally, you'll protect your web server with a password, and you'll learn how to make your web server accessible from anywhere.

Introducing Web Servers

In simple terms, a web server is a “computer” that provides web pages. It stores the website’s files, including all HTML documents and related assets like images, CSS style sheets, fonts, and/or other files. It also brings those files to the user’s web browser device when the user makes a request to the server.

When you access a web page in your browser, you’re actually sending a request via Hypertext Transfer Protocol (HTTP) to a server. This is a process for requesting and returning information on the internet. The server sends back the web page you requested—also through HTTP.

To better understand how all of this works with your ESP32 boards, let's take a look at some terms that you've probably heard before, but you may not know exactly what they mean.

Request-response

Request-response is a message exchange pattern, in which a requestor (your browser) sends a request message to a replier system (the ESP32 as a web server) that receives and processes the request, and returns a message in response.

In most of our projects, the response message will be a web page that displays the latest sensor readings or that provides an interface to control outputs.

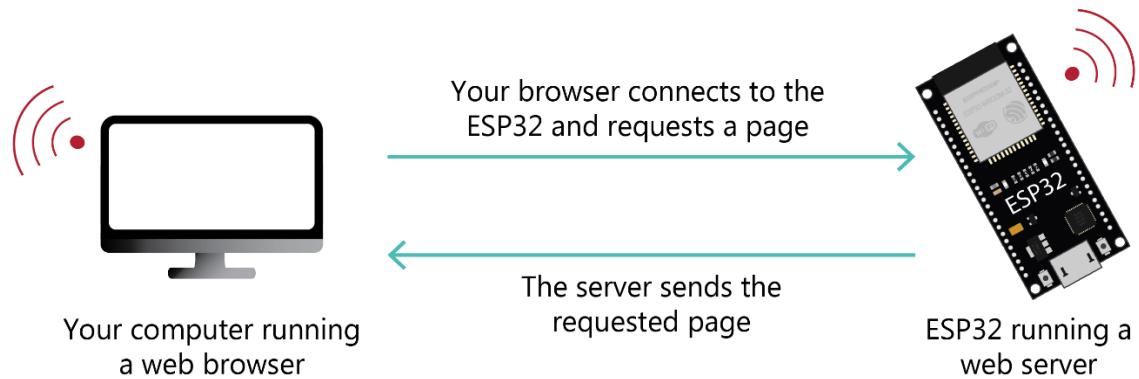


This is a simple, yet powerful messaging pattern, especially in client-server architectures.

Client-Server

When you type an URL in your browser, in the background, you (the client) send a request via Hypertext Transfer Protocol (HTTP) to a server. When the server receives the request, it sends a response also through HTTP, and you will see the web page requested in your browser. Clients and servers communicate over a computer network.

In simple terms, clients make requests to servers. Servers handle the clients' requests.



In this Module, the ESP32 is going to be the server, and you (your browser) are the client. Our projects only have one server (the ESP32 board), but can have multiple clients: multiple web browsers on the same network but on different devices like computers, smartphones or tablets, or multiple web browser tabs opened on the same device.

IP Address

An IP address is a numerical label assigned to each device connected to a computer network.

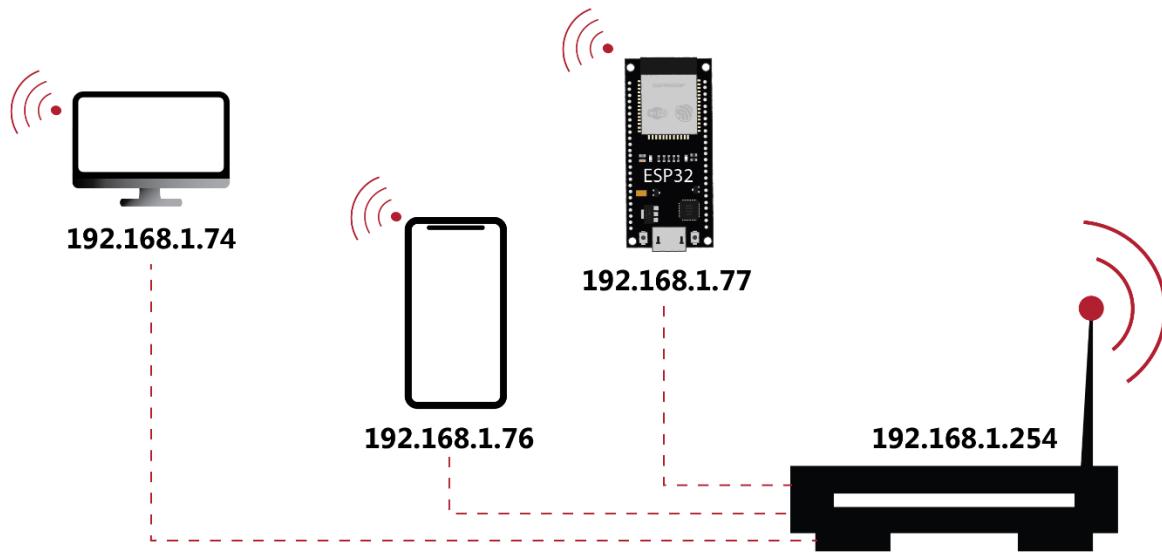
It is a series of four values separated by periods with each value ranging from 0 to 255. Here's an example:

192.168.1.75

At home, your devices are connected to a private network through your router (local network). All devices connected to your router are part of your local network. Inside this network, each device has its own IP address.

Devices that are connected to the same router can access each other via the IP address. Devices outside your local network can't access your local devices using their local IP address.

When you connect your ESP32 boards to your router, they become part of your local network. So, your ESP32 boards are assigned an IP address.



In your local network, the IP address of the ESP board (and other devices) is assigned by the router using DHCP (Dynamic Host Configuration Protocol). You don't need to worry about what DHCP is, you just need to know that it assigns an IP address and other network configuration parameters to each device on the network.

The router keeps track of every device on the network and maps an IP address to each device every time it joins the network. Two devices on the same network can't have the same IP address.

Again, when the ESP32 is connected to your router, the IP address assigned is a local address. This means you can only access it using other devices that are also connected to the same network. In the previous image, you can access the ESP32 board using the computer or the smartphone that are also connected to the same network.

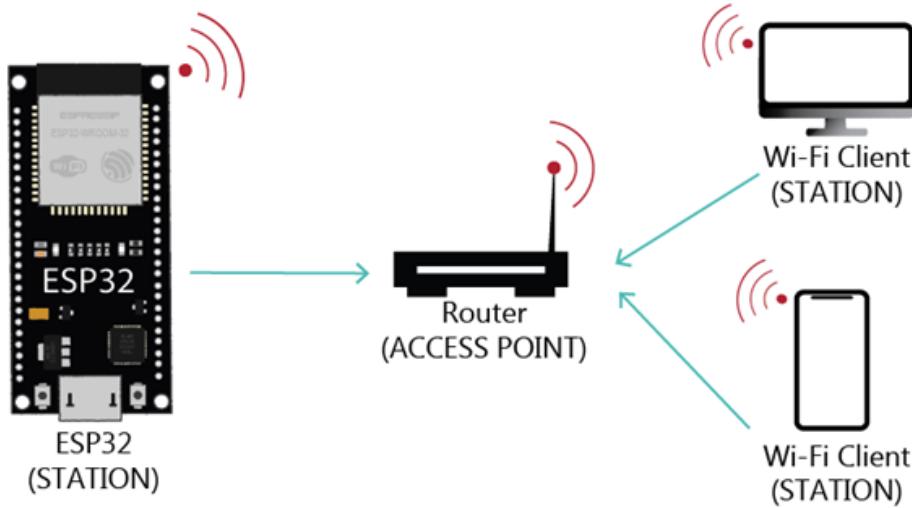
Wi-Fi Station and Wi-Fi Access Point

The ESP32 board can act as Wi-Fi Station, Access Point, or both as we've seen in the previous Module.

Wi-Fi Station

When the ESP32 is set as a Wi-Fi station, it can connect to other networks (like connecting to your router—local network). In this scenario, the router assigns a

unique IP address to your ESP board. You can communicate with the ESP32 using other devices (stations) that are also connected to the same network by referring to the ESP32's local IP address.

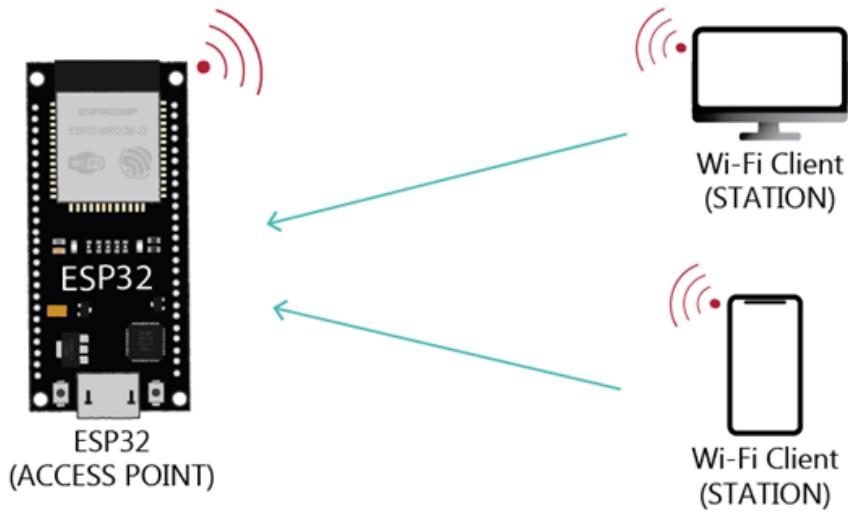


Since the router is also connected to the internet, we can request information from the internet using our ESP32 boards like data from APIs (like we did in the previous module), publish data to online platforms, use icons and images from the internet in our web server pages or include JavaScript libraries.

However, in some cases, we may not have a router nearby to connect the ESP32. In this scenario, you must set your ESP32 board as an access point.

Access Point

When your ESP32 is set up as an Access Point, other devices (such as your smartphone, tablet, or computer) can connect to it without the need for a router; the ESP controls its own Wi-Fi network.



Unlike a router, an ESP32 Access Point doesn't connect further to a wired network or the Internet, so you can't access external libraries, publish sensor readings to the cloud, or use services like mail. In most of our examples, we'll set the boards as stations. You can easily modify our examples and set the boards as access points instead if that's more suitable for your projects.

Client-Server Communication

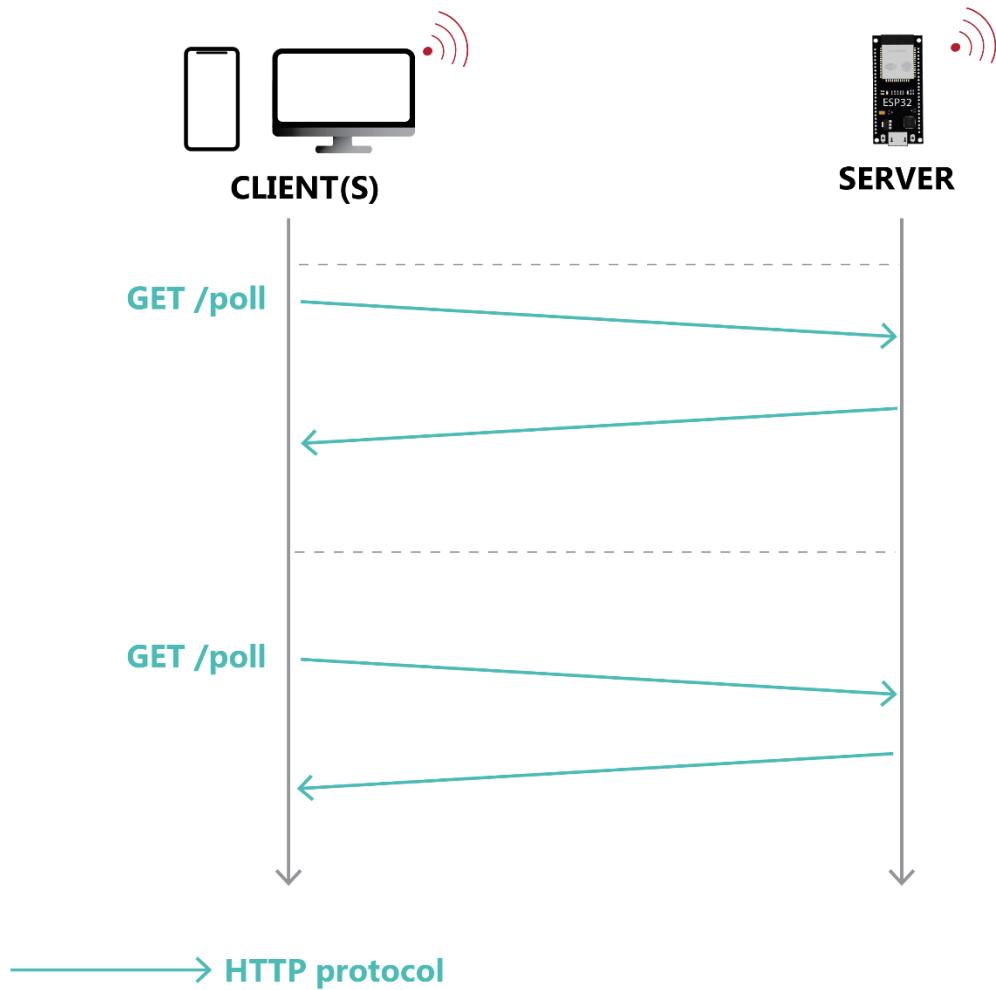
There are several ways to communicate between the client and the server: HTTP Polling, Server-Sent Events (SSE), and WebSocket. We'll focus on HTTP Polling, but we have an eBook dedicated to web servers that explores in great detail those three communication protocols:

- [Build Web Servers with ESP32 and ESP8266 eBook \(2nd Edition\)](#)

HTTP Polling

In HTTP polling, the client polls the server requesting new information. When the server receives a request, it sends a response back to the client. The server will only send information when requested by the client.

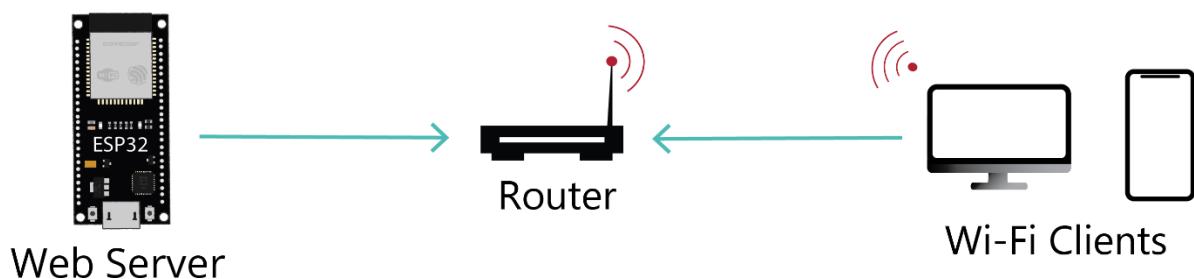
For example, you in your browser send a request to the ESP32, and it responds with whatever you want, for example, a web page with the latest sensor readings. Using this method, to keep your web page updated, you need to make new requests constantly.



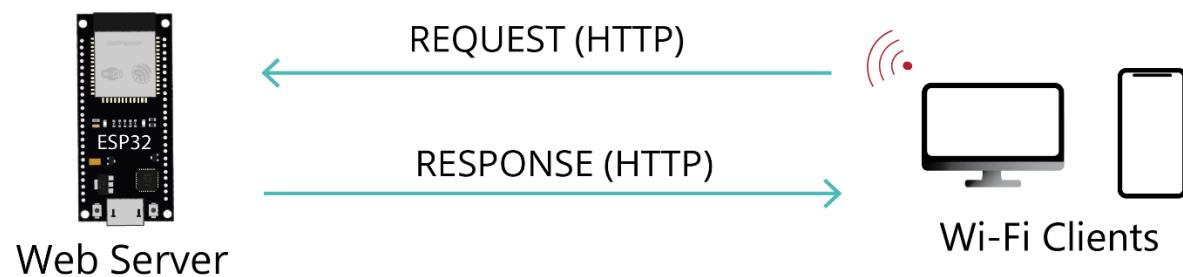
ESP32 Web Server

Let's take a look at a practical example with the ESP32 that acts as a web server in the local network.

Typically, a web server with the ESP32 in the local network looks like this: the ESP32 running as a web server is connected via Wi-Fi to your router. Your computer, smartphone, or tablet, are also connected to your router via Wi-Fi or Ethernet cable. So, the ESP32 and your browser are on the same network.



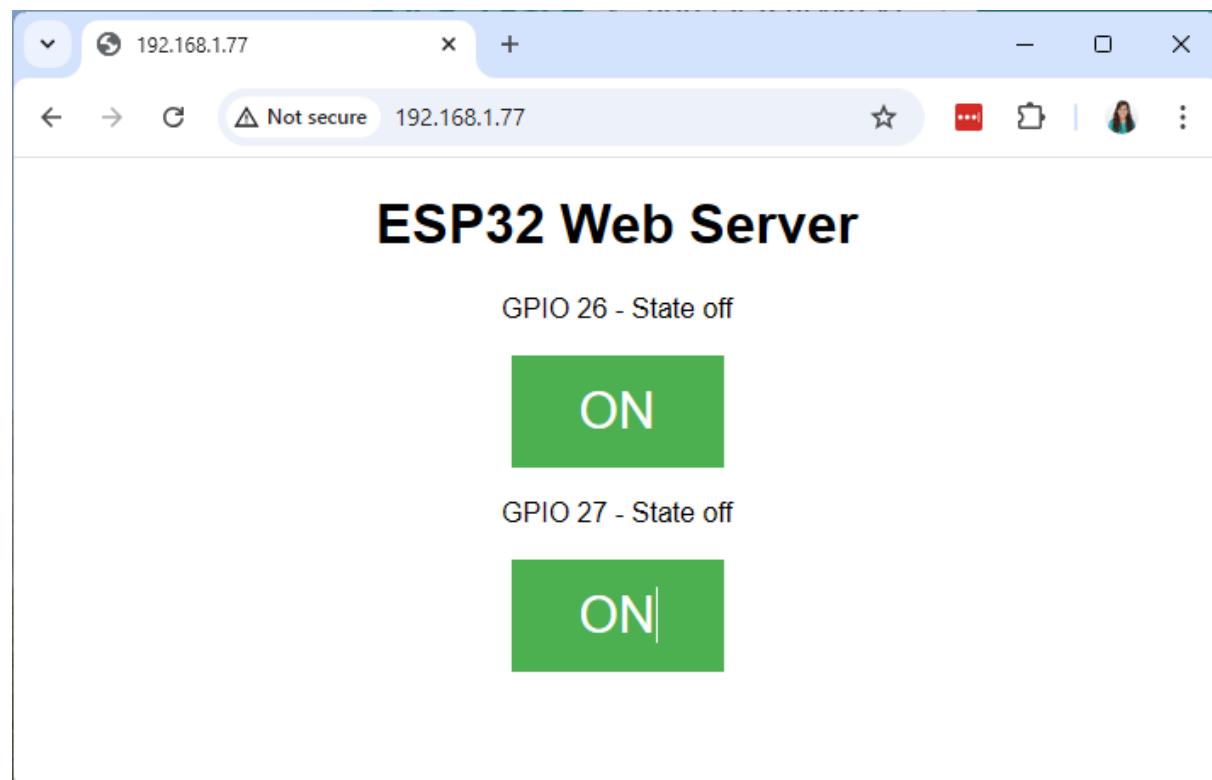
When you type the ESP32 IP address in your browser, you are sending an HTTP request to your ESP32. Then, the ESP32 responds with a response that can contain a value, a reading, HTML text to display a web page or any other data.



Web Server Example

So, how can you put all of this together to make IoT projects with your ESP32? Your ESP32 has GPIOs, so you can connect devices and control or monitor them through the web.

Here's an example of a web server we've built to control an output. The following web page shows up when you enter the ESP32 IP address in a browser.



When you press the ON button, the URL changes to the ESP IP address followed by /on. The ESP receives a request on that new URL, it checks which URL is being requested and changes the LED state accordingly.

- Press the ON button → request: /on → LED turns on

When you press the OFF button, a new request is made to the ESP32 in the /off URL. The ESP checks once again which URL is being requested and turns the LED off.

- Press the OFF button → request: /off → LED turns off

The same concept can be applied to control multiple outputs.

Wrapping Up

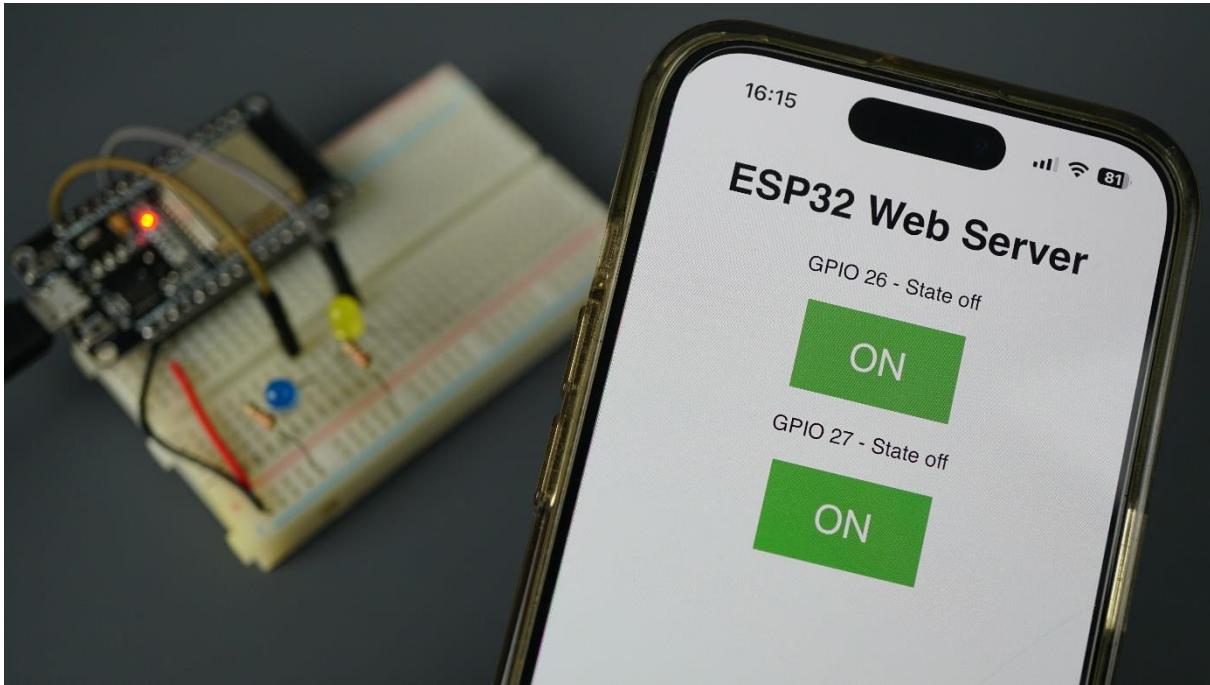
This was just an overview of how an ESP32 web server works.

In the next Unit, we'll take a look at the steps you need to follow to remotely control outputs through the web using your ESP32.

If you are interested in learning about web servers in greater detail, we have an eBook exclusively dedicated to that subject:

- [Build Web Servers with ESP32 and ESP8266 eBook](#)

6.2 - ESP32 Web Server: Control Outputs



In this Unit, you will learn how to create a simple web server with the ESP32 to control outputs. The web server you'll build can be accessed with any device with a browser: smartphone, tablet, or laptop, on the local network.

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 and understand the steps later on.

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

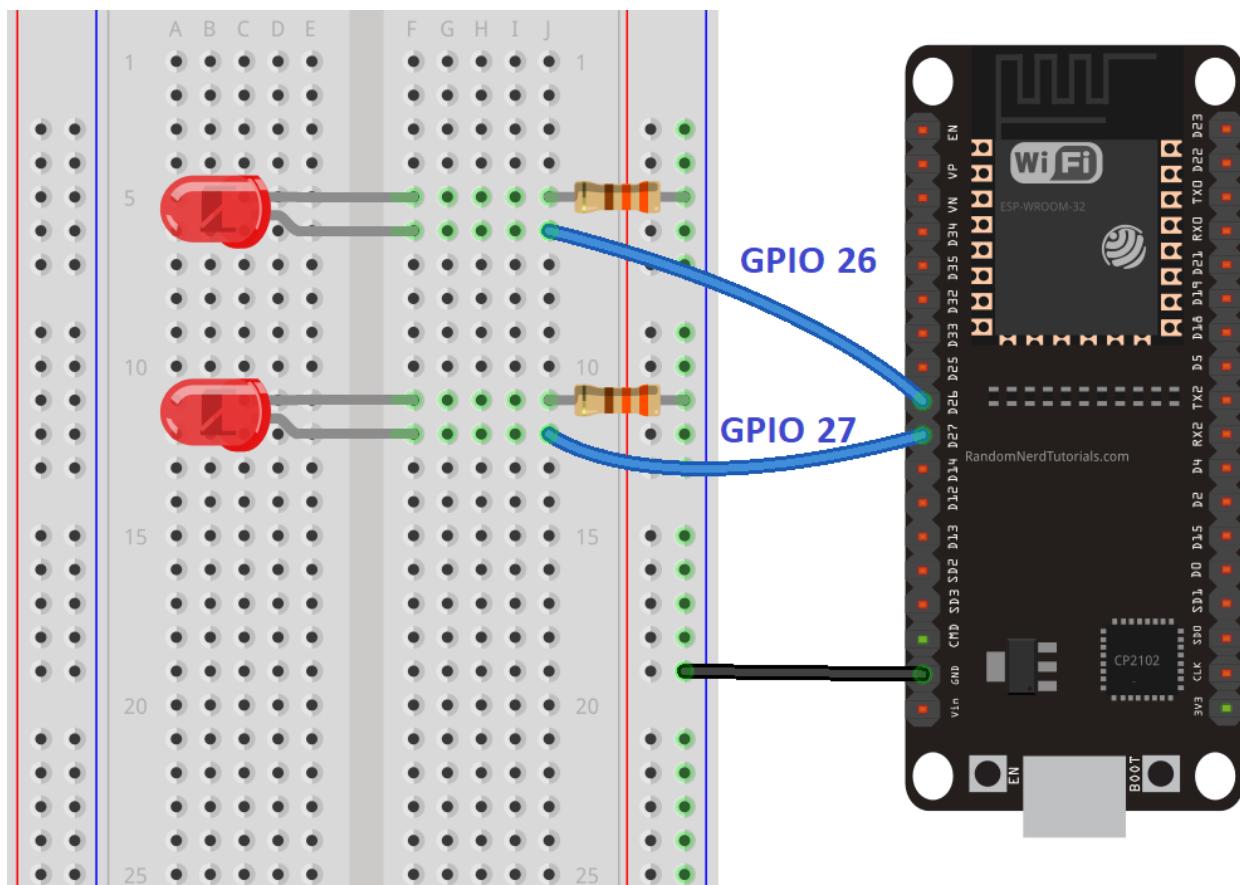
This is a simple example that illustrates how to build a web server that controls two LEDs. The idea is to replace those LEDs with a [relay](#), or any other electronic components you want to control.

Wiring the Circuit

Start by building the circuit. Connect two LEDs to your ESP32 as shown in the following schematic diagram—with one LED connected to GPIO 26 and another to GPIO 27.

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [2x 5mm LED](#)
- [2x 220 Ohm resistor \(or similar value\)](#)
- [Breadboard](#)
- [Jumper wires](#)



Building the Web Server

After wiring the circuit, the next step is uploading the code to your ESP32. Copy the code below to your Arduino IDE, but don't upload it yet. You need to make some changes to make it work.

- [Click here to download the code.](#)

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

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

// Assign output variables to GPIO pins
const int output26 = 26;
const int output27 = 27;
String output26State = "off";
String output27State = "off";

// Create a web server object
WebServer server(80);

// Function to handle turning GPIO 26 on
void handleGPIO26On() {
    output26State = "on";
    digitalWrite(output26, HIGH);
    handleRoot();
}

// Function to handle turning GPIO 26 off
void handleGPIO26Off() {
    output26State = "off";
    digitalWrite(output26, LOW);
    handleRoot();
}

// Function to handle turning GPIO 27 on
void handleGPIO27On() {
    output27State = "on";
    digitalWrite(output27, HIGH);
    handleRoot();
}

// Function to handle turning GPIO 27 off
void handleGPIO27Off() {
    output27State = "off";
    digitalWrite(output27, LOW);
    handleRoot();
}

// Function to handle the root URL and show the current states
void handleRoot() {
    String html = "<!DOCTYPE html><html><head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">";
    html += "<link rel=\"icon\" href=\"data:,\">";
    html += "<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center; }";
    html += ".button { background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; }";
    html += ".button2 { background-color: #555555; }</style></head>";
    html += "<body><h1>ESP32 Web Server</h1>";

    // Display GPIO 26 controls
    html += "<p>GPIO 26 - State " + output26State + "</p>";
}
```

```

if (output26State == "off") {
    html += "<p><a href=\"/26/on\"><button class=\"button\">ON</button></a></p>";
} else {
    html += "<p><a href=\"/26/off\"><button
        class=\"button button2\">OFF</button></a></p>";
}
// Display GPIO 27 controls
html += "<p>GPIO 27 - State " + output27State + "</p>";
if (output27State == "off") {
    html += "<p><a href=\"/27/on\"><button class=\"button\">ON</button></a></p>";
} else {
    html += "<p><a href=\"/27/off\"><button
        class=\"button button2\">OFF</button></a></p>";
}

html += "</body></html>";
server.send(200, "text/html", html);
}

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

    // Initialize the output variables as outputs
    pinMode(output26, OUTPUT);
    pinMode(output27, OUTPUT);
    // Set outputs to LOW
    digitalWrite(output26, LOW);
    digitalWrite(output27, LOW);

    // Connect to Wi-Fi network
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("WiFi connected.");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());

    // Set up the web server to handle different routes
    server.on("/", handleRoot);
    server.on("/26/on", handleGPIO26On);
    server.on("/26/off", handleGPIO26Off);
    server.on("/27/on", handleGPIO27On);
    server.on("/27/off", handleGPIO27Off);

    // Start the web server
    server.begin();
    Serial.println("HTTP server started");
}

void loop() {
    // Handle incoming client requests
    server.handleClient();
}

```

Setting Your Network Credentials

You need to modify the following lines with your network credentials: SSID and password.

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

Finding the ESP IP Address

Now, you can upload the code, and it will work straight away. Don't forget to check if you have the right board and COM port selected.

Open the Serial Monitor at a baud rate of 115200.

The ESP32 connects to Wi-Fi and prints its IP address on the Serial Monitor. Copy that IP address because you need it to access the ESP32 web server.



The screenshot shows the Arduino Serial Monitor window. The title bar says "Serial Monitor". The main area displays the following text:

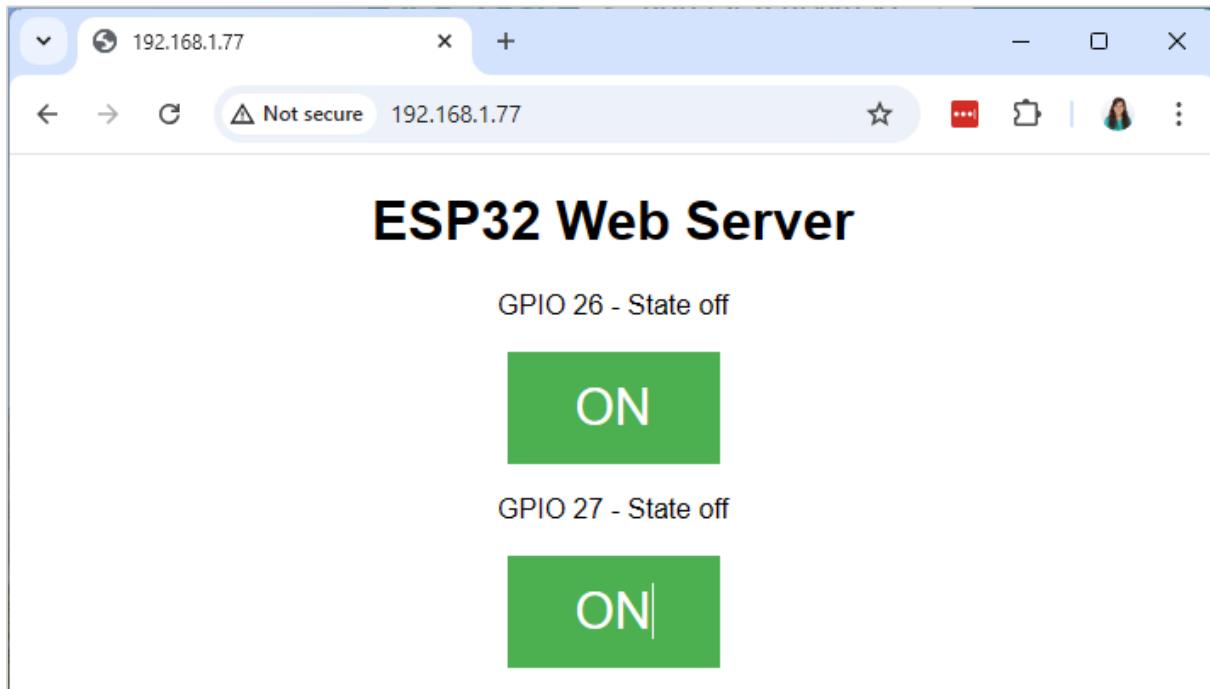
```
Output Serial Monitor X
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1') New Line 115200 baud
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Connecting to MFR ....
.
WiFi connected.
IP address:
192.168.1.77
HTTP server started
```

The IP address "192.168.1.77" is highlighted with a red rectangle. The status bar at the bottom of the monitor shows "Ln 4, Col 41 DOIT ESP32 DEVKIT V1 on COM3" and some icons.

Note: If nothing shows up on the Serial Monitor, press the ESP32 "EN" button (ENABLE/RESET button next to the microUSB port).

Accessing the Web Server

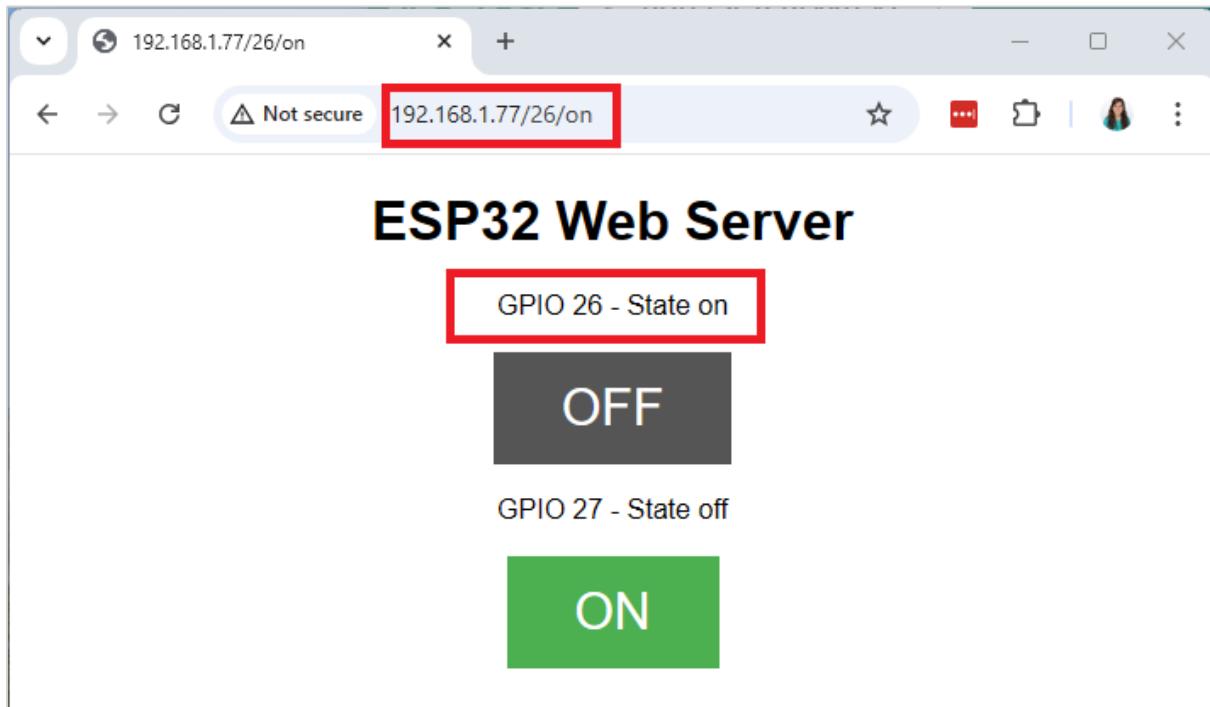
Open your browser, paste the ESP32 IP address, and you'll see the following page.



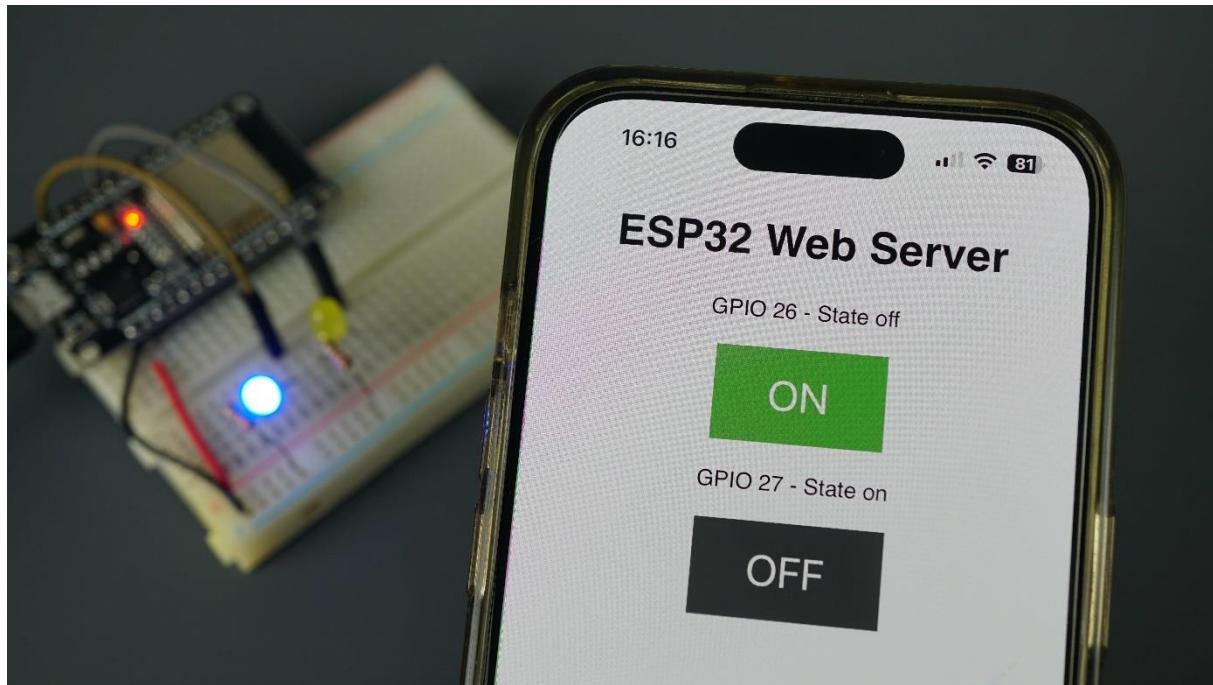
Testing the Web Server

Let's test the web server. Click the button to turn GPIO 26 ON. You can see on the Serial Monitor that the ESP32 receives a request on the **/26/on** URL.

When the ESP receives that request, it turns the LED attached to GPIO 26 ON, and its state is also updated on the web page. Test the button for GPIO 27 and see that it works similarly.



You can also access the web server on your smartphone as long as it is connected to the same network.



How the Code Works

Now, let's take a closer look at the code to see how it works so that you can modify it to fulfill your needs.

The first thing you need to do is include the necessary libraries for Wi-Fi connectivity and set up a web server.

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

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

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

Assign GPIO pins to each of your outputs. Here, we are using GPIO 26 and GPIO 27, but you can use any other suitable GPIOs.

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

Then, create variables to store the states of those outputs. You can add more outputs by defining additional variables.

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

Create a web server object on port 80.

```
// Create a web server object
WebServer server(80);
```

setup()

Now, let's go into the `setup()`. The `setup()` function only runs once when your ESP32 first boots. 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 ESP32 IP address in the Serial Monitor.

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

Finally, we set up the web server and define the routes it should handle. These routes will be requested when you click on the different buttons on the web page.

```
// Set up the web server to handle different routes
server.on("/", handleRoot);
server.on("/26/on", handleGPIO26On);
server.on("/26/off", handleGPIO26Off);
server.on("/27/on", handleGPIO27On);
server.on("/27/off", handleGPIO27Off);

// Start the web server
server.begin();
Serial.println("HTTP server started");
```

For example, when you make a request on the root / URL (you simply paste the ESP32 IP address on the web browser), it will run the `handleRoot()` function. When you click on the GPIO 26 ON button, it will make a request on the `/26/on` route and the board will run the `handleGPIO26On()` function, and so on... Those functions are defined at the beginning of the code before the `setup()`. We'll take a look at them next.

loop()

In the `loop()`, we continuously listen for incoming client requests. This ensures that the ESP32 is always ready to respond to requests from your browser.

```
server.handleClient();
```

Handling GPIO Control

On the web page, you've seen that you have four buttons to control the GPIOs:

- GPIO 26 ON button → request: `/26/on` → function: `handleGPIO26On()`
- GPIO 26 OFF button → request: `/26/off` → function: `handleGPIO26Off()`
- GPIO 27 ON button → request: `/27/on` → function: `handleGPIO27On()`
- GPIO 27 OFF button → request: `/27/off` → function: `handleGPIO27Off()`

Let's take a look at the GPIO 26 ON button. When you click that button, it makes a request to the ESP32 on the `/26/on` URL. When that happens, the `handleGPIO26On()` function will run.

```
// Function to handle turning GPIO 26 on
void handleGPIO26On() {
    output26State = "on";
    digitalWrite(output26, HIGH);
    handleRoot();
}
```

That function updates the state of the GPIO on the `output26State` variable and turns the GPIO on. Finally, it calls the `handleRoot()` function to display the web page with the right GPIO state.

This works similarly to the other buttons and corresponding routes and functions. Notice that all of these functions call the `handleRoot()` function to display the web page with the right GPIO states.

```
// Function to handle turning GPIO 26 off
void handleGPIO26Off() {
    output26State = "off";
    digitalWrite(output26, LOW);
    handleRoot();
}

// Function to handle turning GPIO 27 on
void handleGPIO27On() {
    output27State = "on";
    digitalWrite(output27, HIGH);
    handleRoot();
}

// Function to handle turning GPIO 27 off
void handleGPIO27Off() {
    output27State = "off";
    digitalWrite(output27, LOW);
    handleRoot();
}
```

Displaying the Web Page

The `handleRoot()` function is responsible for generating the web page. It sends the HTML and CSS needed to build the page. The HTML and CSS text required to build the web page are saved on the `html` variable.

```
// Function to handle the root URL and show the current states
void handleRoot() {
    String html = "<!DOCTYPE html><html><head><meta name=\"viewport\""
        "content=\"width=device-width, initial-scale=1\">";
    html += "<link rel=\"icon\" href=\"data:,\">";
    html += "<style>html { font-family: Helvetica; display: inline-block;
        margin: 0px auto; text-align: center;}</style>";
    html += ".button { background-color: #4CAF50; border: none; color: white;
        padding: 16px 40px; text-decoration: none; font-size: 30px;
        margin: 2px; cursor: pointer;}";
    html += ".button2 { background-color: #555555; }</style></head>";
    html += "<body><h1>ESP32 Web Server</h1>";

    // Display GPIO 26 controls
    html += "<p>GPIO 26 - State " + output26State + "</p>";
    if (output26State == "off") {
        html += "<p><a href=\"/26/on\"><button class=\"button\">ON</button></a></p>";
    } else {
```

```

        html += "<p><a href=\"/26/off\"><button
            class=\"button button2\">OFF</button></a></p>";
    }

    // Display GPIO 27 controls
    html += "<p>GPIO 27 - State " + output27State + "</p>";
    if (output27State == "off") {
        html += "<p><a href=\"/27/on\"><button class=\"button\">ON</button></a></p>";
    } else {
        html += "<p><a href=\"/27/off\"><button
            class=\"button button2\">OFF</button></a></p>";
    }

    html += "</body></html>";
    server.send(200, "text/html", html);
}

```

Note that to generate the buttons, we use if and else statements to display the correct button and state accordingly to the current state of the GPIOs.

```

// Display GPIO 26 controls
html += "<p>GPIO 26 - State " + output26State + "</p>";
if (output26State == "off") {
    html += "<p><a href=\"/26/on\"><button class=\"button\">ON</button></a></p>";
} else {
    html += "<p><a href=\"/26/off\"><button class=\"button button2\">OFF</button></a></p>";
}

// Display GPIO 27 controls
html += "<p>GPIO 27 - State " + output27State + "</p>";
if (output27State == "off") {
    html += "<p><a href=\"/27/on\"><button class=\"button\">ON</button></a></p>";
} else {
    html += "<p><a href=\"/27/off\"><button class=\"button button2\">OFF</button></a></p>";
}

```

Finally, the web page is sent to the client:

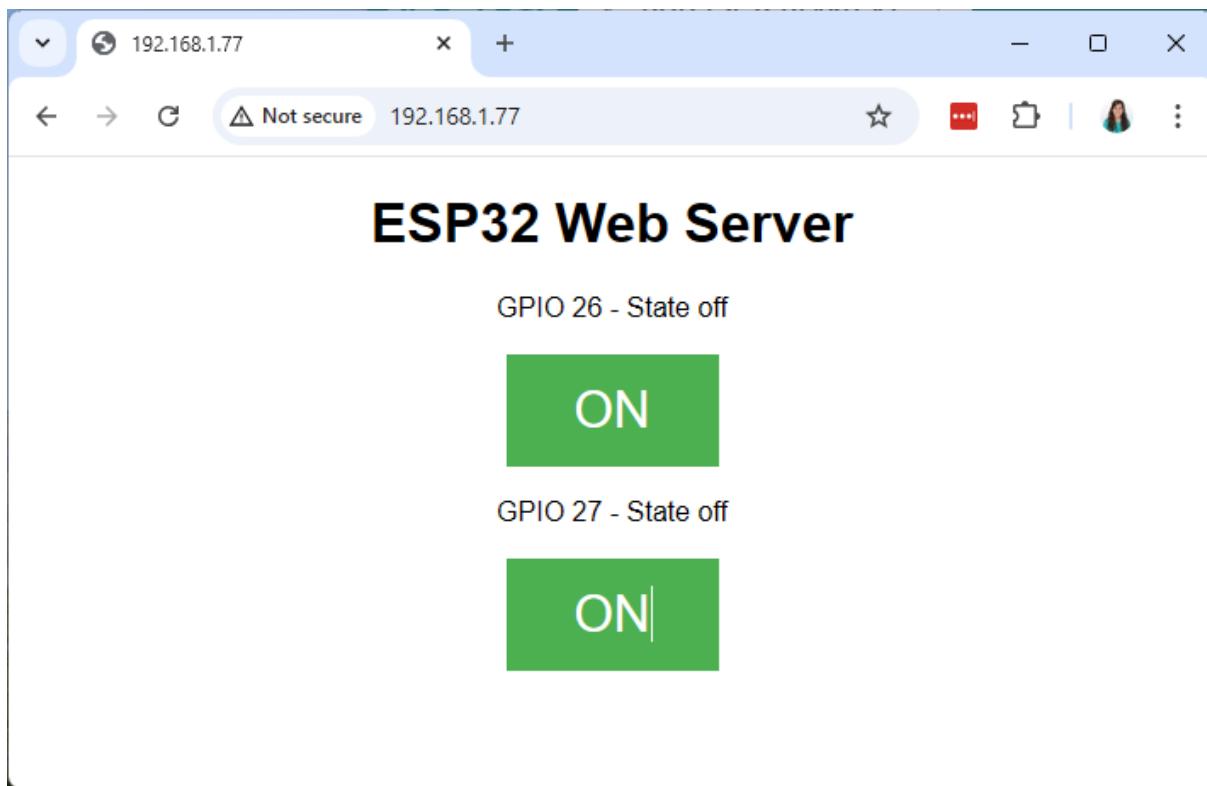
```
server.send(200, "text/html", html);
```

Wrapping Up

Now that you know how the code works, you can modify the code to add more outputs, or change your web page. To modify your web page, you may need to know some HTML and CSS basics. We'll cover those topics in the next unit.

6.3 - ESP32 Web Server: HTML and CSS Basics

In the previous Unit, you've learned how to build a web server with the ESP32 to control outputs.



You've seen that your browser displays a web page when you access your ESP32 IP address—this happens because your ESP32 sends some HTML text to generate the web page when you make a request. You can easily change how your web page looks like by editing the HTML the ESP32 sends to the browser. For that, it is useful to know some HTML and CSS basics.

In this Unit, we're going to build the web page from the previous Unit with step-by-step instructions so that you can easily change how it looks.

Introducing HTML

HTML stands for Hypertext Markup Language and is the predominant markup language used to create web pages. Web browsers were created to read HTML

files—the HTML tags tell the web browser how to display the content on the page. We'll see how tags work in the next example.

Setting up the Basics

The following snippet shows the overall structure of an HTML document.

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
</body>
</html>
```

The first line of any HTML document is always `<!DOCTYPE html>`. This tells the web browser this document is an HTML file.

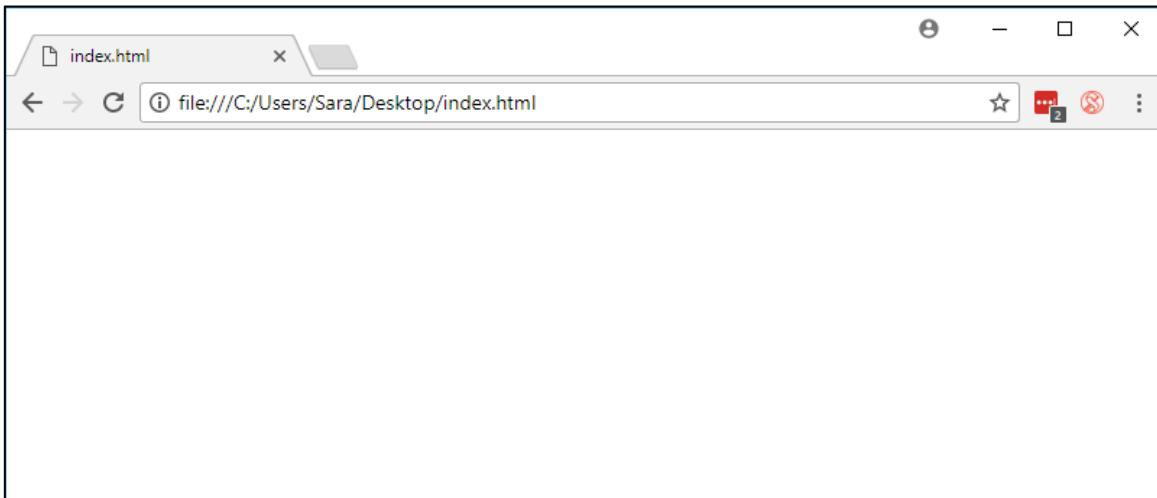
The structure of the web page should go between the `<html>` and `</html>` tags. The `<html>` tag indicates the beginning of a web page, and the `</html>` tag indicates the end of the page.

The HTML document is divided into two main parts: the head and the body. The head goes within the `<head>` and `</head>` tags and the body within the `<body>` and `</body>` tags.

The head is where you insert data about the HTML document that is not directly visible to the end-user but adds functionalities to the web page like the title, scripts, styles, and more—this is called metadata. The body includes the page's content, like headings, text, buttons, tables, etc.

Open a Text Editor program (you can use any text editor, we use [VS Code](#)) and copy the previous HTML text. Save the file as *index.html*. Open your browser and drag the HTML file to a browser tab. You'll just see a blank page because you haven't added anything to the HTML file yet.

Note: the file must be called *index.html*, not *index.html.txt*.

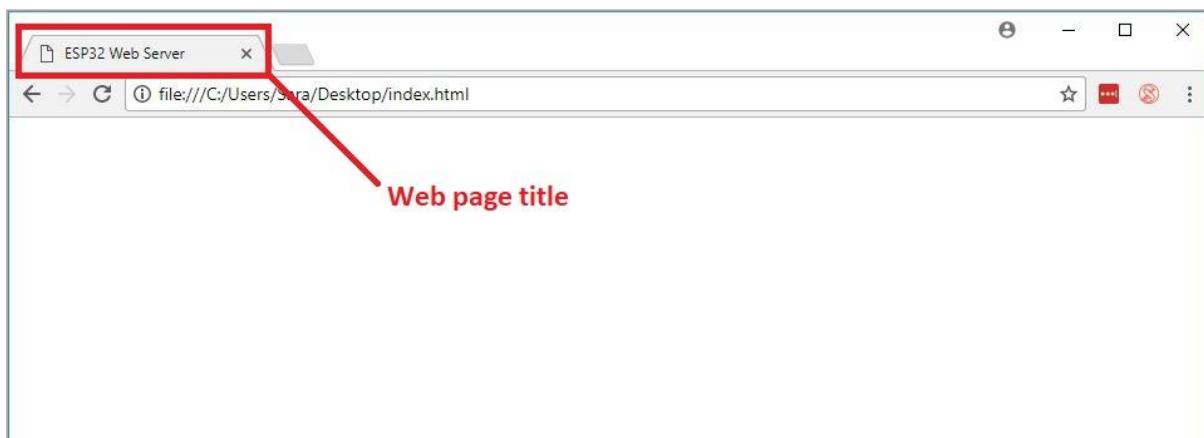


Title

The title of your web page is the text that shows in the web browser tab. The web page title should go between the `<title>` and `</title>` tags, that should go between the `<head>` and `</head>` tags. Add a title to your web page by typing the title between `<title>` and `</title>` tags, as shown in the example below.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
</head>
<body>
</body>
</html>
```

Here, the title of our web page is “**ESP32 Web Server**” but you can call it whatever you want. Save your `index.html` file and refresh the web browser tab. You should see the title in the browser tab as shown in the figure below.



Headings

Headings are used to structure the text on the web page. Headings begin with an **h** followed by a number that indicates the heading strength. For example, `<h1>` and `</h1>` are the tags for heading 1, `<h2>` and `</h2>` for heading 2, until heading 6.

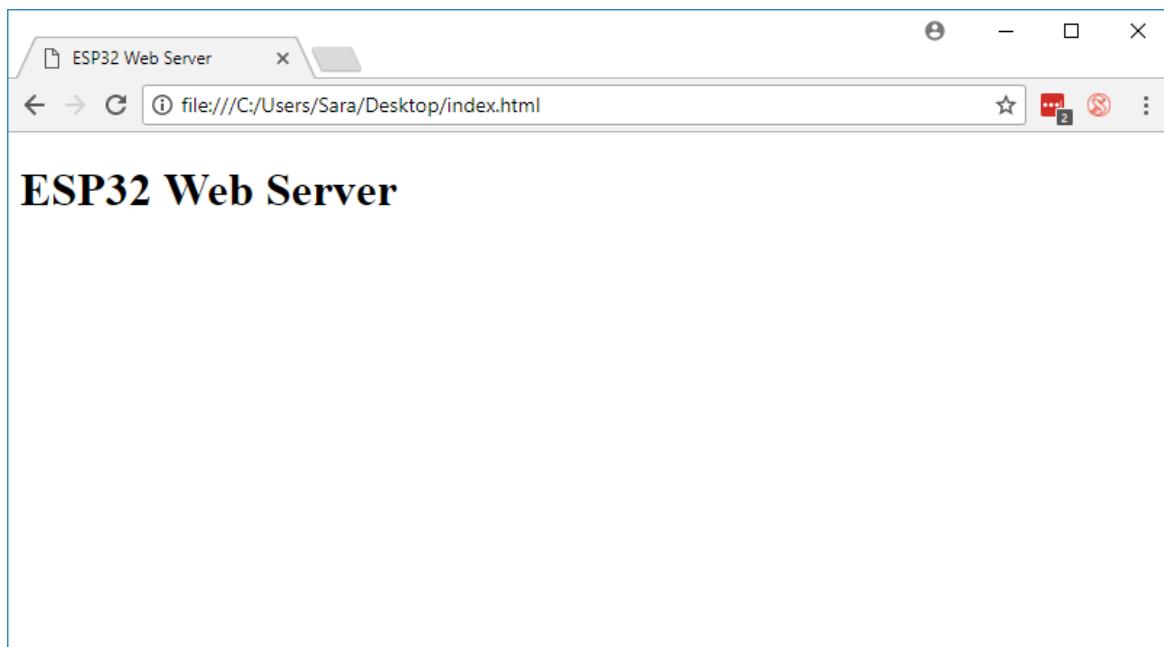
The heading tags should be between the `<body>` and `</body>` tags.

Add some headings to your document. You can use the following as a reference.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title></head>
<body>
  <h1>ESP32 Web Server</h1>
</body>
</html>
```

Here we've added the first heading with the text "**ESP32 Web Server**". We recommend you add several headings with different levels to practice with those tags.

Save your *index.html* file and refresh the web browser tab. You should have something as follows.

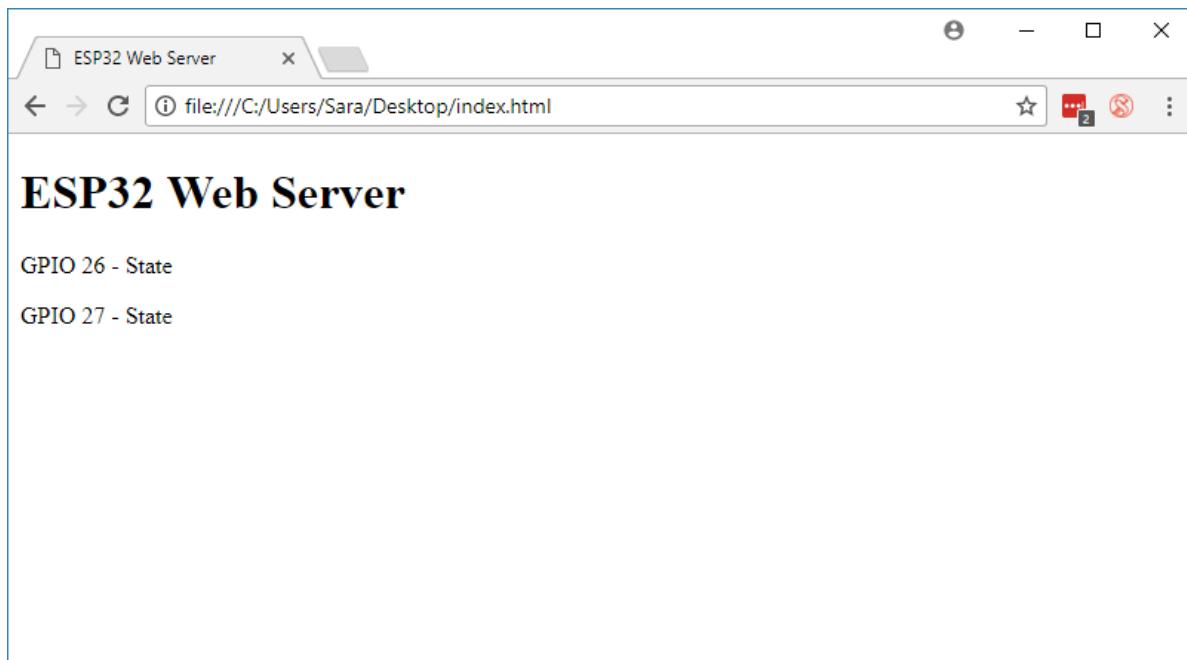


Paragraphs

The paragraphs are used to place text. Every paragraph should go between the `<p>` and `</p>` tags. Add some paragraphs to show the state of GPIO 26 and GPIO 27.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
</head>
<body>
  <h1>ESP32 Web Server</h1>
  <p>GPIO 26 - State</p>
  <p>GPIO 27 - State</p>
</body>
</html>
```

Save the `index.html` file, and refresh the web browser. You should have something like in the following figure.



The “ON” and “OFF” states will then be added to each of the paragraphs using a variable created in the Arduino sketch that saves the state of each GPIO as we’ve seen in the previous unit.

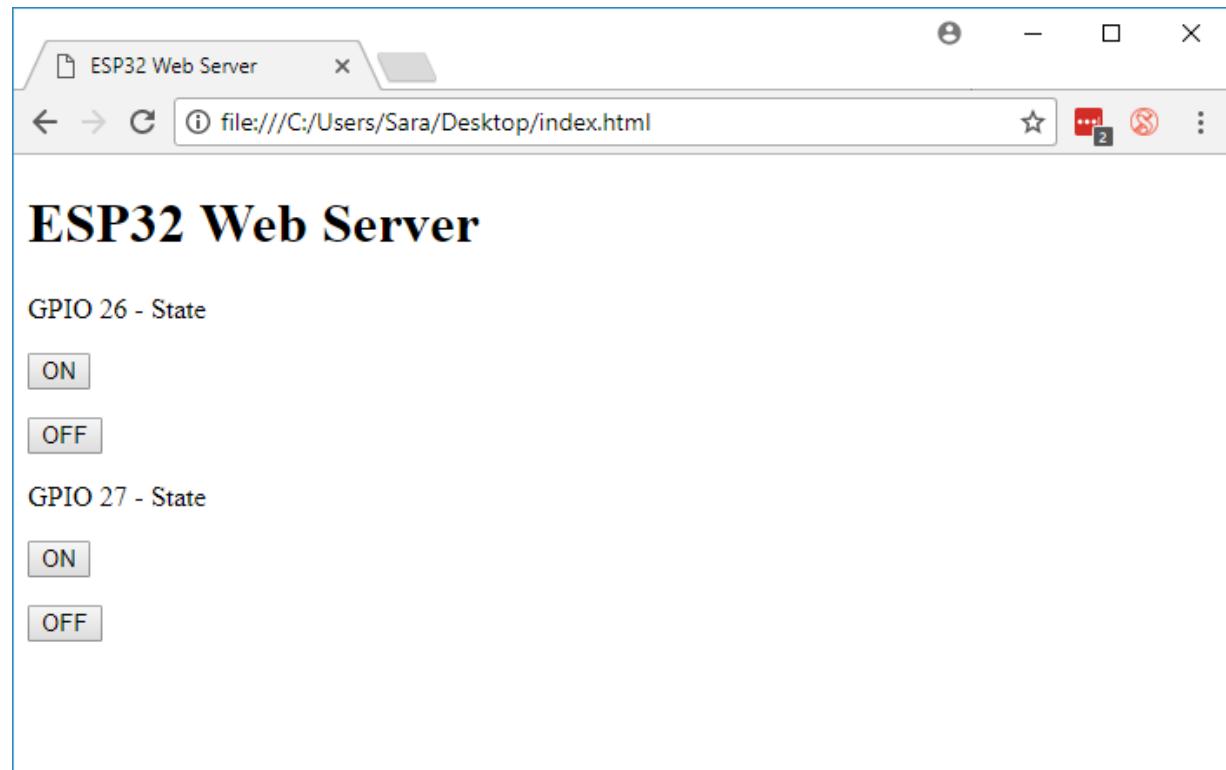
Buttons

To insert a button in your page you use the `<button>` and `</button>` tags. Between the tags you should write the text you want to appear inside the button. Add an

“ON” and an “OFF” button for each of the GPIOs. Note that we’ve inserted the `<button>` and `</button>` tags between paragraph tags `<p>` and `</p>`.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
</head>
<body>
  <h1>ESP32 Web Server</h1>
  <p>GPIO 26 - State</p>
  <p><button>ON</button></p>
  <p><button>OFF</button></p>
  <p>GPIO 27 - State</p>
  <p><button>ON</button></p>
  <p><button>OFF</button></p>
</body>
</html>
```

Now, your web page should have four buttons, as shown in the figure below.



Click on the buttons. Nothing happens because those buttons don’t have any hyperlinks associated with them. We need to add hyperlinks to the buttons—which we’ll do in the next section.

The web server in the previous Unit shows only one button for each GPIO. Depending on its current state, we decide which button is displayed in the Arduino code:

- If the current state is “ON”, the page should display the “OFF” button – the ESP32 should send the HTML text to build the “OFF” button;
- If the current state is “OFF”, the page should display the “ON” button – the ESP32 should send the HTML text to build the “ON” button;

We've seen this in the previous unit.

Hyperlinks

HTML links are called hyperlinks. You can add hyperlinks to text, images, buttons, or any other HTML element. To add a hyperlink, you use the `<a>` and `` tags, in the following format:

```
<a href="url">element</a>
```

Between the `<a>` and `` tags, you should place the HTML element you want to apply the link to. For example, to apply the link to one of the “OFF” buttons:

```
<a href="url"><button>OFF</button></a>
```

The `href` attribute specifies where the link should go. When you click the GPIO 26 ON button, you want to be redirected to the root page followed by `/26/on`. To do that, you should add that URL to the `href` attribute, as follows:

```
<a href="/26/on"><button>ON</button></a>
```

When you click the GPIO 26 OFF button, you want to redirect to `/26/off`:

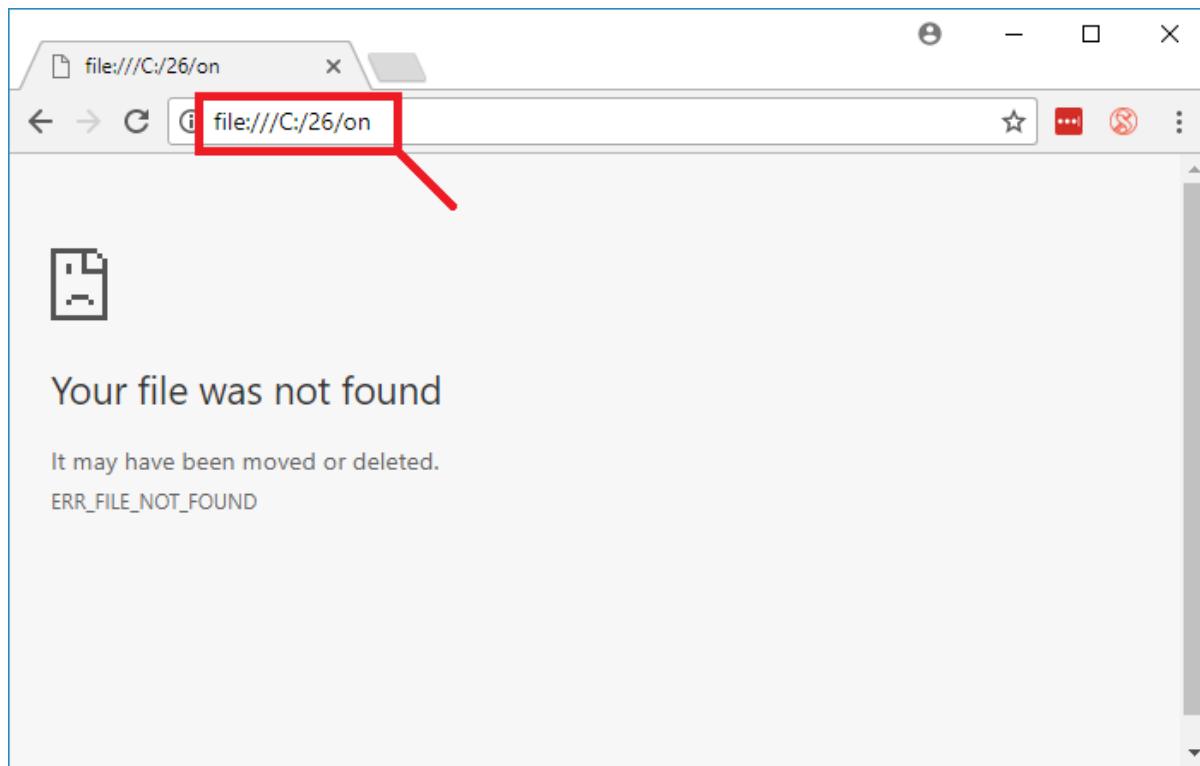
```
<a href="/26/off"><button>OFF</button></a>
```

You should add the appropriate hyperlink to each of your buttons as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
</head>
<body>
  <h1>ESP32 Web Server</h1>
  <p>GPIO 26 - State</p>
  <p><a href="/26/on"><button>ON</button></a></p>
  <p><a href="/26/off"><button>OFF</button></a></p>
  <p>GPIO 27 - State</p>
  <p><a href="/27/on"><button>ON</button></a></p>
  <p><a href="/27/off"><button>OFF</button></a></p>
</body>
```

```
</html>
```

Save your *index.html* file, and refresh your web browser. Your web page won't change. But when you click on the buttons, you'll be redirected to the URL you've set previously. For example, when you click on the GPIO 26 ON button, you'll be redirected to the **/26/on** URL, as shown below:



At the moment, you get the error “file was not found” because you don’t have any file to that URL. This will be solved on the Arduino IDE, because your ESP32 will send different HTML text when you click on the buttons as you’ve seen in the previous unit—so, don’t worry about this error at the moment.

Now, you just need to check that each button redirects to the right URL.

The **class** attribute

The **class** attribute specifies one or more class names for an element. Class names are useful to define styles in CSS. For example, to style a group of HTML elements with the same style, we can give them the same class name.

To add the class attribute to an HTML element, you use the following syntax:

```
<element class="classname"></element>
```

An HTML element can have more than one class name that must be separated by a space. In our example we specify the class name “button” for the ON buttons. For the OFF buttons we specify both the class “button”, and the class “button2”. So, for GPIO 26 buttons you’ll have:

```
<p><a href="/26/on"><button class="button">ON</button></a></p>
<p><a href="/26/off"><button class="button button2">OFF</button></a></p>
```

You should do the same for GPIO 27 buttons. So, your HTML will be as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
</head>
<body>
  <h1>ESP32 Web Server</h1>
  <p>GPIO 26 - State</p>
  <p><a href="/26/on"><button class="button">ON</button></a></p>
  <p><a href="/26/off"><button class="button button2">OFF</button></a></p>
  <p>GPIO 27 - State</p>
  <p><a href="/27/on"><button class="button">ON</button></a></p>
  <p><a href="/27/off"><button class="button button2">OFF</button></a></p>
</body>
</html>
```

Metadata

In the head of the *index.html* file, we also add the `<meta>` tag that makes your web page responsive in any web browser.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

The `<meta>` tag provides metadata about the HTML document. Metadata will not be displayed on the page, but provides useful information to the browser such as how to display the content.

We also add the following line:

```
<link rel="icon" href="data:, ">
```

To prevent requests from the browser to the ESP32 about the favicon. The favicon is the shortcut icon that appears on the web browser tab.

Introducing CSS

CSS stands for Cascading Style Sheets, and it is used to describe how the elements in a web page look. It describes a certain part of the page, like a particular tag or a particular set of tags. The CSS can be added to the HTML file or in a separate file that is referenced by the HTML file. In the previous example, we added the CSS in the middle of the HTML text between the `<style>` and `</style>` tags, that should go in the head of the HTML file.

So, the head of the HTML file will be as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
  <style>YOUR CSS GOES HERE</style>
</head>
```

CSS uses selectors to style your HTML content. The selector points to the HTML element you want to style. Selectors have properties, which in turn have values.

```
selector {
  property: value;
}
```

The style for a particular selector should go between curly brackets {}. The value is attributed to a property using a colon (:). Every value should end with a semicolon (;). Each selector can have, and normally does have, more than one property. Let's better understand how this works with the next section.

Styling the page

In our example, we style the `html` element with the following styles:

```
html {
  font-family: Helvetica;
  display: inline-block;
  margin: 0px auto;
  text-align: center;
}
```

The properties set to the `html` element will be applied to the whole web page—remember that all your web page content goes between the `<html>` and `</html>` tags. We define the font-family to Helvetica, the content

is displayed as a block, you set 0px for the margins and align all the page at the center using "auto". All your text will be aligned at the center.

At this point, your *index.html* file should be as follows (notice the CSS highlighted in yellow):

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,">
  <style>
    html {
      font-family: Helvetica;
      display: inline-block;
      margin: 0px auto;
      text-align: center;
    }
  </style>
</head>
<body>
  <h1>ESP32 Web Server</h1>
  <p>GPIO 26 - State</p>
  <p><a href="/26/on"><button class="button">ON</button></a></p>
  <p><a href="/26/off"><button class="button button2">OFF</button></a></p>
  <p>GPIO 27 - State</p>
  <p><a href="/27/on"><button class="button">ON</button></a></p>
  <p><a href="/27/off"><button class="button button2">OFF</button></a></p>
</body>
</html>
```

And your web page looks like the one showing on the following figure:



You can change how your page looks by applying other values for the properties we've defined. We encourage you to play with the values and see how your web pages look. You can use the following links as a reference to find more values for those properties:

- [font-family](#)
- [display](#)
- [margin](#)
- [text-align](#)

Styling the buttons

Previously we've defined the class "button" for the "ON" buttons and the classes "button" and "button2" for the "OFF" buttons.

To select elements with a specific class, we use a period (.) followed by the class name – as shown below.

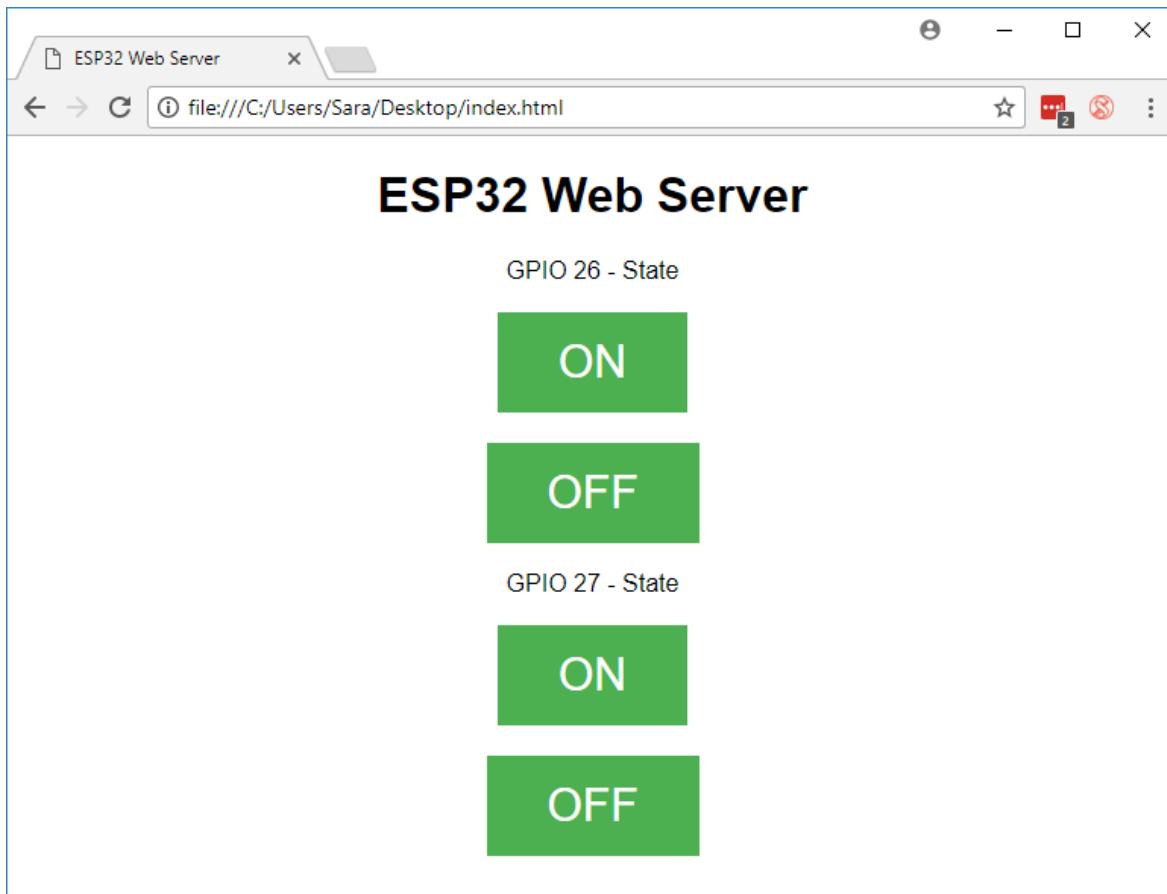
```
.button {  
background-color: #4CAF50;  
border: none;  
color: white;  
padding: 16px 40px;  
text-decoration: none;  
font-size: 30px;  
margin: 2px;  
cursor: pointer;  
}
```

The `background-color` property, as the name suggests, defines the button's background color. The colors can be set by using their name—HTML recognizes basic color names—or by using the hexadecimal or RGB color code. Search on the web for "hexadecimal color picker" to search for a hexadecimal reference for a specific color. Here we're using hexadecimal color code.

We set the `border` property to `none`, and the button text to white. The `padding` defines a space around the button—in this case, we set 16px by 40px. We set the `text-decoration` to `none`, font size of 30px, margin of 2px, and the `cursor` to a `pointer`—this changes the cursor to a pointer when you drag the mouse over the button.

You can set other values for those properties. For that, search the web for the property you want followed by the keyword “values CSS”. For example, to find what values the cursor property can take, you search for “cursor property values CSS”.

Add the previous CSS text to your *index.html* file—that text should go between the `<style>` and `</style>` tags. Save your *index.html* document and refresh the browser tab. You’ll have something as follows:

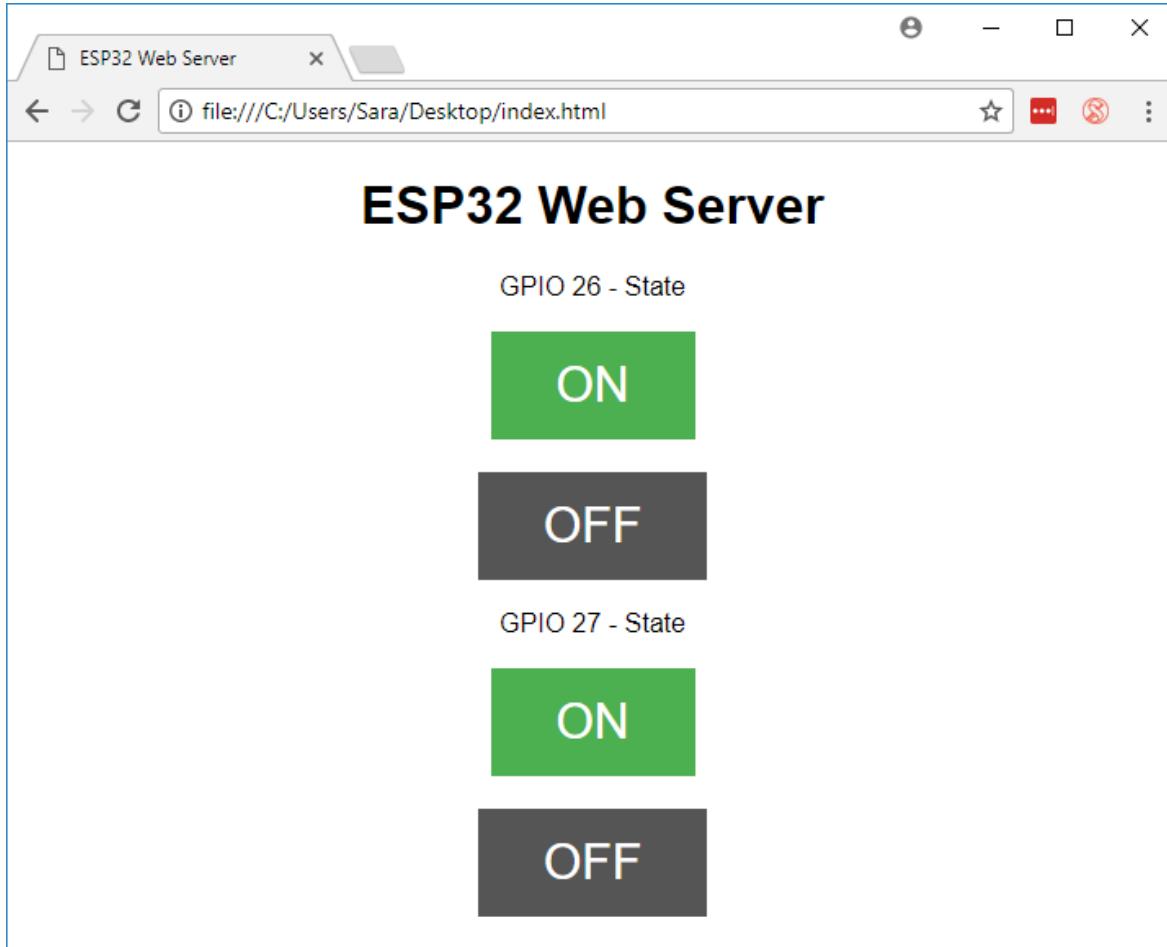


The off button belongs to class name “button” and class name “button2”, so it will have properties from both. We define the style for the “button2” class name as follows:

```
.button2 {  
    background-color: #555555  
}
```

Here we’re just defining a different color for the OFF button. Of course, you may choose any color you want.

Add the CSS to style for the “button2” class name to your HTML file, save the file and refresh the web page. The following figure shows what you should get.



You can add other customization to your buttons by searching for more button properties. You can take a look [here](#) to find more properties to style your buttons.

This was a general introduction to HTML and CSS and how the web page used in the previous Unit was created.

Complete HTML Text

Here is how your complete HTML file should be.

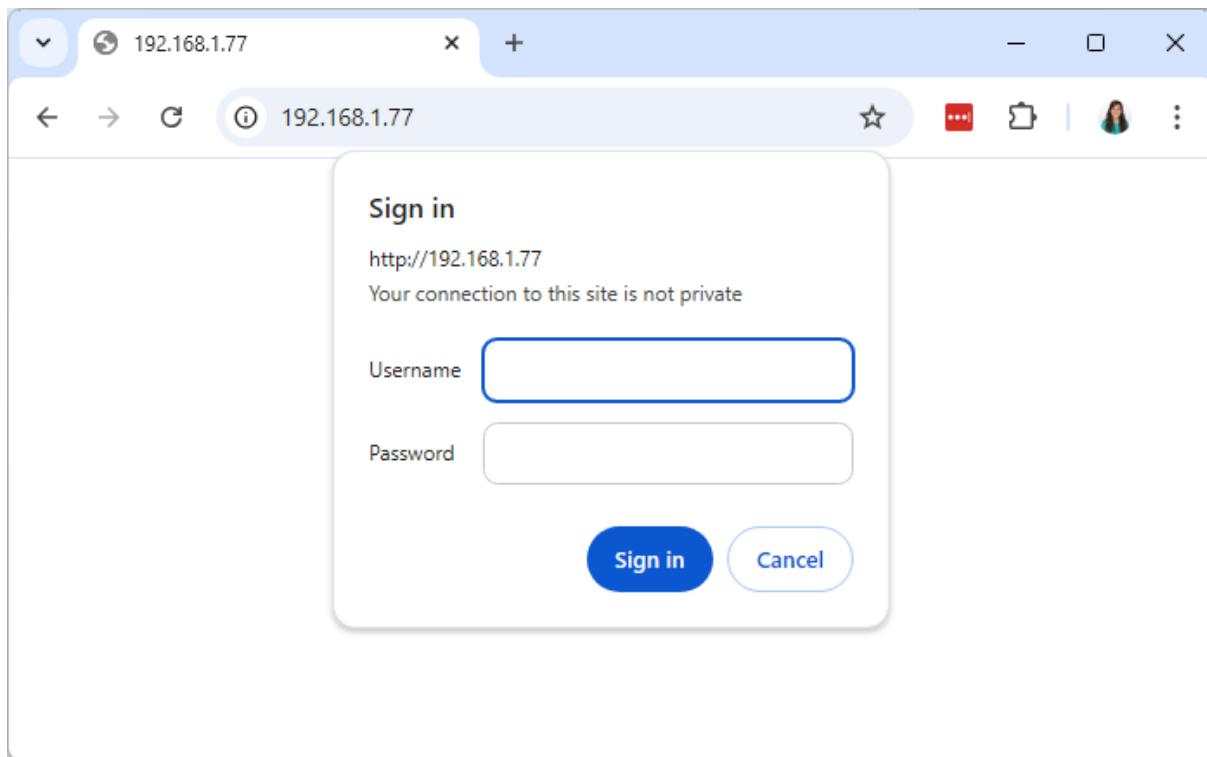
- [Click here to download the HTML file.](#)

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,>
<style>
  html {
    font-family: Helvetica;
    display: inline-block;
    margin: 0px auto;
    text-align: center;
  }
```

```
.button {
    background-color: #4CAF50;
    border: none;
    color: white;
    padding: 16px 40px;
    text-decoration: none;
    font-size: 30px;
    margin: 2px;
    cursor: pointer;
}
.button2 {
    background-color: #555555;
}
</style>
</head>
<body>
    <h1>ESP32 Web Server</h1>
    <p>GPIO 26 - State</p>
    <p><a href="/26/on"><button class="button">ON</button></a></p>
    <p><a href="/26/off"><button class="button button2">OFF</button></a></p>
    <p>GPIO 27 - State</p>
    <p><a href="/27/on"><button class="button">ON</button></a></p>
    <p><a href="/27/off"><button class="button button2">OFF</button></a></p>
</body>
</html>
```

6.4 - ESP32 Web Server: Authentication

In some projects, you might want to keep your ESP32 web server protected. After implementing the feature presented in this Unit, when you try to access your web server, you need to enter a valid username and a password. We'll add this feature to the web server built in the previous units.



Adding Username and Password

The following code protects your web server with username and password. By default, the username is *admin*, and the password is *admin*. The new lines of code that deal with the authentication are highlighted in light yellow.

Upload the next sketch and upload it to your ESP32 (after replacing the variables with the SSID and password):

- [Click here to download the code.](#)

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

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

```

// Username and password for web page access
const char* http_username = "admin";
const char* http_password = "admin";

// Assign output variables to GPIO pins
const int output26 = 26;
const int output27 = 27;
String output26State = "off";
String output27State = "off";

// Create a web server object
WebServer server(80);

// Function to authenticate user
bool isAuthenticated() {
    if (!server.authenticate(http_username, http_password)) {
        server.requestAuthentication();
        return false;
    }
    return true;
}

// Function to handle turning GPIO 26 on
void handleGPIO26On() {
    if (!isAuthenticated()) return;
    output26State = "on";
    digitalWrite(output26, HIGH);
    handleRoot();
}

// Function to handle turning GPIO 26 off
void handleGPIO26Off() {
    if (!isAuthenticated()) return;
    output26State = "off";
    digitalWrite(output26, LOW);
    handleRoot();
}

// Function to handle turning GPIO 27 on
void handleGPIO27On() {
    if (!isAuthenticated()) return;
    output27State = "on";
    digitalWrite(output27, HIGH);
    handleRoot();
}

// Function to handle turning GPIO 27 off
void handleGPIO27Off() {
    if (!isAuthenticated()) return;
    output27State = "off";
    digitalWrite(output27, LOW);
    handleRoot();
}

// Function to handle the root URL and show the current states
void handleRoot() {
    if (!isAuthenticated()) return;

    String html = "<!DOCTYPE html><html><head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">";
    html += "<link rel=\"icon\" href=\"data:,\">";

```

```

html += "<style>html { font-family: Helvetica; display: inline-block;
    margin: 0px auto; text-align: center;}>";
html += ".button { background-color: #4CAF50; border: none;
    color: white; padding: 16px 40px; text-decoration: none;
    font-size: 30px; margin: 2px; cursor: pointer;}";
html += ".button2 { background-color: #555555; }</style></head>";
html += "<body><h1>ESP32 Web Server</h1>";

// Display GPIO 26 controls
html += "<p>GPIO 26 - State " + output26State + "</p>";
if (output26State == "off") {
    html += "<p><a href=\"/26/on\"><button class=\"button\">ON
        </button></a></p>";
} else {
    html += "<p><a href=\"/26/off\"><button
        class=\"button button2\">OFF</button></a></p>";
}

// Display GPIO 27 controls
html += "<p>GPIO 27 - State " + output27State + "</p>";
if (output27State == "off") {
    html += "<p><a href=\"/27/on\"><button class=\"button\">ON
        </button></a></p>";
} else {
    html += "<p><a href=\"/27/off\"><button
        class=\"button button2\">OFF</button></a></p>";
}

html += "</body></html>";
server.send(200, "text/html", html);
}

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

    // Initialize the output variables as outputs
    pinMode(output26, OUTPUT);
    pinMode(output27, OUTPUT);
    // Set outputs to LOW
    digitalWrite(output26, LOW);
    digitalWrite(output27, LOW);

    // Connect to Wi-Fi network
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("WiFi connected.");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());

    // Set up the web server to handle different routes with authentication
    server.on("/", handleRoot);
    server.on("/26/on", handleGPIO26On);
    server.on("/26/off", handleGPIO26Off);
    server.on("/27/on", handleGPIO27On);
    server.on("/27/off", handleGPIO27Off);
}

```

```
// Start the web server
server.begin();
Serial.println("HTTP server started");
}

void loop() {
    // Handle incoming client requests
    server.handleClient();
}
```

Username and Password

In the following lines insert the username and password you must enter to access the web page. By default, we're setting the username to `admin` and the password to `admin`.

```
// Username and password for web page access
const char* http_username = "admin";
const char* http_password = "admin";
```

Handling Authentication and Requests

To protect your web page, we added a simple authentication mechanism. Before accessing any part of the web server, the user must enter the correct username and password. This is handled by the `isAuthenticated()` function.

```
// Function to authenticate user
bool isAuthenticated() {
    if (!server.authenticate(http_username, http_password)) {
        server.requestAuthentication();
        return false;
    }
    return true;
}
```

This function checks if the user is authenticated. If not, it prompts the user to enter the credentials. The function returns `true` if the user is authenticated and `false` otherwise.

Before sending any response to the client, we check if the user is authenticated. Notice the use of the following line before returning anything to the client.

```
if (!isAuthenticated()) return;
```

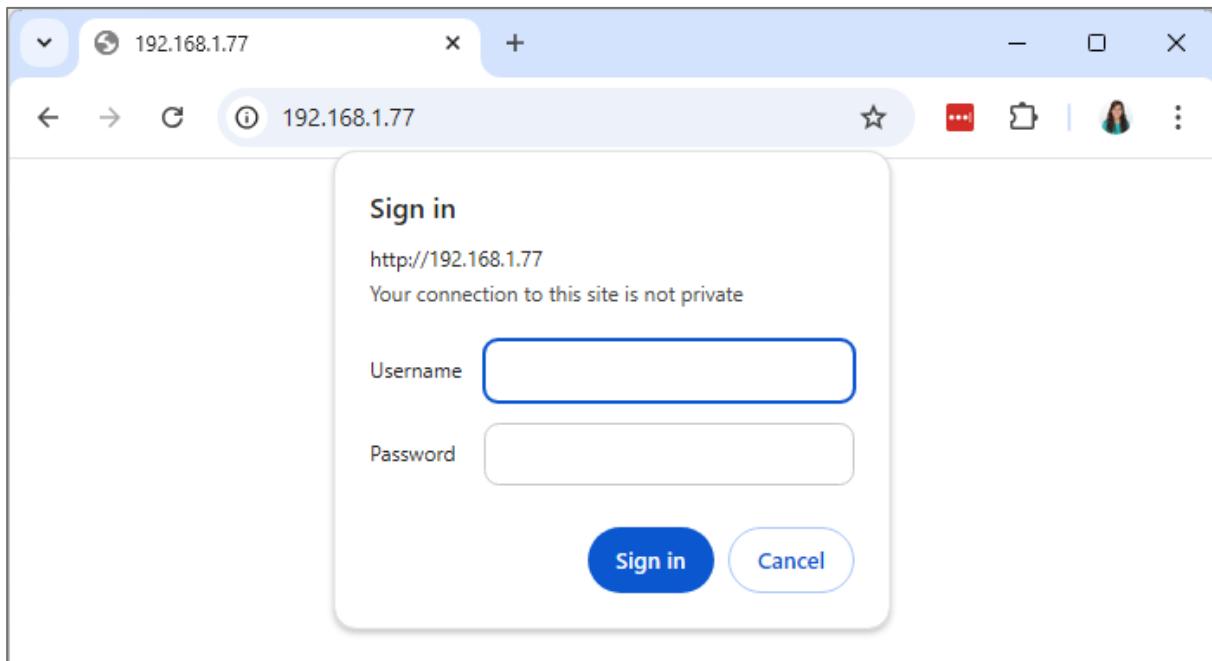
For example, in the case of the `handleGPIO26On()` function:

```
// Function to handle turning GPIO 26 on
```

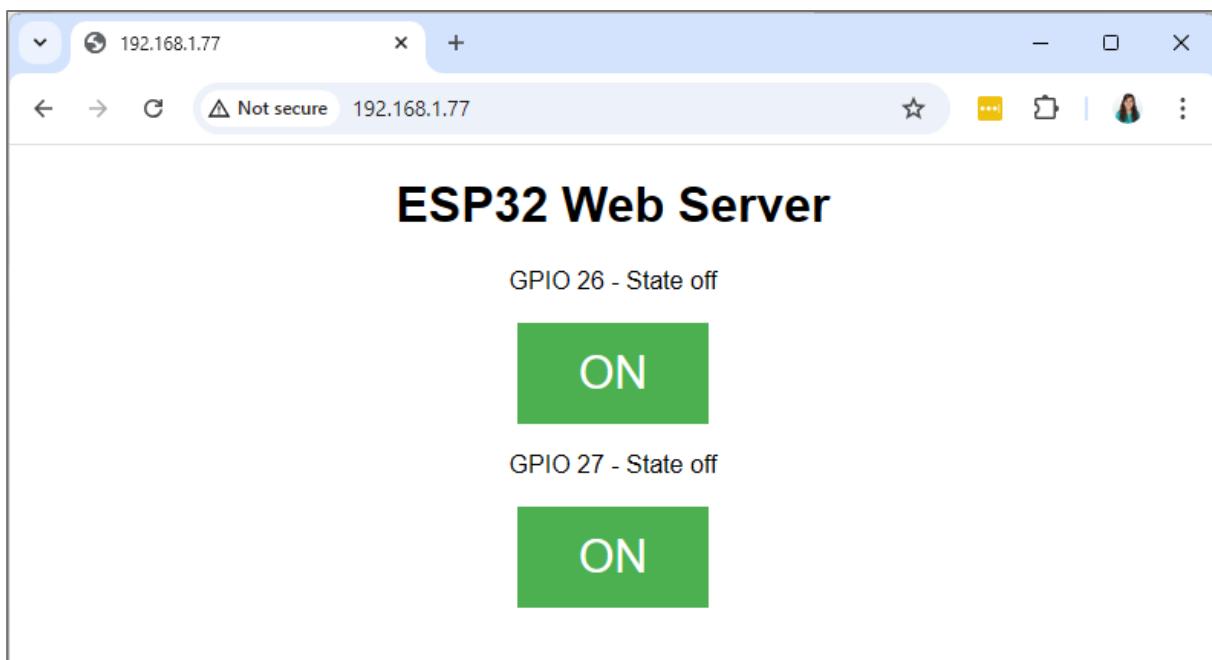
```
void handleGPIO26On() {
    if (!isAuthenticated()) return;
    output26State = "on";
    digitalWrite(output26, HIGH);
    handleRoot();
}
```

Testing the Web Server

Now, when you try to access your ESP32 IP address, you'll be required to enter your username and password. Then, press the “Sign in” button:



You'll be able to see your ESP32 web server:



If you enter the wrong password, the Sign in box will show up again.

Wrapping Up

In this Unit, you've learned how to make your ESP32 web server password protected. You can add this functionality to any web server you want to build using the same libraries we used in this example.

In the next Unit, you'll learn how to make your web server accessible from anywhere (outside the local network).

6.5 - Accessing the ESP32 Web Server from Anywhere

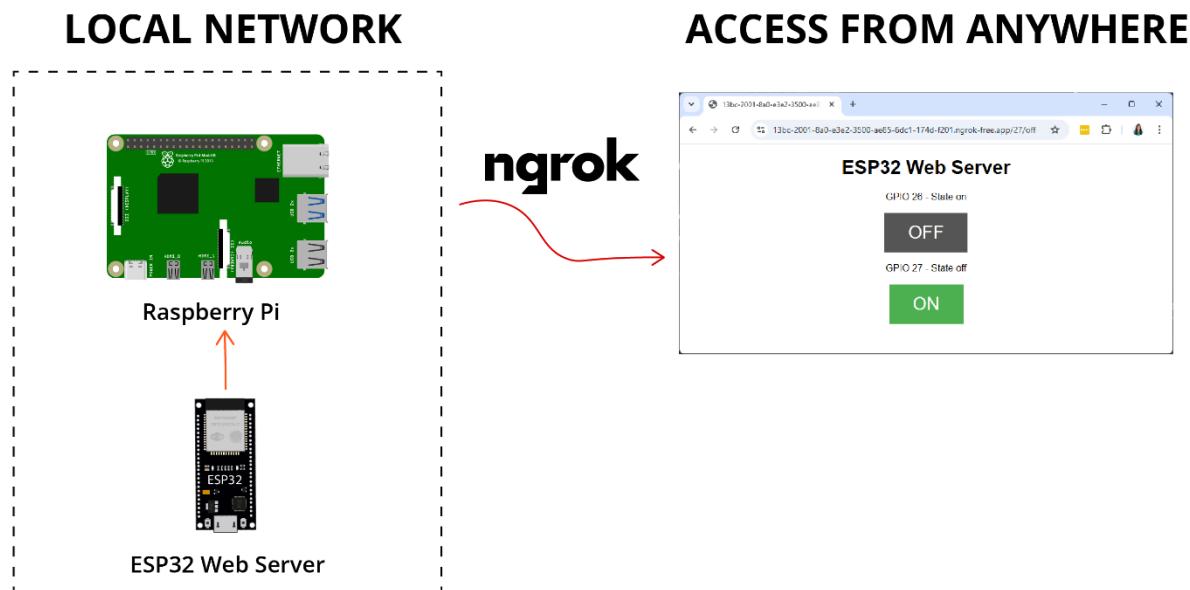
In this Unit, you're going to make your ESP32 web server accessible from anywhere in the world. At this moment, the ESP32 web server is only accessible when you are connected to your local network.

We'll use a free service called [ngrok](#) that allows you to create a secure tunnel to your local network.

Note: this method requires having a computer turned on running the **ngrok** software. You can run this software on a low-cost/low-power computer such as the Raspberry Pi.

Overview

The next diagram shows a high-level overview of how everything works.



Note: this method requires having a computer running *ngrok* software. For a permanent solution, it is a good idea to run this software on the low-cost Raspberry Pi computer. Ngrok can also run on your Windows PC, Mac OS X or Linux.

Prerequisites (if using a Raspberry Pi)

Before continuing:

- You should be familiar with the Raspberry Pi board – [read Getting Started with Raspberry Pi](#);
- You should have the Raspberry Pi OS or Raspberry Pi OS installed in your Raspberry Pi – [read Installing Raspbian Lite, Enabling and Connecting with SSH](#);
- You also need the following hardware: [Raspberry Pi board](#) – read [Best Raspberry Pi Starter Kits](#), [MicroSD Card – 16GB Class10](#), and [Power Supply \(5V 2.5A\)](#);

Getting the ESP IP Address

For this project, you can grab any ESP32 web server code presented in this eBook.

We'll use the web server code from previous unit.

- [Click here to download the code.](#)

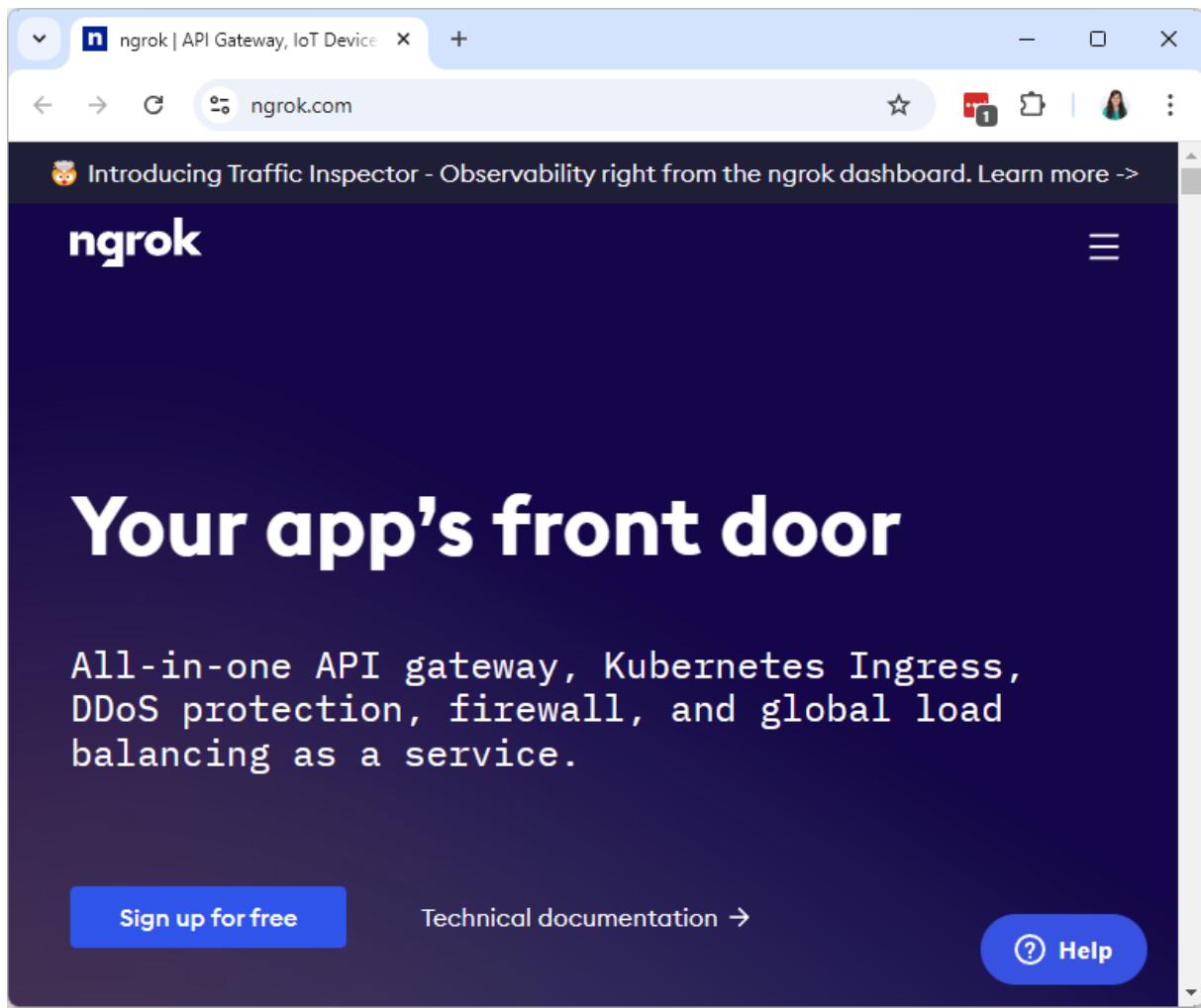
After uploading the code, get the board IP address. Save the IP address (in our case, the IP address is `192.168.1.77`). You'll need it to run ngrok later.

Note: if every time you reset your board, you have a different IP address, make sure to set a static IP address ([take a look at Unit 5.1](#)).

Introducing ngrok

To make your web servers accessible from anywhere, you will use a free service called **ngrok** to create a tunnel to your local web server.

Go to <https://ngrok.com> to create an account.

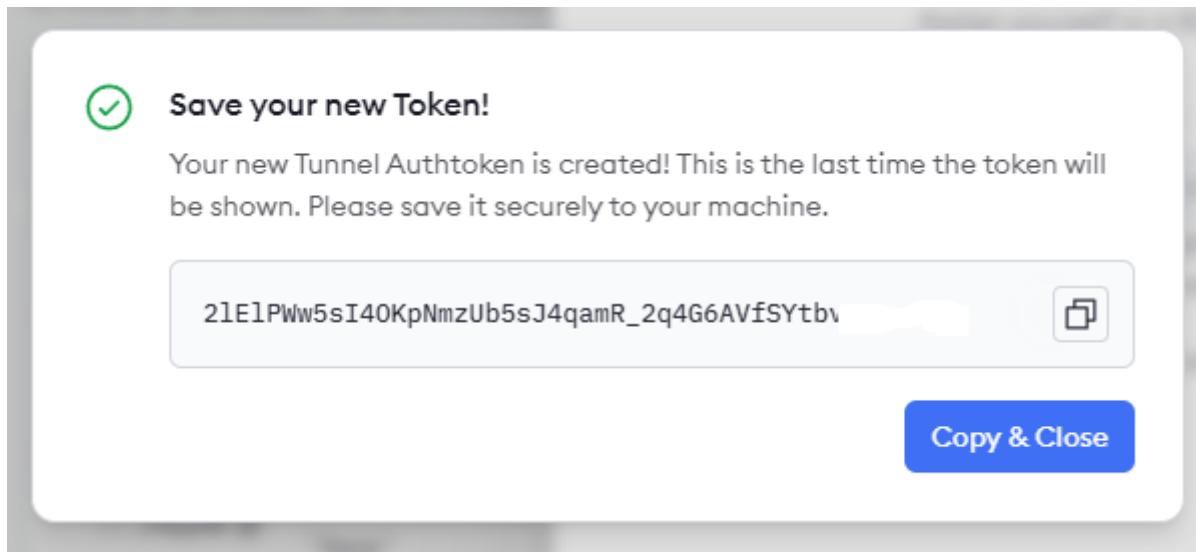


At the left sidebar click on the **Authtoken** tab and then create a new tunnel authtoken by clicking on **+ Add Tunnel Authtoken**.

The screenshot shows the "Tunnel Agent Authtokens" page in the ngrok dashboard. On the left, there's a sidebar with navigation links: "Getting Started", "Cloud Edge", "Tunnels", "Authtokens" (which is highlighted in blue), "Billing", "Usage", "Support", and "Documentation". The main content area has a title "Tunnel Agent Authtokens". It explains that authtokens connect ngrok agents to your account for fine-grained permissions management. Below this, there's a search bar "Filter Authtokens...", a "API Docs" button, and a prominent blue "Add Tunnel Authtoken" button. A table lists existing authtokens:

ID	Description	Owner	Metadata	Created
cr...veRf72	credential for '.....'.com'	Sara	0 bytes	[trash]
cr...J4qamR	Tunnel Authtoken for "Sara"	Sara	0 bytes	[trash]

Make sure to save your tunnel authtoken because you won't be able to see it again.



Installing ngrok in a different Operating System

We'll install ngrok on a Raspberry Pi. If you prefer using ngrok in a different operating system, you can download the installation files for all operating systems at the next link:

- <https://dashboard.ngrok.com/get-started/setup>

The command that makes the ESP32 web server available from anywhere is the same for all operating systems, but we'll use it on a Raspberry Pi.

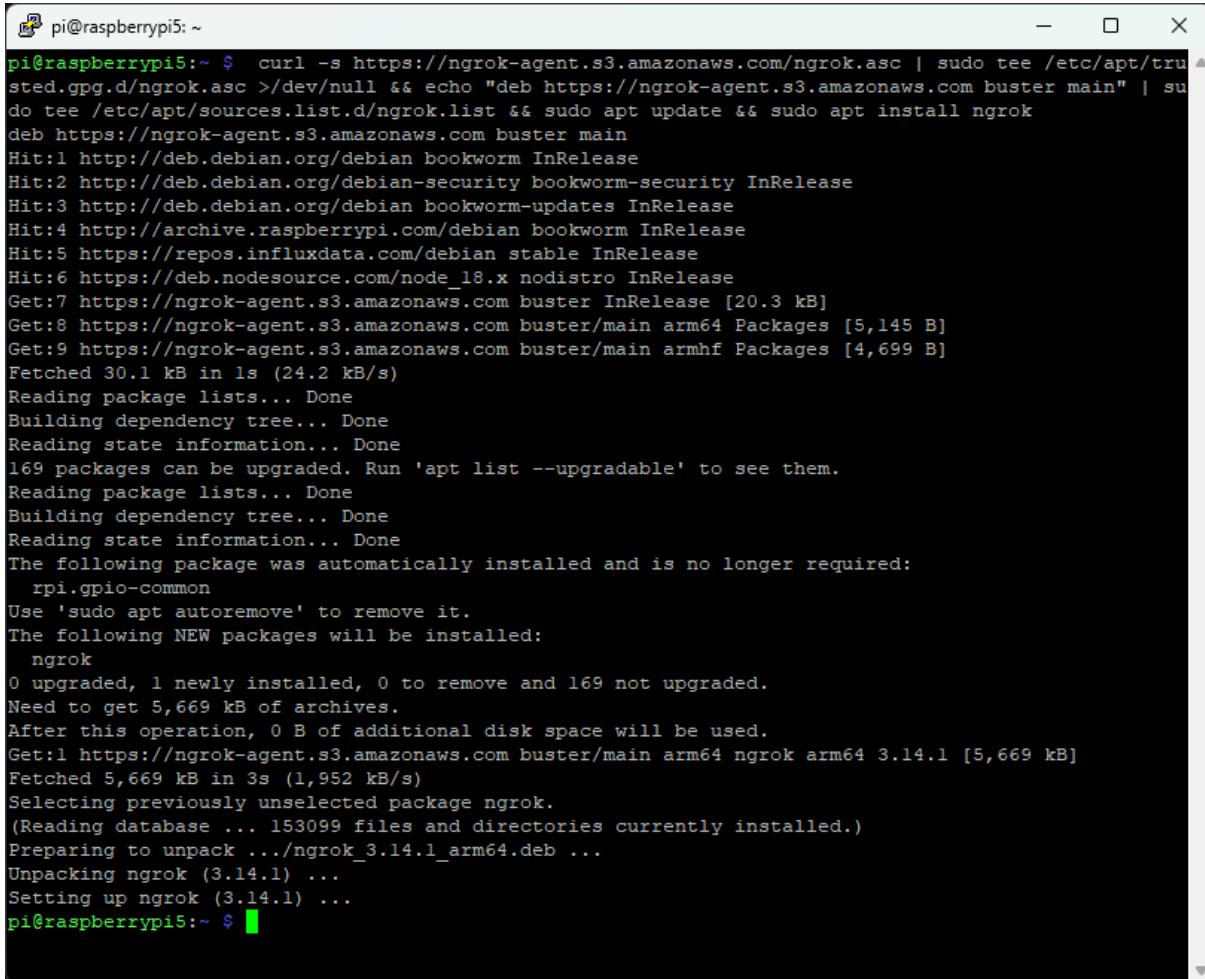
Installing ngrok on Raspberry Pi

Having an SSH communication established with your Raspberry Pi:



Copy and run the following command to download and install the latest version of ngrok software:

```
curl -s https://ngrok-agent.s3.amazonaws.com/ngrok.asc | sudo tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null && echo "deb https://ngrok-agent.s3.amazonaws.com buster main" | sudo tee /etc/apt/sources.list.d/ngrok.list && sudo apt update && sudo apt install ngrok
```



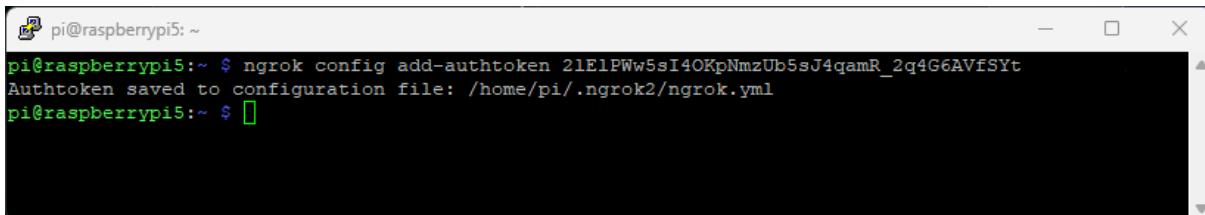
```
pi@raspberrypi5: ~
pi@raspberrypi5:~ $ curl -s https://ngrok-agent.s3.amazonaws.com/ngrok.asc | sudo tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null && echo "deb https://ngrok-agent.s3.amazonaws.com buster main" | sudo tee /etc/apt/sources.list.d/ngrok.list && sudo apt update && sudo apt install ngrok
deb https://ngrok-agent.s3.amazonaws.com buster main
Hit:1 http://deb.debian.org/debian bookworm InRelease
Hit:2 http://deb.debian.org/debian-security bookworm-security InRelease
Hit:3 http://deb.debian.org/debian bookworm-updates InRelease
Hit:4 http://archive.raspberrypi.com/debian bookworm InRelease
Hit:5 https://repos.influxdata.com/debian stable InRelease
Hit:6 https://deb.nodesource.com/node_18.x nodistro InRelease
Get:7 https://ngrok-agent.s3.amazonaws.com buster InRelease [20.3 kB]
Get:8 https://ngrok-agent.s3.amazonaws.com buster/main arm64 Packages [5,145 B]
Get:9 https://ngrok-agent.s3.amazonaws.com buster/main armhf Packages [4,699 B]
Fetched 30.1 kB in 1s (24.2 kB/s)
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
169 packages can be upgraded. Run 'apt list --upgradable' to see them.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following package was automatically installed and is no longer required:
  rpi.gpio-common
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  ngrok
0 upgraded, 1 newly installed, 0 to remove and 169 not upgraded.
Need to get 5,669 kB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 https://ngrok-agent.s3.amazonaws.com buster/main arm64 ngrok arm64 3.14.1 [5,669 kB]
Fetched 5,669 kB in 3s (1,952 kB/s)
Selecting previously unselected package ngrok.
(Reading database ... 153099 files and directories currently installed.)
Preparing to unpack .../ngrok_3.14.1_arm64.deb ...
Unpacking ngrok (3.14.1) ...
Setting up ngrok (3.14.1) ...
pi@raspberrypi5:~ $
```

Note: if you're having issues copying/pasting the command, you can copy it from the official installation instructions. Go to this link and copy the command to install ngrok via apt: <https://ngrok.com/download>

Running ngrok

Enter the following command to authenticate your ngrok account (replace the red highlighted text with your own AuthToken):

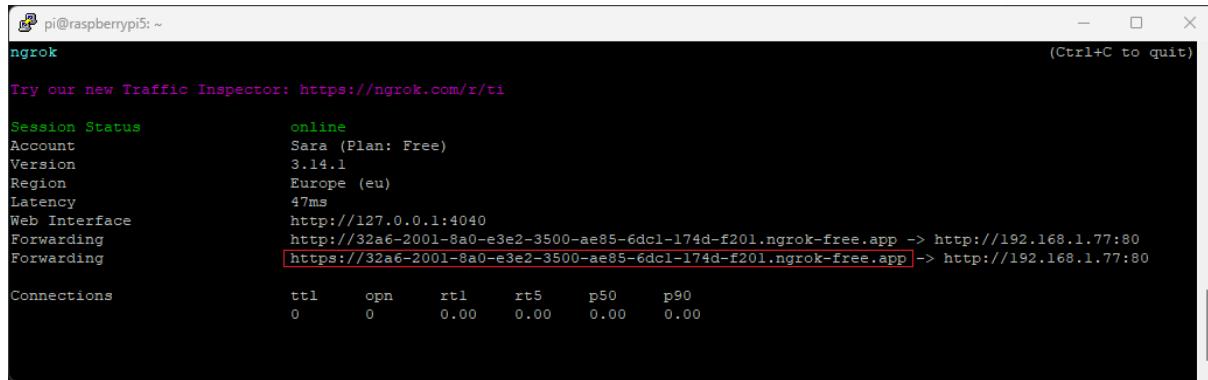
```
ngrok config add-authtoken 21ElPWw5sI4OKpNmzUb5sJ4qamR_2q4G--
```



```
pi@raspberrypi5: ~
pi@raspberrypi5:~ $ ngrok config add-authtoken 21ElPWw5sI4OKpNmzUb5sJ4qamR_2q4G6AVfSYt
Auth token saved to configuration file: /home/pi/.ngrok2/ngrok.yml
pi@raspberrypi5:~ $
```

In your terminal window, enter the following command, but replace the red text with your ESP32 IP address:

```
ngrok http 192.168.1.77:80 --scheme http,https
```



```
pi@raspberrypi5: ~
ngrok
Try our new Traffic Inspector: https://ngrok.com/r/ti

Session Status      online
Account             Sara (Plan: Free)
Version            3.14.1
Region              Europe (eu)
Latency            47ms
Web Interface     http://127.0.0.1:4040
Forwarding          http://32a6-2001-8a0-e3e2-3500-ae85-6dc1-174d-f201.ngrok-free.app -> http://192.168.1.77:80
Forwarding          https://32a6-2001-8a0-e3e2-3500-ae85-6dc1-174d-f201.ngrok-free.app -> http://192.168.1.77:80

Connections        ttl     opn      rtt1     rt5      p50      p90
                      0       0       0.00    0.00    0.00    0.00
```

Copy your unique link (it is highlighted in the preceding figure):

```
https://13bc-2001-8a0-e3e2-3500-ae85-6dc1-174d-f201.ngrok-free.app/
```

Don't press CTRL+C in your Terminal window, otherwise it will quit ngrok. To copy the URL, select it and click with the right side of the mouse. The URL will be copied to the clipboard.

Important: you need to let your computer on and running ngrok in order to maintain the tunnel online.

Accessing Your Web Server from Anywhere

If everything runs smoothly, you can open the ESP web server from any web browser in the world. You just need to enter the URL you've got, as shown below:

```
https://13bc-2001-8a0-e3e2-3500-ae85-6dc1-174d-f201.ngrok-free.app/
```

You'll be asked to enter the username (**admin**) and password you defined in the code (**admin**).

After entering the correct credentials your ESP32 web server loads.

Now, you have full control over your ESP32 from anywhere in the world.



Important: you can close the SSH window, but you can't quit the ngrok software. Otherwise, your tunnel stops. You also need to leave your Raspberry Pi on and running ngrok in order to maintain the tunnel online. If the Raspberry Pi restarts, and the tunnel stops, then, it will generate a new link for you to access the web server. You can check the link on your ngrok account dashboard online under the "Tunnels Agents" tab.

Autostart ngrok on boot

To make ngrok software autostart when the Raspberry Pi first boots, you need to install the screen software that allows you to run commands in the background:

```
sudo apt install screen -y
```

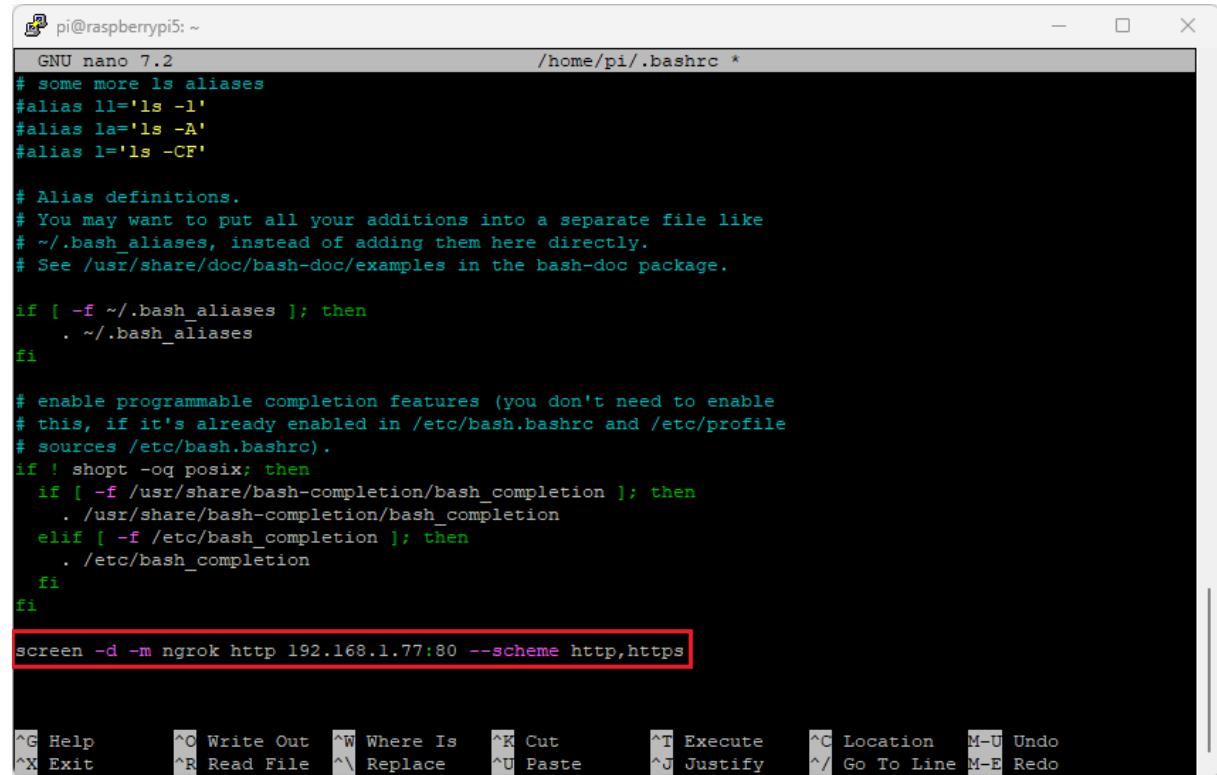
Finally, you need to edit the file `/home/pi/.bashrc` to tell your Raspberry Pi to run ngrok on start. Type the next command in your terminal window:

```
pi@raspberry:~ $ sudo nano /home/pi/.bashrc
```

```
pi@raspberrypi:~ $ sudo nano /home/pi/.bashrc
```

Scroll down to the bottom of the `.bashrc` file and add the following command (but replace with your web server local IP address):

```
screen -d -m ngrok http 192.168.1.77:80 --scheme http,https
```



```
pi@raspberrypi: ~
GNU nano 7.2                               /home/pi/.bashrc *
# some more ls aliases
alias ll='ls -l'
alias la='ls -A'
alias l='ls -CF'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
    if [ -f /usr/share/bash-completion/bash_completion ]; then
        . /usr/share/bash-completion/bash_completion
    elif [ -f /etc/bash_completion ]; then
        . /etc/bash_completion
    fi
fi

screen -d -m ngrok http 192.168.1.77:80 --scheme http,https

^G Help      ^O Write Out   ^W Where Is   ^K Cut          ^T Execute   ^C Location   M-U Undo
^X Exit      ^R Read File   ^\ Replace    ^U Paste       ^J Justify   ^/ Go To Line M-E Redo
```

Press **Ctrl+X**, followed by **Y** and **Enter** to save the file. Then, restart your Raspberry Pi to test if it's auto starting ngrok on boot:

```
sudo reboot
```

Wait a few minutes for your Raspberry Pi to fully start all services. Then, if you go to ngrok Dashboard under the “Agents Tunnels” menu, you should see the new URL that you must access to access your web server.

The screenshot shows the ngrok dashboard at `dashboard.ngrok.com/tunnels/agents/ts_2lF4QuiJRVc4WcWa71ygJ8hZ5Uo`. The left sidebar has a 'Agents' section selected. The main area is titled 'Agent' and displays detailed information about a running agent. A table shows the following data:

	Value
Session ID	ts_2lF4QuiJRVc4WcWa71ygJ8hZ5Uo
Credential ID	cr_2lElPWw5sI40KpNmzUb5sJ4qamR
Region	EU - Europe
Agent Ingress Hostname	connect.ngrok-agent.com
IP	2001:8a0:e3e2:3500:ae85:6dc1:174d:f201
Agent Version	ngrok-agent/3.14.1
User-Agent	ngrok-agent/3.14.1 ngrok-go/1.9.1
OS	linux
Transport	muxado
Started	Aug 27, 2024 1:41 PM
Metadata	0 bytes

Below this, a table lists tunnels:

Tunnel	Forwards To
https://8bdc-2001-8a0-e3e2-3500-ae85-6dc1-174df201.ngrok-free.app	http://192.168.1.77:80
http://8bdc-2001-8a0-e3e2-3500-ae85-6dc1-174df201.ngrok-free.app	http://192.168.1.77:80

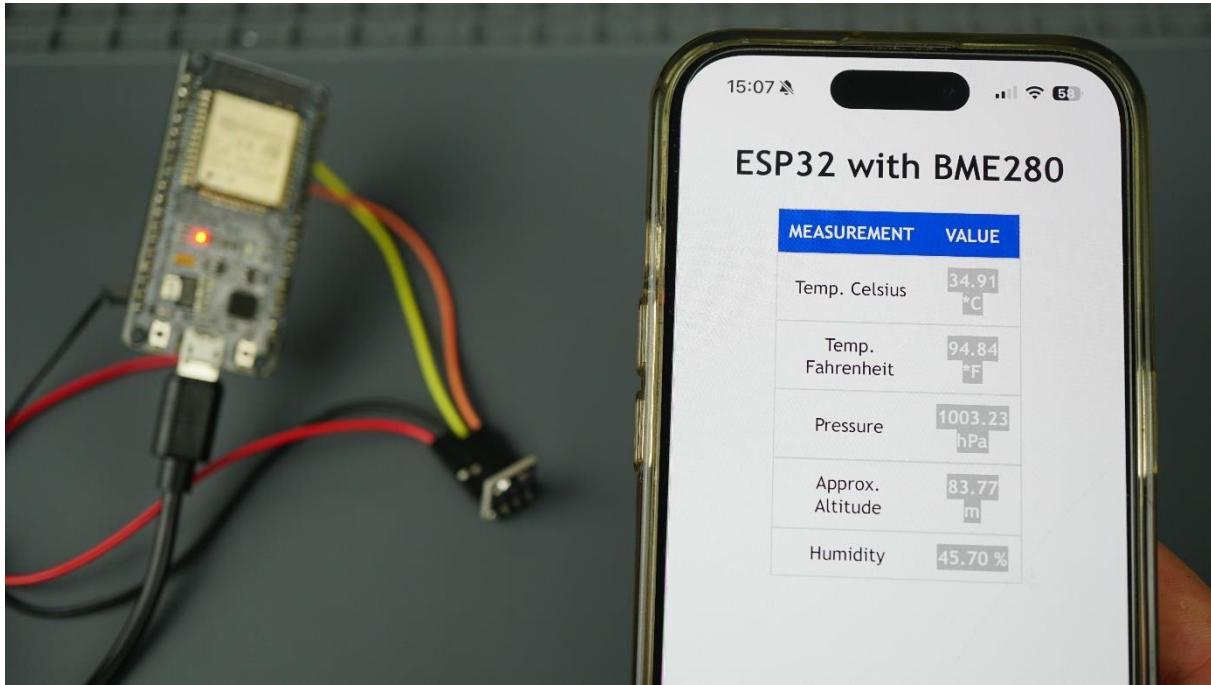
A red box highlights the first tunnel entry.

Tip: If you close the ngrok tunnel, and run it again, it will have a different URL. You can go to your ngrok account and under the **Agents** tab, check the information about the current tunnels and the current URL.

Wrapping Up

In this Unit, you've learned how to make your ESP32 accessible from anywhere. This is an exciting feature as it allows you to control and monitor your board anytime, anywhere. As always, you can apply this concept to any of your other web server projects.

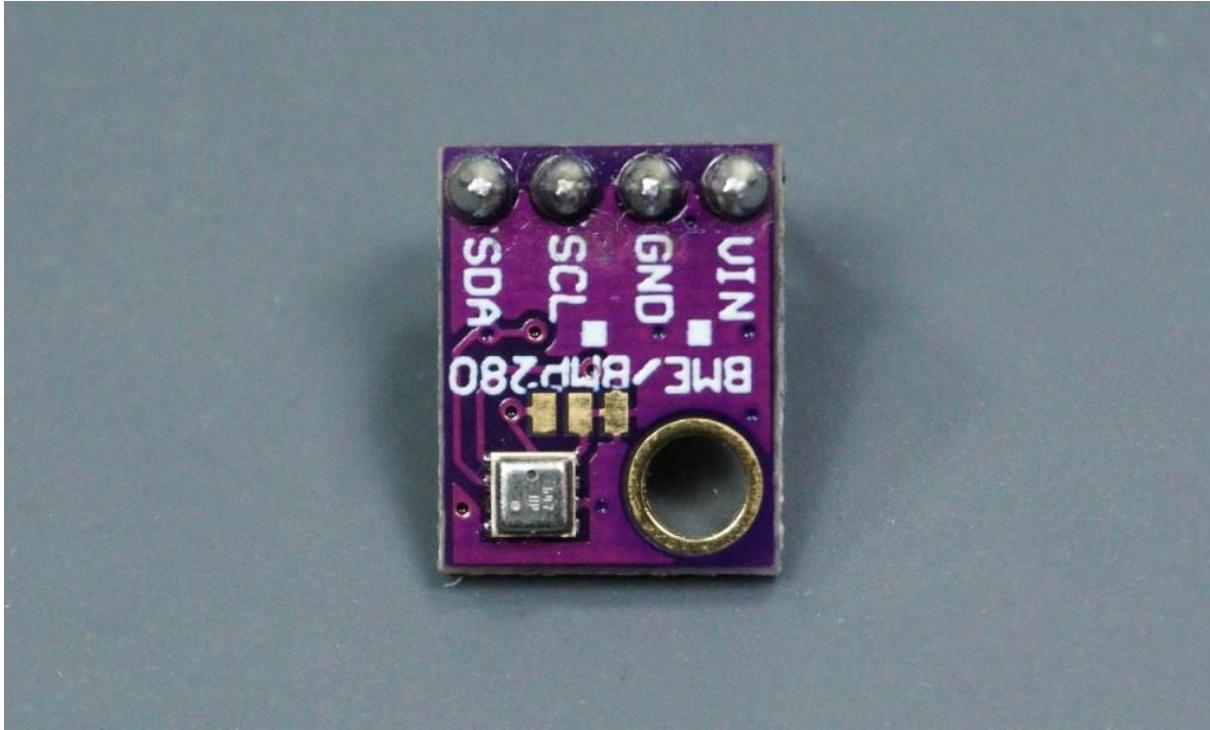
6.6 - ESP32 Web Server – Display Sensor Readings



In this Unit you'll learn how to create a simple web server with the ESP32 to display readings from the BME280 sensor module. This sensor measures temperature, humidity, and pressure. So, you can easily build a mini and compact weather station and monitor the results using your ESP32 web server. That's what we're going to do in this project.

Introducing the BME280 Sensor Module

The BME280 sensor module reads temperature, humidity, and pressure. Because pressure changes with altitude, you can also estimate altitude. There are several versions of this sensor module, but we're using the one shown in the figure below.



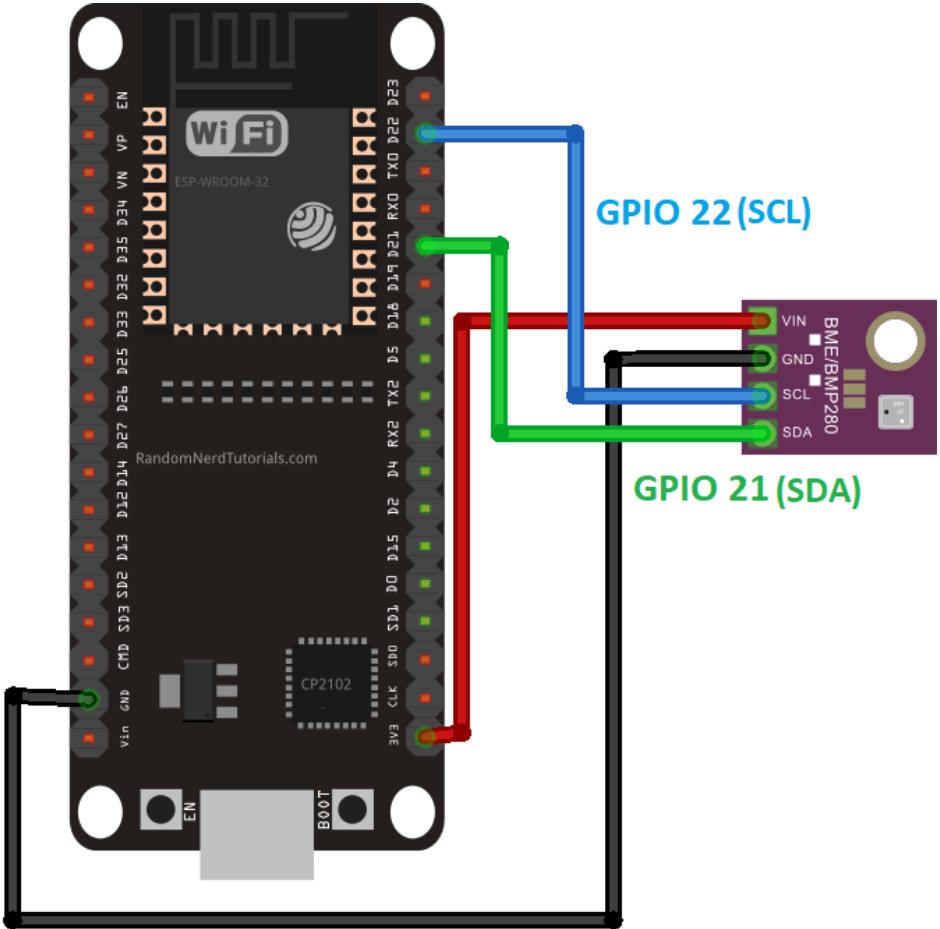
This sensor communicates using I2C communication protocol, so the wiring is very simple. Use your ESP32 board default I2C pins.

Wiring the Circuit

Wire the sensor to the ESP32 SDA and SCL pins, as shown in the following schematic diagram. For the ESP32 Devkit board, the default pins are GPIO 21 (SDA) and GPIO 22 (SCL). Double-check the default pins for the board you're using.

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [BME280 sensor module](#)
- [Breadboard](#)
- [Jumper wires](#)
- [Schematics](#)



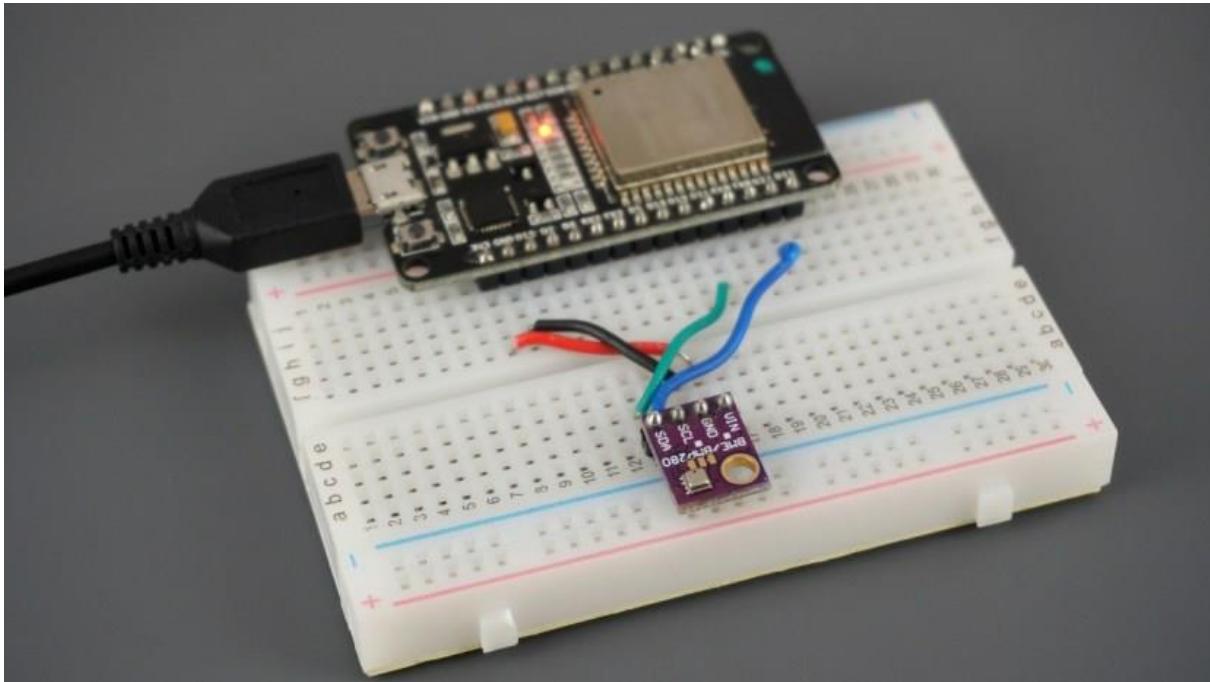
Installing the BME280 and Adafruit_Sensor libraries

To get readings from the BME280 sensor module, we'll use the [Adafruit BME280 library](#). Follow the next steps to install the library in your Arduino IDE:

- 1- Go Sketch ▶ Include Library ▶ Manage Libraries, and type “adafruit bme280” to search for the library. Then, click install.
- 2- A pop-up window will open asking you to install the Adafruit Unified Sensor library. Install that library and any other dependencies.

Reading Temperature, Humidity, and Pressure

To get familiar with the BME280 sensor, we'll start with a basic example sketch that reads temperature, humidity, and pressure and prints them on the Serial Monitor.



After installing the required libraries and wiring the circuit, you can upload the following code to your board.

- [Click here to download the code.](#)

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

#define SEALEVELPRESSURE_HPA (1013.25)

Adafruit_BME280 bme; // I2C

unsigned long delayTime;

void setup() {
    Serial.begin(115200);
    Serial.println("BME280 test");

    bool status;
    status = bme.begin(0x76);
    if (!status) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }

    Serial.println("-- Default Test --");
    delayTime = 1000;

    Serial.println();
}

void loop() {
    printValues();
    delay(delayTime);
}
```

```
void printValues() {
    Serial.print("Temperature = ");
    Serial.print(bme.readTemperature());
    Serial.println(" *C");

    // Convert temperature to Fahrenheit
    /*Serial.print("Temperature = ");
    Serial.print(1.8 * bme.readTemperature() + 32);
    Serial.println(" *F");*/

    Serial.print("Pressure = ");

    Serial.print(bme.readPressure() / 100.0F);
    Serial.println(" hPa");

    Serial.print("Approx. Altitude = ");
    Serial.print(bme.readAltitude(SEALEVELPRESSURE_HPA));
    Serial.println(" m");

    Serial.print("Humidity = ");
    Serial.print(bme.readHumidity());
    Serial.println(" %");

    Serial.println();
}
```

Libraries

The code starts by including the needed libraries.

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
```

Sea level pressure

A variable called `SEALEVELPRESSURE_HPA` is created.

```
#define SEALEVELPRESSURE_HPA (1013.25)
```

This variable saves the pressure at the sea level in hectopascal (is equivalent to millibar). It is used to estimate the altitude for a given pressure by comparing it with the sea level pressure. This example uses the default value, but for more accurate results, replace the value with the current sea level pressure at your location.

I2C

This example uses I2C communication by default. As you can see, you just need to create an `Adafruit_BME280` object called `bme` and it will initialize it on your board's default I2C pins.

```
Adafruit_BME280 bme; // I2C
```

setup()

In the `setup()` you start a serial communication.

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

And the sensor is initialized:

```
bool status;
status = bme.begin(0x76);
if (!status) {
    Serial.println("Could not find a valid BME280 sensor, check wiring!");
    while (1);
}
```

Printing values

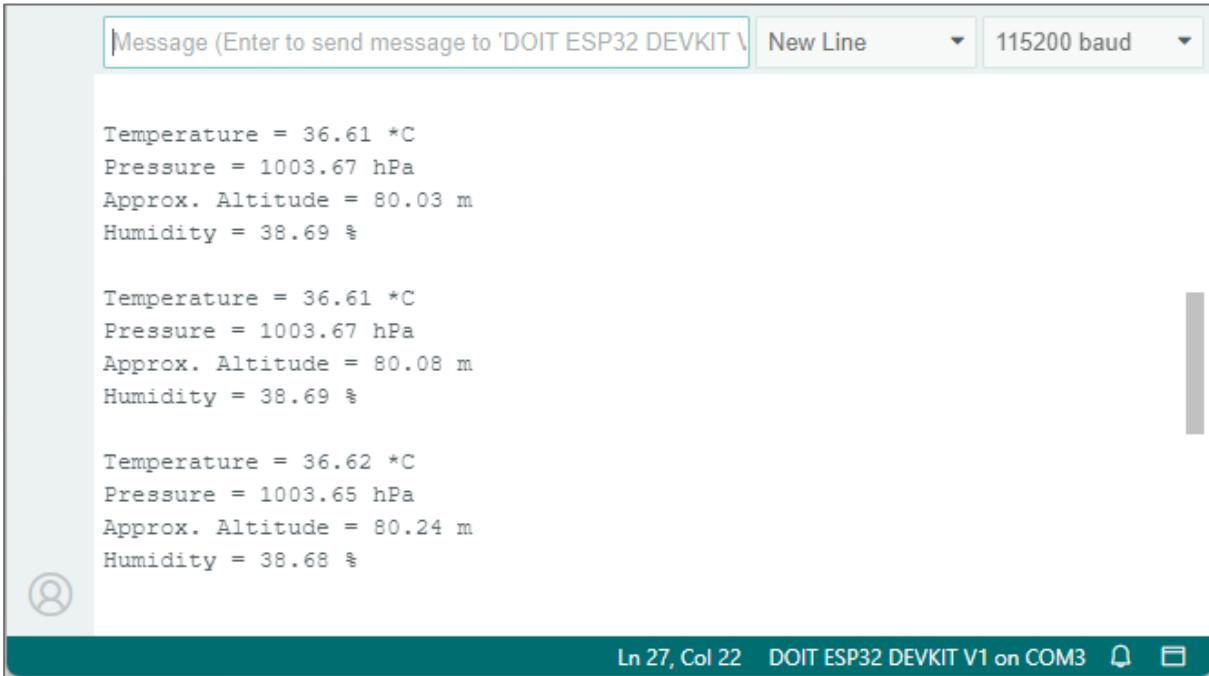
In the `loop()`, the `printValues()` function reads the values from the BME280 and prints the results in the Serial Monitor.

```
void loop() {
    printValues();
    delay(delayTime);
}
```

Reading temperature, humidity, pressure, and estimated altitude is as simple as using:

- `bme.readTemperature()` – reads temperature in Celsius;
- `bme.readHumidity()` – reads absolute humidity;
- `bme.readPressure()` – reads pressure in hPa (hectoPascal = millibar);
- `bme.readAltitude(SEALEVELPRESSURE_HPA)` – estimates altitude in meters based on the pressure at the sea level.

Upload the code to your ESP32, and open the serial monitor at a baud rate of 115200. You should see the readings displayed on the serial monitor.



The screenshot shows a serial monitor window with the following interface elements:

- Top bar: "Message (Enter to send message to 'DOIT ESP32 DEVKIT V1')", "New Line", and "115200 baud".
- Main area: Three sets of sensor data:
 - Temperature = 36.61 *C
Pressure = 1003.67 hPa
Approx. Altitude = 80.03 m
Humidity = 38.69 %
 - Temperature = 36.61 *C
Pressure = 1003.67 hPa
Approx. Altitude = 80.08 m
Humidity = 38.69 %
 - Temperature = 36.62 *C
Pressure = 1003.65 hPa
Approx. Altitude = 80.24 m
Humidity = 38.68 %
- Bottom bar: "Ln 27, Col 22" and "DOIT ESP32 DEVKIT V1 on COM3" along with icons for message, settings, and close.

Creating a Table in HTML

In this example, we're displaying the BME280 sensor readings on a table. So, we need to write HTML text to build a table.

To create a table in HTML you use the `<table>` and `</table>` tags.

To create a row, you use the `<tr>` and `</tr>` tags. The table heading is defined using the `<th>` and `</th>` tags, and each table cell is defined using the `<td>` and `</td>` tags.

To create a table for our readings, you use the following html text:

```
<table>
  <tr>
    <th>MEASUREMENT</th>
    <th>VALUE</th>
  </tr>
  <tr>
    <td>Temp. Celsius</td>
    <td>--- *C</td>
  </tr>
  <tr>
    <td>Temp. Fahrenheit</td>
    <td>--- *F</td>
  </tr>
  <tr>
    <td>Pressure</td>
    <td>--- hPa</td>
  </tr>
  <tr>
    <td>Approx. Altitude</td>
```

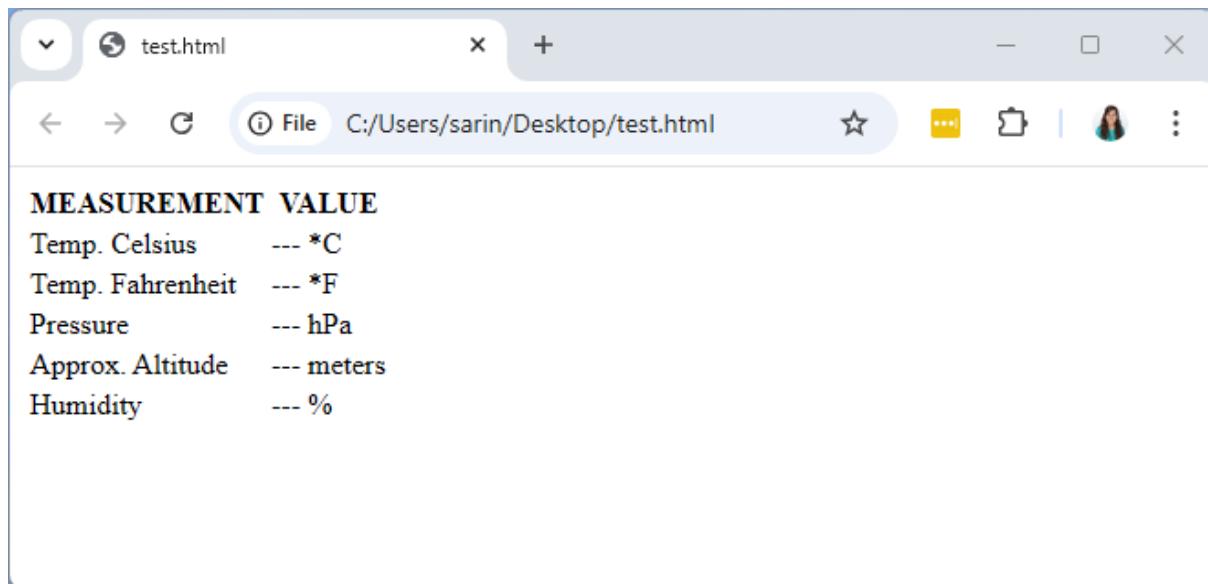
```
<td>--- meters</td></tr>
<tr>
  <td>Humidity</td>
  <td>--- %</td>
</tr>
</table>
```

We create the header of the table with a cell called **MEASUREMENT**, and another named **VALUE**.

Then, we create six rows to display each of the readings using the **<tr>** and **</tr>** tags.

Inside each row, we create two cells, using the **<td>** and **</td>** tags, one with the name of the measurement and another to hold the measurement value. The three dashes “---” should then be replaced with the actual measurements from the BME280 sensor.

You can save this text as *table.html*, drag the file into your browser and see what you have. The previous HTML text creates the following table.



The table doesn't have any applied styles. You can use CSS to style the table. You can read the tutorial in the following link to learn how to style a table: [CSS Styling Tables](#).

Creating the Web Server

Now that you know how to get readings from the sensor and build a table to display the results, it's time to make the web server. If you've followed the previous Units, you should be familiar with most lines of code.

Copy the following code to your Arduino IDE. Before uploading, you need to insert your SSID and password.

- [Click here to download the code.](#)

```
#include <WiFi.h>
#include <Wire.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
#include <WebServer.h>

#define SEALEVELPRESSURE_HPA (1013.25)

Adafruit_BME280 bme; // I2C

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

// Create an instance of the WebServer on port 80
WebServer server(80);

void handleRoot() {
    String html = "<!DOCTYPE html><html>";
    html += "<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">";
    html += "<link rel=\"icon\" href=\"data:,\">";
    html += "<style>body { text-align: center;
        font-family: \"Trebuchet MS\", Arial;}";
    html += "table { border-collapse: collapse; width:40%;
        margin-left:auto; margin-right:auto; }";
    html += "th { padding: 12px; background-color: #0043af; color: white; }";
    html += "tr { border: 1px solid #ddd; padding: 12px; }";
    html += "tr:hover { background-color: #bcbcbc; }";
    html += "td { border: none; padding: 12px; }";
    html += ".sensor { color:white; font-weight: bold;
        background-color: #bcbcbc; padding: 1px; }</style></head>";
    html += "<body><h1>ESP32 with BME280</h1>";
    html += "<table><tr><th>MEASUREMENT</th><th>VALUE</th></tr>";
    html += "<tr><td>Temp. Celsius</td><td><span class=\"sensor\">"; 
    html += String(bme.readTemperature());
    html += " *C</span></td></tr>";
    html += "<tr><td>Temp. Fahrenheit</td><td><span class=\"sensor\">"; 
    html += String(1.8 * bme.readTemperature() + 32);
    html += " *F</span></td></tr>";
    html += "<tr><td>Pressure</td><td><span class=\"sensor\">"; 
    html += String(bme.readPressure() / 100.0F);
    html += " hPa</span></td></tr>";
    html += "<tr><td>Approx. Altitude</td><td><span class=\"sensor\">";
```

```

html += String(bme.readAltitude(SEALEVELPRESSURE_HPA));
html += " m</span></td></tr>";
html += "<tr><td>Humidity</td><td><span class=\"sensor\">";
html += String(bme.readHumidity());
html += " %</span></td></tr></table></body></html>";

// Send the response to the client
server.send(200, "text/html", html);
}

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

  // Initialize the BME280 sensor
  if (!bme.begin(0x76)) {
    Serial.println("Could not find a valid BME280 sensor, check wiring!");
    while (1);
  }

  // Connect to Wi-Fi
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  // Set up the routes
  server.on("/", handleRoot);

  // Start the server
  server.begin();
}

void loop() {
  server.handleClient();
}

```

Modify the following lines to include your SSID and password.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

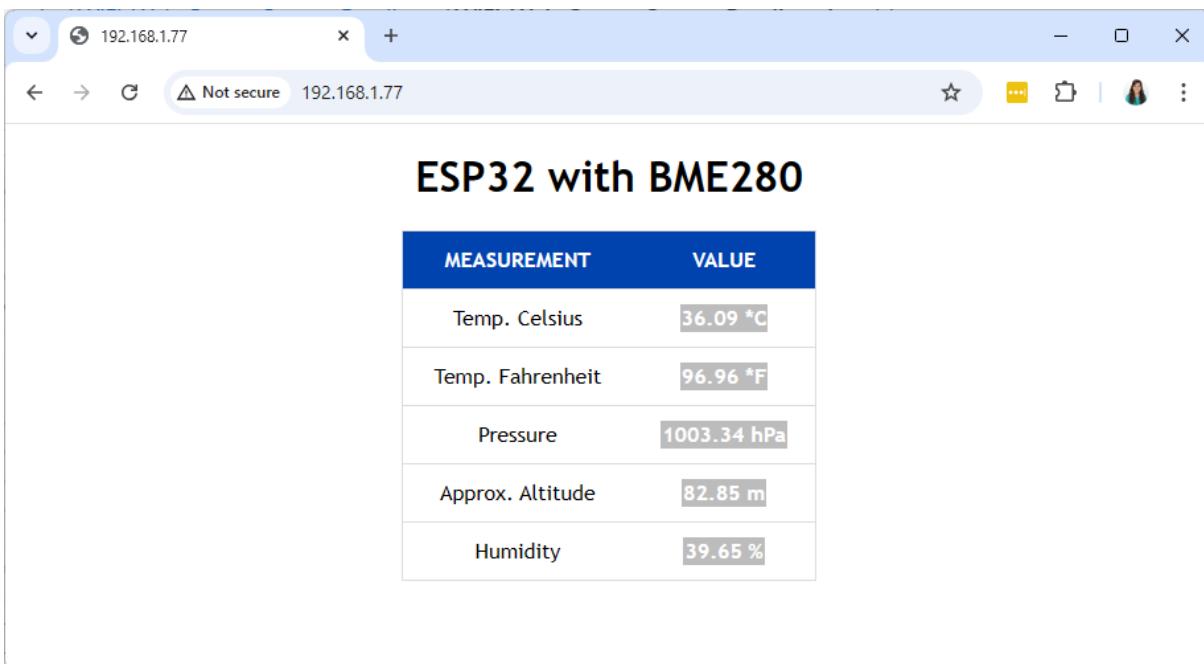
Then, check that you have the right board and COM port selected, and upload the code to your ESP32. After uploading, open the serial monitor at a baud rate of 115200, and copy the ESP32 IP address.

The screenshot shows the Arduino Serial Monitor window. At the top, there are tabs for 'Output' and 'Serial Monitor'. Below the tabs, a message input field contains 'Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'COM3')'. To the right of the input field are buttons for 'New Line' and '115200 baud'. The main text area displays the following log output:

```
louuu.uuuuuuuu, luuu... luu...
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Connecting to MEO-
.
WiFi connected.
IP address:
192.168.1.77
```

At the bottom of the window, it says 'Ln 8, Col 1 DOIT ESP32 DEVKIT V1 on COM3' and has icons for a serial port and a refresh button.

Open your browser, paste the IP address, and you should see the latest sensor readings. To update the readings, you just need to refresh the web page.



How the code works

This sketch is very similar to the sketch used in Unit 6.2. You must be familiar with most parts of the code. First, you include the `WiFi` and `WebServer` libraries to create the web server and the needed libraries to read from the BME280 sensor.

```
#include <WiFi.h>
#include <Wire.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
#include <WebServer.h>
```

The following line defines a variable to save the pressure at the sea level. For more accurate altitude estimation, replace the value with the current sea level pressure at your location.

```
#define SEALEVELPRESSURE_HPA (1013.25)
```

In the following line, you create an `Adafruit_BME280` object called `bme` that by default establishes a communication with the sensor using I2C.

```
Adafruit_BME280 bme; // I2C
```

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

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

Then, initialize a server on port 80.

```
WebServer server(80);
```

setup()

In the `setup()`, we start a serial communication at a baud rate of 115200 for debugging purposes.

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

You check that the BME280 sensor was successfully initialized.

```
if (!bme.begin(0x76)) {
    Serial.println("Could not find a valid BME280 sensor, check wiring!");
    while (1);
}
```

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
Serial.print("Connecting to ");
Serial.println(ssid);
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
}
Serial.println("");
```

```
Serial.println("WiFi connected.");
Serial.println("IP address: ");
Serial.println(WiFi.localIP());
```

Finally, we set up the web server and define the routes it should handle. This web server will just handle the root URL to display the latest sensor readings. When you access the root URL, it will call the `handleRoot()` function.

```
// Set up the routes
server.on("/", handleRoot);

// Start the server
server.begin();
```

Displaying the Web Page

The `handleRoot()` function is responsible for generating the web page. It sends the HTML and CSS needed to build the page. The HTML and CSS text required to build the web page are saved on the `html` variable. We concatenate the current sensor readings with the `html` variable. This way, the HTML sent to the web browser will contain the latest sensor readings.

```
void handleRoot() {
    String html = "<!DOCTYPE html><html>";
    html += "<head><meta name=\"viewport\""
           content=\"width=device-width, initial-scale=1\">";
    html += "<link rel=\"icon\" href=\"data:,\">";
    html += "<style>body { text-align: center;
                  font-family: \"Trebuchet MS\", Arial;}";
    html += "table { border-collapse: collapse; width:50%;
                 margin-left:auto; margin-right:auto; }";
    html += "th { padding: 10px; background-color: #0043af; color: white; }";
    html += "tr { border: 1px solid #ddd; padding: 12px; }";
    html += "tr:hover { background-color: #bcbcbc; }";
    html += "td { border: none; padding: 10px; }";
    html += ".sensor { color:white; font-weight: bold;
                  background-color: #bcbcbc; padding: 1px; }</style></head>";
    html += "<body><h1>ESP32 with BME280</h1>";
    html += "<table><tr><th>MEASUREMENT</th><th>VALUE</th></tr>";
    html += "<tr><td>Temp. Celsius</td><td><span class=\"sensor\">";
    html += String(bme.readTemperature());
    html += " *C</span></td></tr>";
    html += "<tr><td>Temp. Fahrenheit</td><td><span class=\"sensor\">";
    html += String(1.8 * bme.readTemperature() + 32);
    html += " *F</span></td></tr>";
    html += "<tr><td>Pressure</td><td><span class=\"sensor\">";
    html += String(bme.readPressure() / 100.0F);
    html += " hPa</span></td></tr>";
    html += "<tr><td>Approx. Altitude</td><td><span class=\"sensor\">";
    html += String(bme.readAltitude(SEALEVELPRESSURE_HPA));
    html += " m</span></td></tr>";
    html += "<tr><td>Humidity</td><td><span class=\"sensor\">";
    html += String(bme.readHumidity());
```

```

html += " %</span></td></tr></table></body></html>";

// Send the response to the client
server.send(200, "text/html", html);
}

```

To display the sensor readings on the table, we just need to send them between the corresponding `<td>` and `</td>` tags. For example, to display the temperature:

```

html += "<tr><td>Temp. Celsius</td><td><span class=\"sensor\">";
html += String(bme.readTemperature());
html += " *C</span></td></tr>";

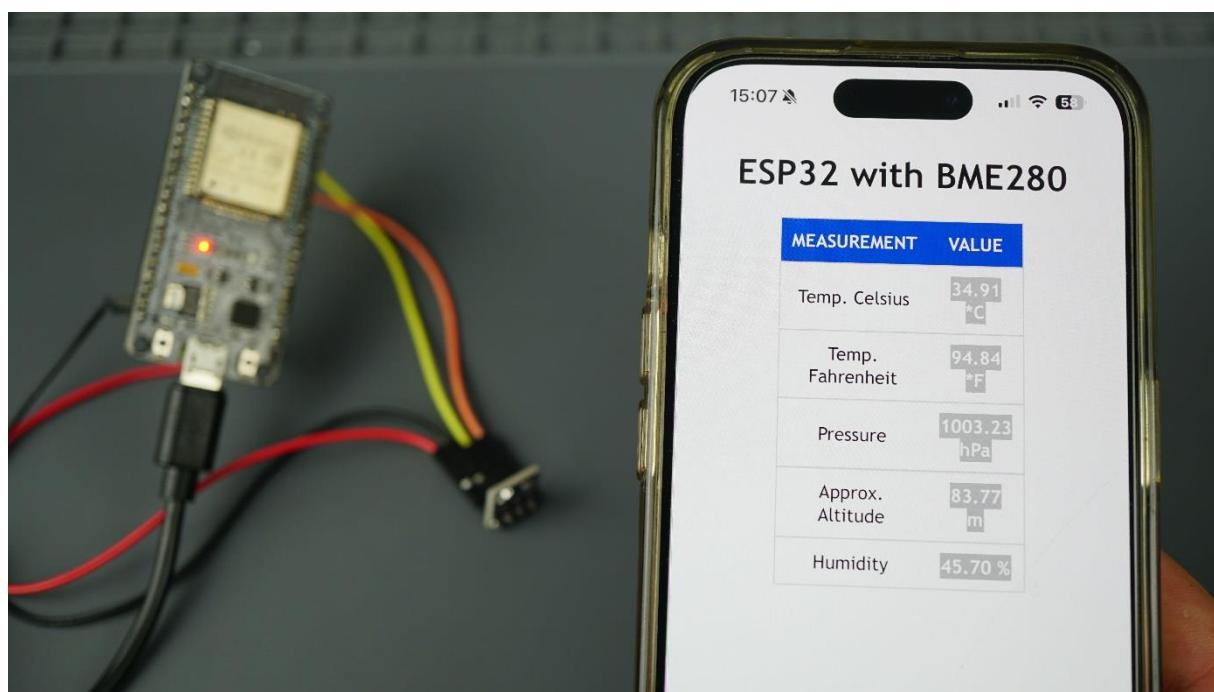
```

Note: the `` tag is useful to style a particular part of a text. In this case, we're using the `` tag to include the sensor reading in a class called "sensor". This is useful to style that particular part of text using CSS.

By default, the table displays the temperature readings in both Celsius degrees and Fahrenheit.

Wrapping Up

In summary, you've learned how to read temperature, humidity, and pressure, and estimate altitude using the BME280 sensor module. You also learned how to build a web server that displays a table with sensor readings. You can easily modify this project to display data from any other sensor.



6.7 - Asynchronous Web Server: Temperature and Humidity Readings



In this project, we'll show you how to build an asynchronous web server that displays temperature and humidity from DHT11 or DHT22 sensors. The web server we'll build updates the readings automatically without the need to refresh the web page.

With this project, you'll learn:

- How to read temperature and humidity from DHT sensors (DHT11 and DHT22);
- Build an asynchronous web server using the [ESPAsyncWebServer library](#);
- Update the sensor readings automatically without the need to refresh the web page.

Asynchronous Web Server

To build the web server, we'll use the [ESPAsyncWebServer library](#) that provides an easy way to create an asynchronous web server. Building an asynchronous web server has several advantages, as mentioned in the library GitHub page, such as:

- “Handle more than one connection at the same time”;
- “When you send the response, you are immediately ready to handle other connections while the server is taking care of sending the response in the background”;
- “Simple template processing engine to handle templates”;

And much more. Take a look at the [library documentation on its GitHub page](#).

Wiring the Circuit

Before proceeding to the web server, you need to wire the DHT11 or DHT22 sensor to the ESP32.

Parts required

Here's a list of the parts required to follow this project:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [DHT11](#) or [DHT22](#) Temperature and Humidity Sensor
- [4.7kOhm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)

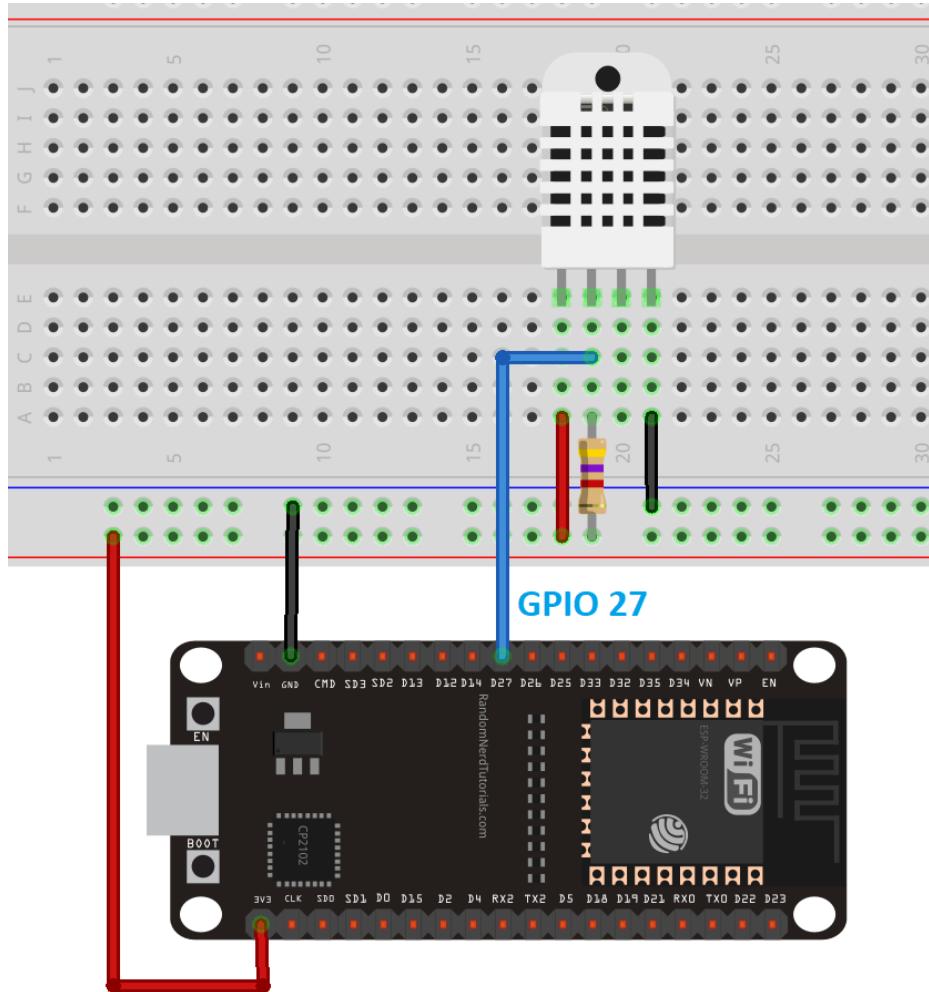
To wire the circuit, follow the next schematic diagram.

In this case, we're connecting the data pin to GPIO 27, but you can connect it to any other digital pin.

Note: you can follow the same schematic diagram for the DHT11 sensor.

Note: there are DHT11 and DHT22 sensor modules that only come with three pins.

Those sensors already have the 4.7kOhm resistor built-in. So, if you have one of those sensor modules, you don't need to connect the resistor.



Installing Libraries

You need to install the following libraries for this project:

- The [DHT](#) and the [Adafruit Unified Sensor Driver](#) libraries to read from the DHT sensor.
- [ESPAsyncWebServer](#) and [Async_TCP](#) libraries to build the asynchronous web server.

Follow the next instructions to install those libraries:

- 1- Go **Sketch > Include Library > Manage Libraries**, and type “**DHT sensor library**” to search for the library. Then install the DHT sensor library by Adafruit.
- 2- A pop-up window will open asking you to install the Adafruit Unified Sensor library. Install that library if you haven’t already.

The **ESPAsyncWebServer** and **AsyncTCP** libraries aren't available to install through the Arduino Library Manager. So, you need to click the following links to download the library files.

1. [Click here to download the ESPAsyncWebServer library.](#)
2. [Click here to download the Async TCP library.](#)

In your Arduino IDE, go to **Sketch > Include Library > Add .zip Library** and select the libraries you've just downloaded.

Code

Open your Arduino IDE and copy the following code.

- [Click here to download the code.](#)

```
// Import required libraries
#include "WiFi.h"
#include "ESPAsyncWebServer.h"
#include <Adafruit_Sensor.h>
#include <DHT.h>

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

#define DHTPIN 27      // Digital pin connected to the DHT sensor

// Uncomment the type of sensor in use:
// #define DHTTYPE    DHT11      // DHT 11
#define DHTTYPE    DHT22      // DHT 22 (AM2302)
// #define DHTTYPE    DHT21      // DHT 21 (AM2301)

DHT dht(DHTPIN, DHTTYPE);

// Create AsyncWebServer object on port 80
AsyncWebServer server(80);

String readDHTTemperature() {
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
    // Read temperature as Celsius (the default)
    float t = dht.readTemperature();
    // Read temperature as Fahrenheit (isFahrenheit = true)
    // float t = dht.readTemperature(true);
    // Check if any reads failed and exit early (to try again).
    if (isnan(t)) {
        Serial.println("Failed to read from DHT sensor!");
        return "--";
    }
    else {
        Serial.println(t);
        return String(t);
    }
}
```

```

    }

}

String readDTHumidity() {
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
    float h = dht.readHumidity();
    if (isnan(h)) {
        Serial.println("Failed to read from DHT sensor!");
        return "--";
    }
    else {
        Serial.println(h);
        return String(h);
    }
}

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE HTML><html>
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script src="https://kit.fontawesome.com/0294e3a09e.js"
    crossorigin="anonymous"></script>

    <style>
        html {
            font-family: Arial;
            display: inline-block;
            margin: 0px auto;
            text-align: center;
        }
        h2 { font-size: 3.0rem; }
        p { font-size: 3.0rem; }
        .units { font-size: 1.2rem; }
        .dht-labels{
            font-size: 1.5rem;
            vertical-align:middle;
            padding-bottom: 15px;
        }
    </style>
</head>
<body>
    <h2>ESP32 DHT Server</h2>
    <p>
        <i class="fas fa-thermometer-half" style="color:#059e8a;"></i>
        <span class="dht-labels">Temperature</span>
        <span id="temperature">%TEMPERATURE%</span>
        <sup class="units">&deg;C</sup>
    </p>
    <p>
        <i class="fas fa-tint" style="color:#00add6;"></i>
        <span class="dht-labels">Humidity</span>
        <span id="humidity">%HUMIDITY%</span>
        <sup class="units">&percnt;</sup>
    </p>
</body>
<script>
setInterval(function ( ) {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("temperature").innerHTML = this.responseText;
        }
    }
}
);

```

```

        }
    };
    xhttp.open("GET", "/temperature", true);
    xhttp.send();
}, 10000 ) ;

setInterval(function () {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("humidity").innerHTML = this.responseText;
        }
    };
    xhttp.open("GET", "/humidity", true);
    xhttp.send();
}, 10000 ) ;
</script>
</html>)rawliteral";

// Replaces placeholder with DHT values
String processor(const String& var){
    //Serial.println(var);
    if(var == "TEMPERATURE"){
        return readDHTTemperature();
    }
    else if(var == "HUMIDITY"){
        return readDHTHumidity();
    }
    return String();
}

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

    dht.begin();

    // Connect to Wi-Fi
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.println("Connecting to WiFi..");
    }

    // Print ESP32 Local IP Address
    Serial.println(WiFi.localIP());

    // Route for root / web page
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send_P(200, "text/html", index_html, processor);
    });
    server.on("/temperature", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send_P(200, "text/plain", readDHTTemperature().c_str());
    });
    server.on("/humidity", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send_P(200, "text/plain", readDHTHumidity().c_str());
    });

    // Start server
    server.begin();
}

```

```
void loop(){  
}
```

Insert your network credentials in the following variables, and the code will work straight away.

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

How Does the Code Work

In the following paragraphs, we'll explain how the code works. Keep reading if you want to learn more or jump to the Demonstration section to see the final result.

Importing libraries

First, import the required libraries. The WiFi, ESPAsyncWebServer, and the ESPAsyncTCP are needed to build the web server.

The Adafruit_Sensor and the DHT libraries are needed to read from the DHT11 or DHT22 sensors.

```
#include "WiFi.h"  
#include "ESPAsyncWebServer.h"  
#include <Adafruit_Sensor.h>  
#include <DHT.h>
```

Setting your network credentials

Insert your network credentials in the following variables so that the ESP32 can connect to your local network.

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

Variables definition

Define the GPIO that the DHT data pin is connected to. In this case, it's connected to GPIO 27.

```
#define DHTPIN 27 // Digital pin connected to the DHT sensor
```

Then, select the DHT sensor type you're using. In our example, we're using the DHT22. If you're using another type, you just need to uncomment your sensor and comment all the others.

```
#define DHTTYPE DHT22 // DHT 22 (AM2302)
```

Instantiate a `DHT` object with the type and pin we've defined earlier.

```
DHT dht(DHTPIN, DHTTYPE);
```

Create An `AsyncWebServer` object on port 80.

```
AsyncWebServer server(80);
```

Read Temperature and Humidity Functions

We've created two functions: one to read the temperature `readDHTTemperature()` and the other to read humidity `readDHTHumidity()`. Below is the snippet that reads temperature.

```
String readDHTTemperature() {
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
    // Read temperature as Celsius (the default)
    float t = dht.readTemperature();
    // Read temperature as Fahrenheit (isFahrenheit = true)
    //float t = dht.readTemperature(true);
    // Check if any reads failed and exit early (to try again).
    if (isnan(t)) {
        Serial.println("Failed to read from DHT sensor!");
        return "--";
    }
    else {
        Serial.println(t);
        return String(t);
    }
}
```

Getting sensor readings is as simple as using the `readTemperature()` and `readHumidity()` methods on the `dht` object.

```
float t = dht.readTemperature();
```

```
float h = dht.readHumidity();
```

We also have a condition that returns two dashes (--) if the sensor fails to get the readings.

```
if (isnan(t)) {
    Serial.println("Failed to read from DHT sensor!");
    return "--";
}
```

The readings are returned as string type. To convert a float to a string, use the `String()` function.

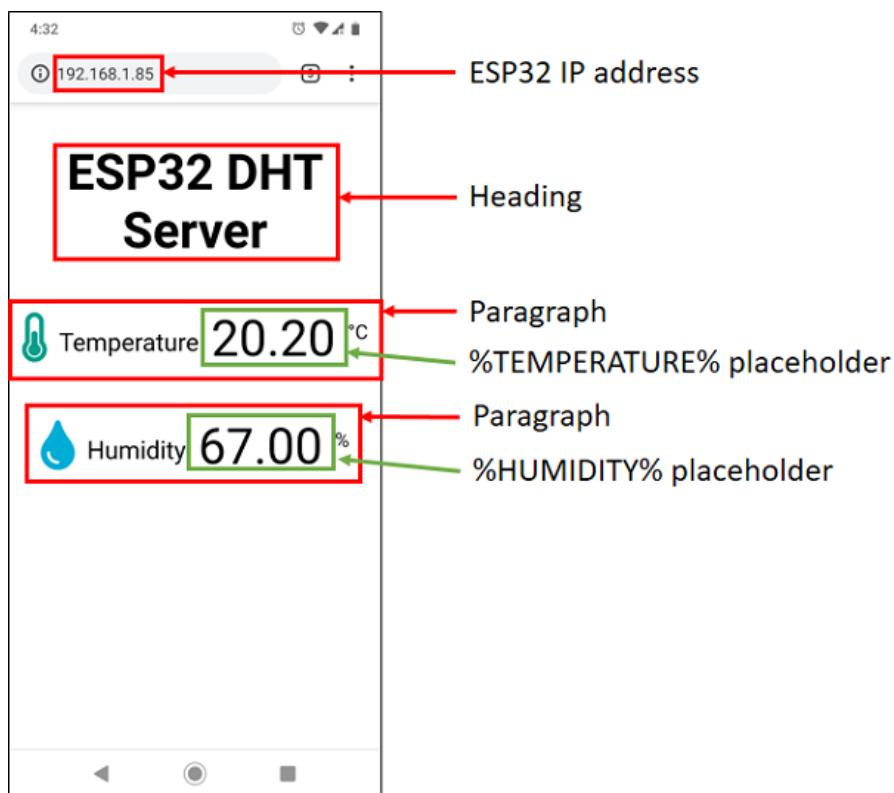
```
return String(t);
```

By default, we're reading the temperature in Celsius degrees. To get the temperature in Fahrenheit degrees, comment the temperature in Celsius and uncomment the temperature in Fahrenheit so that you have the following:

```
//float t = dht.readTemperature();
// Read temperature as Fahrenheit (isFahrenheit = true)
float t = dht.readTemperature(true);
```

Building the Web Page

Proceeding to the web page.



As you can see in the previous figure, the web page shows one heading and two paragraphs. There is a paragraph to display the temperature and another to display the humidity. There are also two icons to style our page.

All the HTML text with styles included is stored in the `index_html` variable. Now we'll go through the HTML text and see what each part does.

The following `<meta>` tag makes your web page responsive in any browser.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

The following `<script>` tag is needed to load the icons from the fontawesome website (the thermometer and the droplet icons).

```
<script src="https://kit.fontawesome.com/0294e3a09e.js" crossorigin="anonymous"></script>
```

Styles

Between the `<style></style>` tags, add some CSS to style the web page.

```
<style>
  html {
    font-family: Arial;
    display: inline-block;
    margin: 0px auto;
    text-align: center;
  }
  h2 { font-size: 3.0rem; }
  p { font-size: 3.0rem; }
  .units { font-size: 1.2rem; }
  .dht-labels{
    font-size: 1.5rem;
    vertical-align:middle;
    padding-bottom: 15px;
  }
</style>
```

This sets the full web page with Arial font, displayed as a in block without margin, and aligned at the center.

```
html {
  font-family: Arial;
  display: inline-block;
  margin: 0px auto;
  text-align: center;
}
```

We set the font size for the heading (`h2`), paragraph (`p`) and the units (`.units`) of the readings.

```
h2 { font-size: 3.0rem; }
p { font-size: 3.0rem; }
.units { font-size: 1.2rem; }
```

The labels for the readings are styled as shown below:

```
.dht-labels{
  font-size: 1.5rem;
  vertical-align:middle;
  padding-bottom: 15px;
}
```

All of the previous tags should go between the `<head>` and `</head>` tags. These tags are used to include content that is not directly visible to the user, like the `<meta>`, the `<script>` tags, and the styles.

HTML Body

Inside the `<body></body>` tags is where we add the web page content.

The `<h2></h2>` tags add a heading to the web page. In this case, the “ESP32 DHT server” text, but you can add any other text.

```
<h2>ESP32 DHT Server</h2>
```

Then, there are two paragraphs. One to display the temperature and another to display the humidity. The paragraphs are delimited by the `<p>` and `</p>` tags. The paragraph for the temperature is the following:

```
<p>
  <i class="fas fa-thermometer-half" style="color:#059e8a;"></i>
  <span class="dht-labels">Temperature</span>
  <span id="temperature">%TEMPERATURE%</span>
  <sup class="units">&deg;C</sup>
</p>
```

And the paragraph for the humidity is on the following snippet:

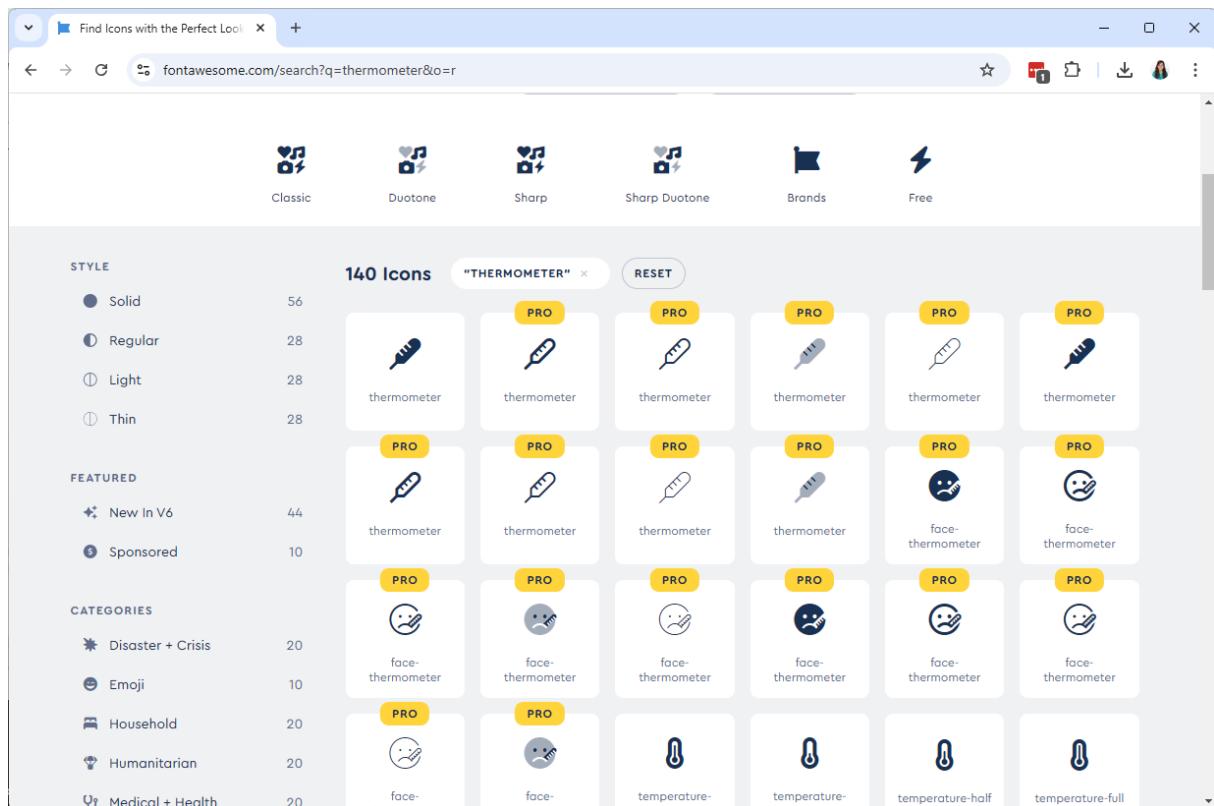
```
<p>
  <i class="fas fa-tint" style="color:#00add6;"></i>
  <span class="dht-labels">Humidity</span>
  <span id="humidity">%HUMIDITY%</span>
  <sup class="units">&percnt;</sup>
</p>
```

The `<i>` tags display the fontawesome icons.

How to display icons

This section shows how to choose icons from the fontawesome website if you want to change them. To select the icons, go to the [Font Awesome Icons website](#).

Search the icon you’re looking for. For example, “thermometer”.



Click the desired icon. Then, you just need to copy the HTML text provided.

```
<i class="fas fa-thermometer-half">
```

The screenshot shows the detailed view of the 'temperature-half' icon. On the left, there's a large preview of the icon. To its right is a dropdown menu for 'Classic' style. Below the preview, there are four smaller versions of the icon in different colors. At the bottom of this section are navigation buttons for 'WEATHER', 'temperature-2', 'thermometer-2', 'thermometer-half', and two version numbers, 4.7.0 and 6.6.0. A red arrow points from the text above to the highlighted HTML code in the screenshot.

To choose the color, you just need to pass the `style` parameter with the color in hexadecimal, as follows:

```
<i class="fas fa-thermometer-half" style="color:#059e8a;"></i>
```

Proceeding with the HTML text...

The next line writes the word “Temperature” into the web page.

```
<span class="dht-labels">Temperature</span>
```

The TEMPERATURE text between % signs is a placeholder for the temperature value.

```
<span id="temperature">%TEMPERATURE%</span>
```

This means that this **%TEMPERATURE%** text is like a variable that will be replaced by the actual temperature value from the DHT sensor. The placeholders on the HTML text should go between % signs.

Finally, we add the degree symbol.

```
<sup class="units">&deg;C</sup>
```

The `` tags make the text superscript.

We use the same approach for the humidity paragraph, but it uses a different icon and the **%HUMIDITY%** placeholder.

```
<p>
  <i class="fas fa-tint" style="color:#00add6;"></i>
  <span class="dht-labels">Humidity</span>
  <span id="humidity">%HUMIDITY%</span>
  <sup class="units">&percnt;</sup>
</p>
```

Automatic Updates

Finally, there's some JavaScript code in our web page that updates the temperature and humidity automatically, every 10 seconds.

Scripts in HTML text should go between the `<script></script>` tags.

```
<script>
setInterval(function () {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("temperature").innerHTML = this.responseText;
    }
  };
  xhttp.open("GET", "/temperature", true);
  xhttp.send();
}, 10000 ) ;
setInterval(function () {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
```

```
        document.getElementById("humidity").innerHTML = this.responseText;
    }
};

xhttp.open("GET", "/humidity", true);
xhttp.send();
}, 10000 ) ;
</script>
```

To update the temperature in the background, we have a `setInterval()` function that runs every 10 seconds.

Basically, it makes a request in the `/temperature` URL to get the latest temperature reading.

```
xhttp.open("GET", "/temperature", true);
xhttp.send();
}, 10000 ) ;
```

When it receives the response with the temperature, it updates the HTML element whose id is `temperature`.

```
if (this.readyState == 4 && this.status == 200) {
    document.getElementById("humidity").innerHTML = this.responseText;
}
```

In summary, this previous section is responsible for updating the temperature asynchronously. The same process is repeated for the humidity readings.

Important: since the DHT sensor is quite slow getting the readings, if you plan to have multiple clients connected to an ESP32 at the same time, we recommend increasing the request interval or removing the automatic updates.

Processor

Now, we need to create the `processor()` function that will replace the placeholders in our HTML text with the actual temperature and humidity values.

```
// Replaces placeholder with DHT values
String processor(const String& var){
    //Serial.println(var);
    if(var == "TEMPERATURE"){
        return readDHTTemperature();
    }
    else if(var == "HUMIDITY"){
        return readDHTHumidity();
    }
    return String();
}
```

When the web page is requested, we check if the HTML has any placeholders. If it finds the **%TEMPERATURE%** placeholder, we return the temperature by calling the `readDHTTemperature()` function created previously.

```
if(var == "TEMPERATURE"){
    return readDHTTemperature();
}
```

If the placeholder is **%HUMIDITY%**, we return the humidity value.

```
else if(var == "HUMIDITY"){
    return readDHTHumidity();
}
```

setup()

In the `setup()`, initialize the Serial Monitor for debugging purposes.

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

Initialize the DHT sensor.

```
dht.begin();
```

Connect to your local network and print the ESP32 IP address.

```
// Connect to Wi-Fi
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi..");
}

// Print ESP32 Local IP Address
Serial.println(WiFi.localIP());
```

Finally, add the next lines of code to handle the web server.

```
// Route for root / web page
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/html", index_html, processor);
});
server.on("/temperature", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/plain", readDHTTemperature().c_str());
});
server.on("/humidity", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/plain", readDHTHumidity().c_str());
});
```

When we make a request on the root URL, we send the HTML text that is stored in the `index_html` variable. We also need to pass the `processor` function, that will replace all the placeholders with the right values.

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/html", index_html, processor);
});
```

We need to add two additional handlers to update the temperature and humidity readings. When we receive a request on the `/temperature` URL, we simply need to send the updated temperature value. It is plain text, and it should be sent as a char, so, we use the `c_str()` method.

```
server.on("/temperature", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/plain", readDHTTemperature().c_str());
});
```

The same process is repeated for the humidity.

```
server.on("/humidity", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/plain", readDHTHumidity().c_str());
});
```

Lastly, we can start the server.

```
server.begin();
```

Because this is an asynchronous web server, we don't need to write anything in the `loop()`.

```
void loop(){  
}
```

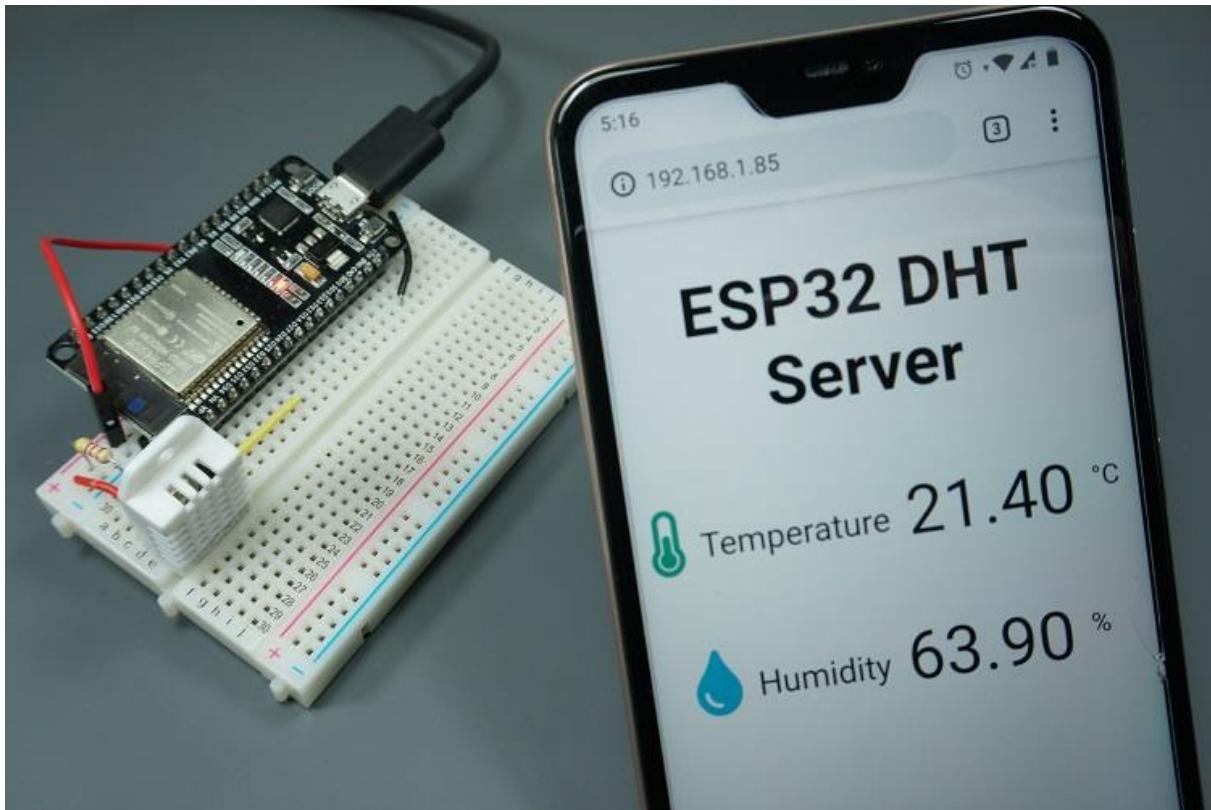
Upload the Code

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

After uploading, open the Serial Monitor at a baud rate of 115200. Press the ESP32 reset button. The ESP32 IP address should be printed on the serial monitor.

Demonstration

Open a browser and type the ESP32 IP address. Your web server should display the latest sensor readings.



Notice that the temperature and humidity readings are updated automatically without the need to refresh the web page.

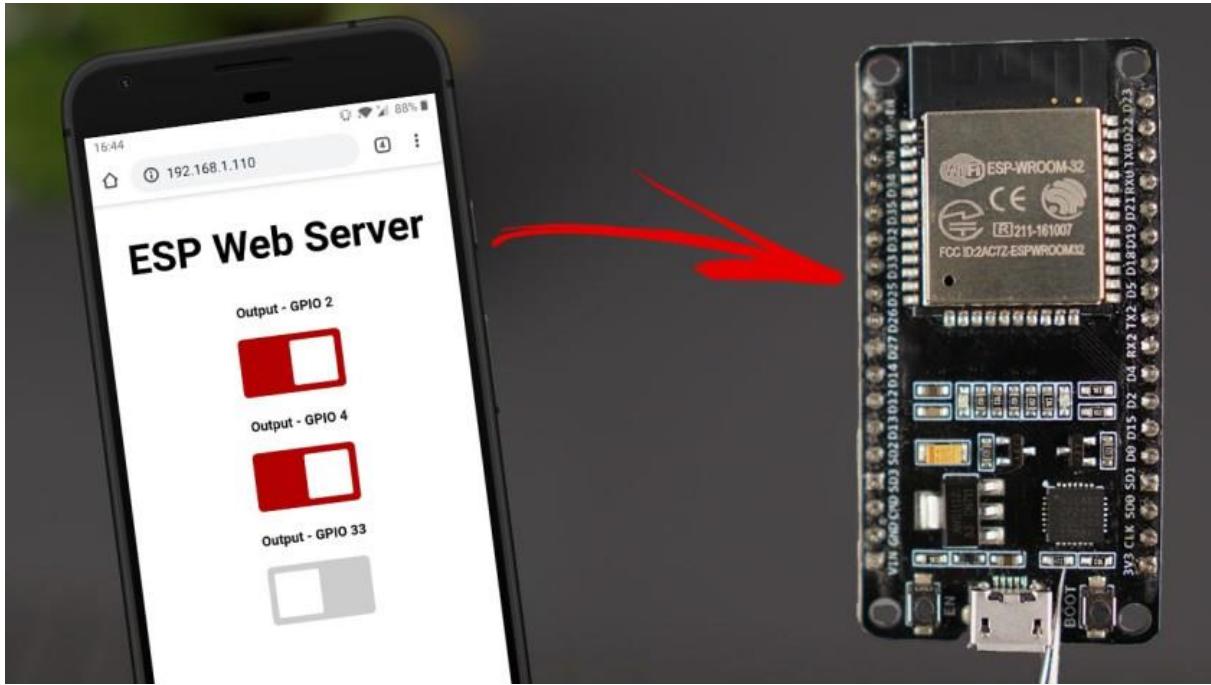
Wrapping Up

In this unit, we've provided a different way to build a web server using the [ESPAsyncWebServer](#) library. This library provides an easy way to build an asynchronous web server.

Building an asynchronous web server has several advantages like handling more than one connection at a time, not hanging waiting for clients, and much more. You can use this library to build web servers with different functionalities. We recommend taking a look at the library documentation for more information [here](#).

You can also build a web server and store the HTML and CSS text on separate files that you then refer to in the code—[see Unit 6.9](#).

6.8 - Asynchronous Web Server: Control Outputs



In this tutorial, you'll learn how to build an asynchronous web server with the ESP32 board to control its outputs using the `ESPAsyncWebServer` library.

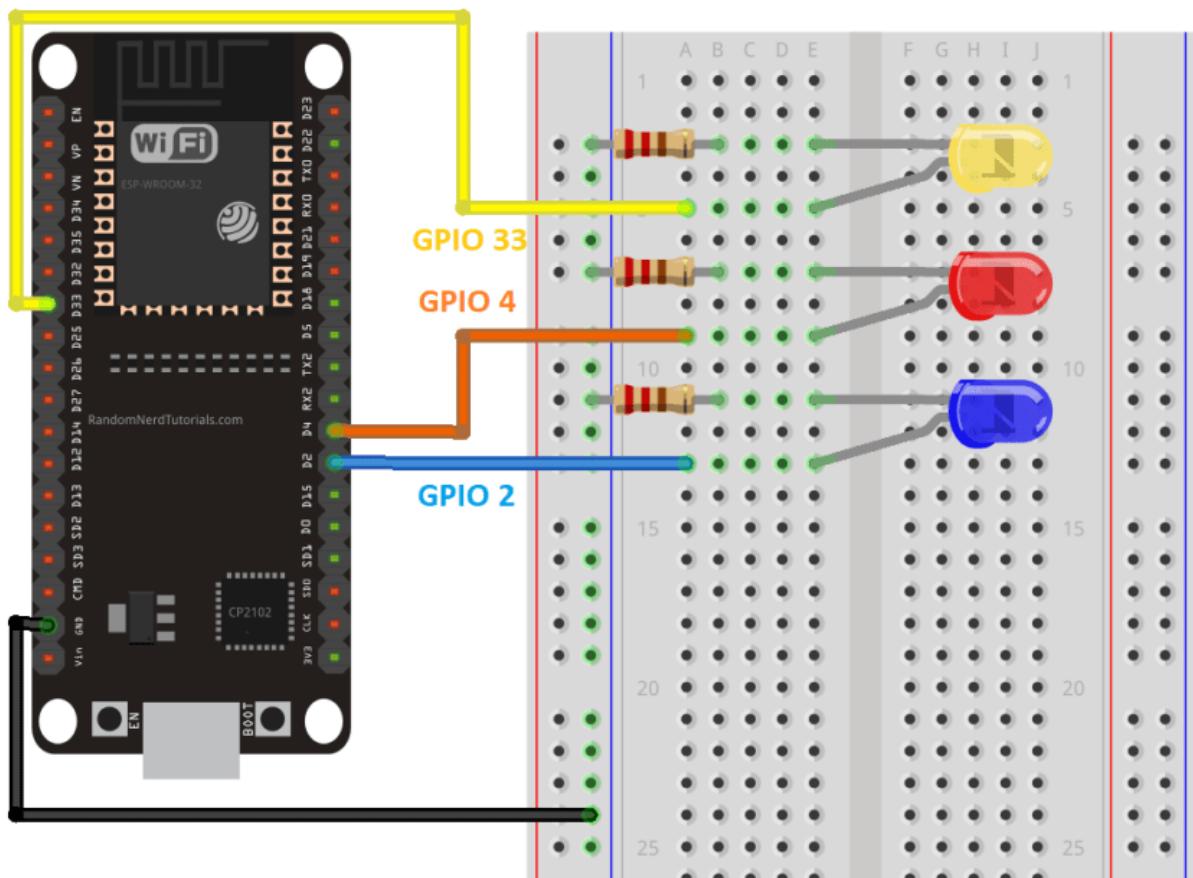
Parts Required

As an example, we'll control three LEDs. So, you need the following parts:

- [ESP32](#)
- 3x [LEDs](#)
- 3x [220 Ohm Resistors \(or similar values\)](#)
- [Breadboard](#)
- [Jumper wires](#)

Schematic

Before proceeding to the code, wire 3 LEDs to the ESP32, we're connecting the LEDs to GPIOs 2, 4, and 33, but you can use any other GPIOs.



Installing Libraries – Async Web Server

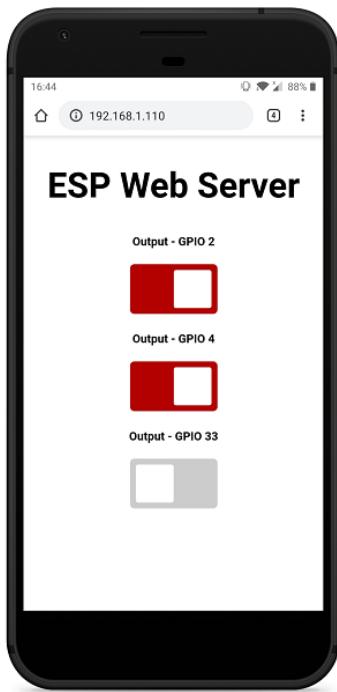
To build the web server, you need to install the following libraries. If you've followed previous Units, you should have already installed them. Otherwise, click the links below to download the libraries.

- [ESPAsyncWebServer](#)
- [AsyncTCP](#)

These libraries aren't available to install through the Arduino Library Manager, so you need to go to **Sketch > Include Library > Add .zip Library** and select the libraries you've just downloaded.

Project Overview

To better understand the code, let's see how the web server works.

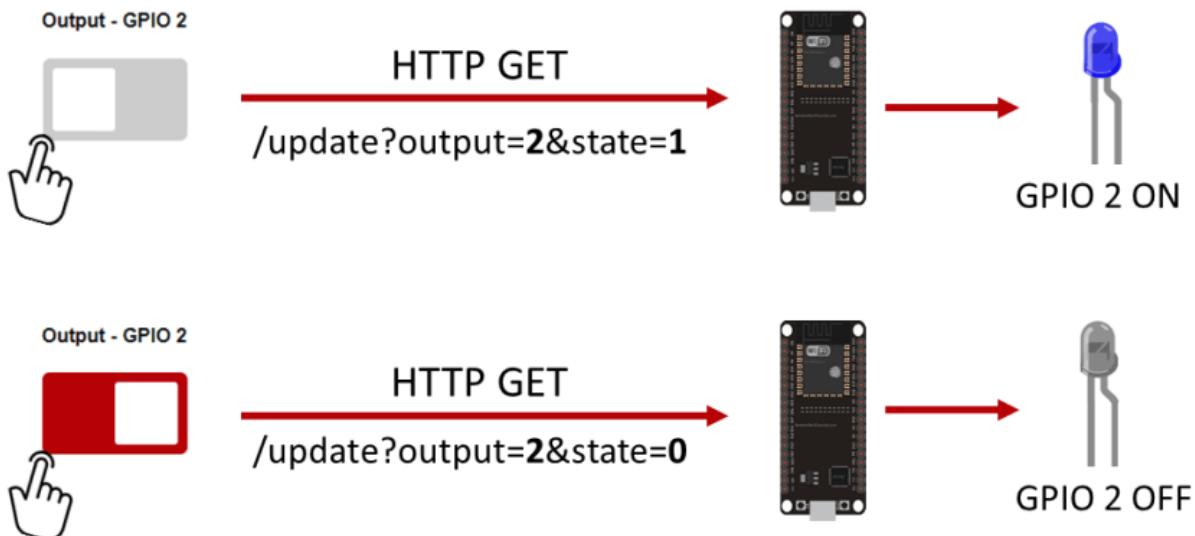


The web server contains one heading “ESP Web Server” and three buttons (toggle switches) to control three outputs. Each slider button has a label indicating the GPIO output pin. You can easily remove/add more outputs.

When the slider is red, it means the output is on (its state is HIGH). If you toggle the slider, it turns off the output (change the state to LOW).

When the slider is gray, it means the output is off (its state is LOW). If you toggle the slider, it turns on the output (change the state to HIGH).

How it Works?



Let's see what happens when you toggle the buttons. We'll see the example for GPIO 2. It works similarly for the other buttons.

1. In the first scenario, you toggle the button to turn GPIO 2 on. When that happens, it makes an HTTP GET request on the `/update?output=2&state=1` URL. Based on that URL, we change the state of GPIO 2 to 1 (HIGH) and turn the LED on.
2. In the second example, when you toggle the button to turn GPIO 2 off. When that happens, make an HTTP GET request on the `/update?output=2&state=0` URL. Based on that URL, we change the state of GPIO 2 to 0 (LOW) and turn the LED off.

Code for ESP32 Async Web Server

Copy the following code to your Arduino IDE.

- [Click here to download the code.](#)

```
// Import required libraries
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>

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

const char* PARAM_INPUT_1 = "output";
const char* PARAM_INPUT_2 = "state";
```

```

// Create AsyncWebServer object on port 80
AsyncWebServer server(80);

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE HTML><html>
<head>
  <title>ESP Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,>
  <style>
    html {font-family: Arial; display: inline-block; text-align: center;}
    h2 {font-size: 3.0rem;}
    p {font-size: 3.0rem;}
    body {max-width: 600px; margin:0px auto; padding-bottom: 25px;}
    .switch {position: relative; display: inline-block; width: 120px; height: 68px}
    .switch input {display: none}
    .slider {position: absolute; top: 0; left: 0; right: 0; bottom: 0;
              background-color: #ccc; border-radius: 6px}
    .slider:before {position: absolute; content: ""; height: 52px; width: 52px;
                    left: 8px; bottom: 8px; background-color: #fff;
                    -webkit-transition: .4s; transition: .4s; border-radius: 3px}
    input:checked+.slider {background-color: #b30000}
    input:checked+.slider:before {-webkit-transform: translateX(52px);
                                  -ms-transform: translateX(52px); transform: translateX(52px)}
  </style>
</head>
<body>
  <h2>ESP Web Server</h2>
  %BUTTONPLACEHOLDER%
<script>function toggleCheckbox(element) {
  var xhr = new XMLHttpRequest();
  if(element.checked)
  { xhr.open("GET", "/update?output="+element.id+"&state=1", true); }
  else { xhr.open("GET", "/update?output="+element.id+"&state=0", true); }
  xhr.send();
}
</script>
</body>
</html>
)rawliteral";

// Replaces placeholder with button section in your web page
String processor(const String& var){
  //Serial.println(var);
  if(var == "BUTTONPLACEHOLDER"){
    String buttons = "";
    buttons += "<h4>Output - GPIO 2</h4><label class=\"switch\">
      <input type=\"checkbox\" onchange=\"toggleCheckbox(this)\"
        id=\"2\" " + outputState(2) + "><span class=\"slider\">
      </span></label>";
    buttons += "<h4>Output - GPIO 4</h4><label class=\"switch\">
      <input type=\"checkbox\" onchange=\"toggleCheckbox(this)\"
        id=\"4\" " + outputState(4) + "><span class=\"slider\">
      </span></label>";
    buttons += "<h4>Output - GPIO 33</h4><label class=\"switch\">
      <input type=\"checkbox\" onchange=\"toggleCheckbox(this)\"
        id=\"33\" " + outputState(33) + ">
      <span class=\"slider\"></span></label>";
    return buttons;
  }
}

```

```

    return String();
}

String outputState(int output){
    if(digitalRead(output)){
        return "checked";
    }
    else {
        return "";
    }
}

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

    pinMode(2, OUTPUT);
    digitalWrite(2, LOW);
    pinMode(4, OUTPUT);
    digitalWrite(4, LOW);
    pinMode(33, OUTPUT);
    digitalWrite(33, LOW);

    // Connect to Wi-Fi
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.println("Connecting to WiFi..");
    }

    // Print ESP Local IP Address
    Serial.println(WiFi.localIP());

    // Route for root / web page
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send_P(200, "text/html", index_html, processor);
    });

    // Send a GET request to <ESP_IP>/update?output=<inputMessage1>&state=<inputMessage2>
    server.on("/update", HTTP_GET, [] (AsyncWebServerRequest *request) {
        String inputMessage1;
        String inputMessage2;
        // GET input1 value on <ESP_IP>/update?output=<inputMessage1>&state=<inputMessage2>
        if (request->hasParam(PARAM_INPUT_1) && request->hasParam(PARAM_INPUT_2)) {
            inputMessage1 = request->getParam(PARAM_INPUT_1)->value();
            inputMessage2 = request->getParam(PARAM_INPUT_2)->value();
            digitalWrite(inputMessage1.toInt(), inputMessage2.toInt());
        }
        else {
            inputMessage1 = "No message sent";
            inputMessage2 = "No message sent";
        }
        Serial.print("GPIO: ");
        Serial.print(inputMessage1);
        Serial.print(" - Set to: ");
        Serial.println(inputMessage2);
        request->send(200, "text/plain", "OK");
    });

    // Start server
    server.begin();
}

```

```
}

void loop() {

}
```

How Does the Code Work

In this section, we'll explain how the code works. Keep reading if you want to learn more or jump to the Demonstration section to see the final result.

Importing libraries

First, import the required libraries. You need to include the `WiFi`, `ESPAsyncWebserver` and the `ESPAsyncTCP` libraries.

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

Setting your network credentials

Insert your network credentials in the following variables so that the ESP32 can connect to your local network.

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

Input Parameters

To check the parameters passed on the URL (GPIO number and its state), we create two variables, one for the `output` and other for the `state`.

```
const char* PARAM_INPUT_1 = "output";
const char* PARAM_INPUT_2 = "state";
```

Remember that the ESP32 receives requests like this: `/update?output=2&state=0`

AsyncWebServer object

Create an `AsyncWebServer` object on port 80.

```
AsyncWebServer server(80);
```

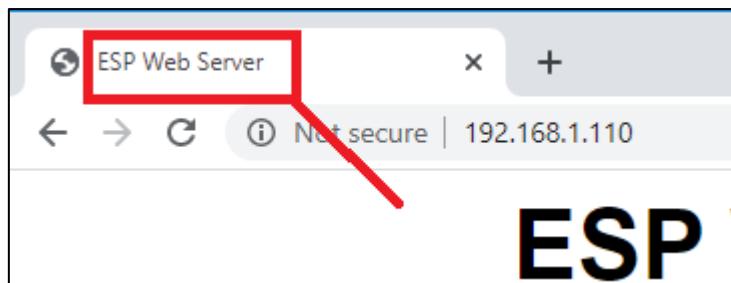
Building the Web Page

All the HTML text with styles and JavaScript is stored in the `index_html` variable.

Now we'll go through the HTML text and see what each part does.

The title goes inside the `<title>` and `</title>` tags. The title is exactly what it sounds like: the title of your document, which shows up in your web browser's title bar. In this case, it is "ESP Web Server".

```
<title>ESP Web Server</title>
```



The following `<meta>` tag makes your web page responsive in any browser (laptop, tablet, or smartphone).

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

The next line prevents requests on the favicon. In this case, we don't have a favicon. The favicon is the website icon that shows next to the title in the web browser tab. If we don't add the following line, the ESP32 will receive a request for the favicon every time we access the web server.

```
<link rel="icon" href="data:,">
```

Between the `<style></style>` tags, we add some CSS to style the web page. We won't go into detail on how this CSS styling works.

```
<style>
  html {font-family: Arial; display: inline-block; text-align: center;}
  h2 {font-size: 3.0rem;}
  p {font-size: 3.0rem;}
  body {max-width: 600px; margin: 0px auto; padding-bottom: 25px;}
  .switch {position: relative; display: inline-block; width: 120px; height: 68px}
  .switch input {display: none}
  .slider {position: absolute; top: 0; left: 0; right: 0; bottom: 0;
            background-color: #ccc; border-radius: 6px}
  .slider:before {position: absolute; content: ""; height: 52px; width: 52px;
                  left: 8px; bottom: 8px; background-color: #fff;
                  -webkit-transition: .4s; transition: .4s; border-radius: 3px}
  input:checked+.slider {background-color: #b30000}
```

```
input:checked+.slider:before {-webkit-transform: translateX(52px);  
-ms-transform: translateX(52px); transform: translateX(52px)}  
</style>
```

HTML Body

Inside the `<body></body>` tags is where we add the web page content.

The `<h2></h2>` tags add a heading to the web page. In this case, the “ESP Web Server” text, but you can add any other text.

```
<h2>ESP Web Server</h2>
```

After the heading, we have the buttons. The way the buttons show up on the web page (red: if the GPIO is on; or gray: if the GPIO is off) varies depending on the current GPIO state.

When you access the web server page, you want it to show the right current GPIO states. So, instead of adding the HTML text to build the buttons, we'll add a placeholder `%BUTTONPLACEHOLDER%`. When the web page is loaded, this placeholder will then be replaced with the actual HTML text to build the buttons with the right states.

```
%BUTTONPLACEHOLDER%
```

JavaScript

Then, there's some JavaScript that is responsible to make an HTTP GET request when you toggle the buttons as we've explained previously.

```
<script>function toggleCheckbox(element) {  
  var xhr = new XMLHttpRequest();  
  if(element.checked)  
  { xhr.open("GET", "/update?output="+element.id+"&state=1", true); }  
  else { xhr.open("GET", "/update?output="+element.id+"&state=0", true); }  
  xhr.send();  
}  
</script>
```

Here's the line that makes the request:

```
if(element.checked)  
{xhr.open("GET", "/update?output="+element.id+"&state=1", true); }
```

`element.id` returns the id of an HTML element. The id of each button will be the GPIO controlled as we'll see in the next section:

- GPIO 2 button → `element.id = 2`
- GPIO 4 button → `element.id = 4`
- GPIO 33 button → `element.id = 33`

Processor

Now, we need to create the `processor()` function that replaces the placeholders in the HTML text with what we define.

When the web page is requested, check if the HTML has any placeholders. If it finds the `%BUTTONPLACEHOLDER%` placeholder, it returns the HTML text to create the buttons.

```
// Replaces placeholder with button section in your web page
String processor(const String& var){
    //Serial.println(var);
    if(var == "BUTTONPLACEHOLDER"){
        String buttons = "";
        buttons += "<h4>Output - GPIO 2</h4><label class=\"switch\">
                    <input type=\"checkbox\" onchange=\"toggleCheckbox(this)\" id=\"2\" " + outputState(2) + ">
                    <span class=\"slider\"></span></label>";
        buttons += "<h4>Output - GPIO 4</h4><label class=\"switch\">
                    <input type=\"checkbox\" onchange=\"toggleCheckbox(this)\" id=\"4\" " + outputState(4) + ">
                    <span class=\"slider\"></span></label>";
        buttons += "<h4>Output - GPIO 33</h4><label class=\"switch\">
                    <input type=\"checkbox\" onchange=\"toggleCheckbox(this)\" id=\"33\" " + outputState(33) + ">
                    <span class=\"slider\"></span></label>";
        return buttons;
    }
    return String();
}
```

You can easily delete or add more lines to create more buttons.

Let's take a look at how the buttons are created. We create a `String` variable called `buttons` that contains the HTML text to build the buttons. We concatenate the HTML text with the current output state so that the toggle button is either gray or red.

The current output state is returned by the `outputState(<GPIO>)` function (it accepts as argument the GPIO number). See below:

```
buttons += "<h4>Output - GPIO 2</h4><label class=\"switch\">\n    <input type=\"checkbox\" onchange=\"toggleCheckbox(this)\"\n        id=\"2\" " + outputState(2) + "><span class=\"slider\"></span></label>";
```

The \ is used so that we can pass "" inside the String.

The `outputState()` function returns either "checked" if the GPIO is on or an empty field "" if the GPIO is off.

```
String outputState(int output){\n    if(digitalRead(output)){\n        return "checked";\n    }\n    else {\n        return \"\";\n    }\n}
```

So, the HTML text for GPIO 2 when it is on, would be:

```
<h4>Output - GPIO 2</h4>\n<label class="switch">\n    <input type="checkbox" onchange="toggleCheckbox(this)" id="2" checked>\n    <span class="slider"></span>\n</label>
```

Let's break this down into smaller sections to understand how it works.

In HTML, a toggle switch is an input type. The `<input>` tag specifies an input field where the user can enter data. The toggle switch is an input field of type `checkbox`. There are many other input field types.

```
<input type="checkbox">
```

The checkbox can be checked or not. When it is checked, you have something as follows:

```
<input type="checkbox" checked>
```

The `onchange` is an event attribute that occurs when we change the value of the element (the checkbox). Whenever you check or uncheck the toggle switch, it calls the `toggleCheckbox()` JavaScript function for that specific element id (`this`).

The `id` specifies a unique id for that HTML element. The `id` allows us to manipulate the element using JavaScript or CSS.

```
<input type="checkbox" onchange="toggleCheckbox(this)" id="2" checked>
```

setup()

In the `setup()`, initialize the Serial Monitor for debugging purposes.

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

Set the GPIOs you want to control as outputs using the `pinMode()` function and set them to `LOW` when the ESP32 first starts. If you've added more GPIOs, do the same procedure.

```
pinMode(2, OUTPUT);
digitalWrite(2, LOW);
pinMode(4, OUTPUT);
digitalWrite(4, LOW);
pinMode(33, OUTPUT);
digitalWrite(33, LOW);
```

Connect to your local network and print the ESP32 IP address.

```
// Connect to Wi-Fi
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(1000);
  Serial.println("Connecting to WiFi..");
}

// Print ESP Local IP Address
Serial.println(WiFi.localIP());
```

In the `setup()`, you need to handle what happens when the ESP32 receives requests. As we've seen previously, you receive a request of this type:

```
<ESP_IP>/update?output=<inputMessage1>&state=<inputMessage2>
```

So, we check if the request contains the `PARAM_INPUT1` variable value (`output`) and the `PARAM_INPUT2(state)` and save the corresponding values on the `input1Message` and `input2Message` variables.

```
if (request->hasParam(PARAM_INPUT_1) && request->hasParam(PARAM_INPUT_2)) {
  inputMessage1 = request->getParam(PARAM_INPUT_1)->value();
  inputMessage2 = request->getParam(PARAM_INPUT_2)->value();
```

Then, we control the corresponding GPIO with the corresponding state (the `inputMessage1` variable saves the GPIO number and the `inputMessage2` saves the state—`0` or `1`):

```
digitalWrite(inputMessage1.toInt(), inputMessage2.toInt());
```

Here's the complete code to handle the HTTP GET /update request:

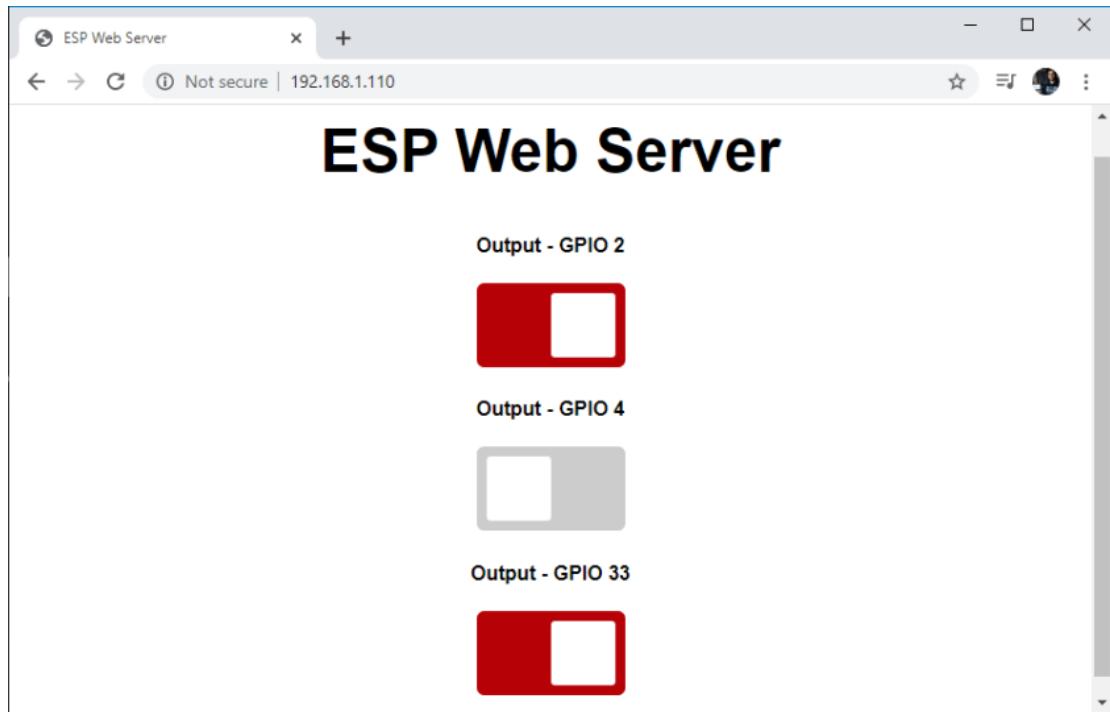
```
server.on("/update", HTTP_GET, [] (AsyncWebServerRequest *request) {
    String inputMessage1;
    String inputMessage2;
    // GET input1 value on <ESP_IP>/update?output=<inputMessage1>&state=<inputMessage2>
    if (request->hasParam(PARAM_INPUT_1) && request->hasParam(PARAM_INPUT_2)) {
        inputMessage1 = request->getParam(PARAM_INPUT_1)->value();
        inputMessage2 = request->getParam(PARAM_INPUT_2)->value();
        digitalWrite(inputMessage1.toInt(), inputMessage2.toInt());
    }
    else {
        inputMessage1 = "No message sent";
        inputMessage2 = "No message sent";
    }
    Serial.print("GPIO: ");
    Serial.print(inputMessage1);
    Serial.print(" - Set to: ");
    Serial.println(inputMessage2);
    request->send(200, "text/plain", "OK");
});
```

Finally, start the server:

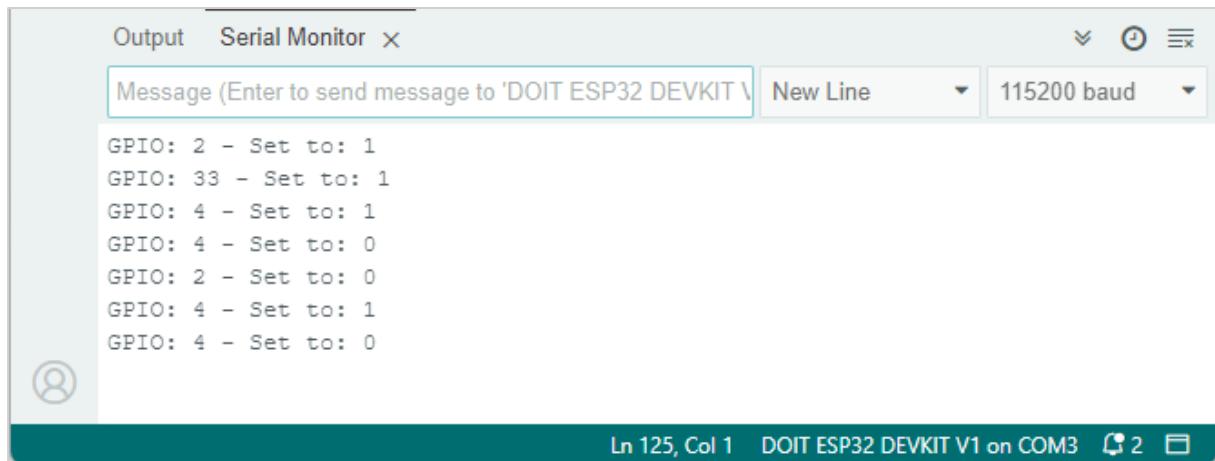
```
server.begin();
```

Demonstration

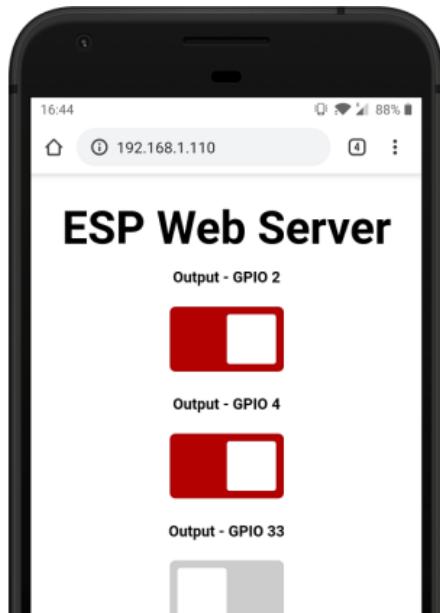
After uploading the code to your ESP32, open the serial monitor at a baud rate of 115200. Press the on-board RST/EN button. You should get its IP address. Open a browser and type the ESP IP address. You'll get access to a similar web page.



Press the toggle buttons to control the ESP32 GPIOs. At the same time, you should get the following messages on the serial monitor.



You can also access the web server from a browser on your smartphone. Whenever you open the web server, it shows the current GPIO states. Red indicates the GPIO is on, and gray that the GPIO is off.

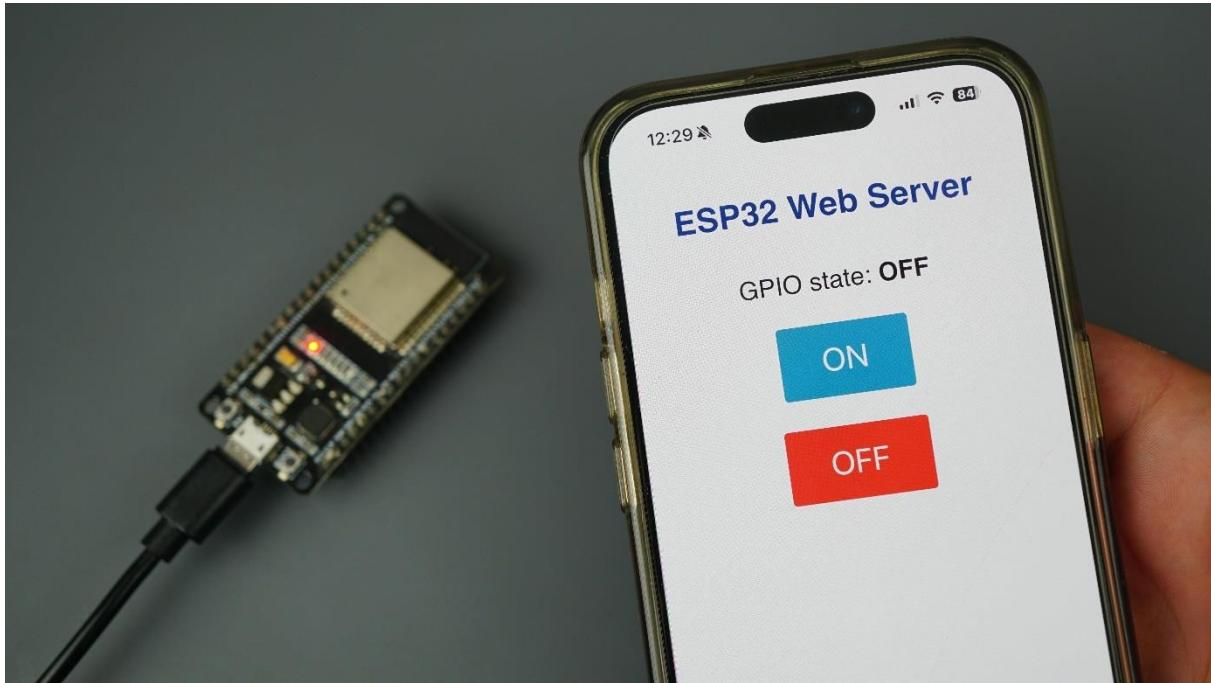


Wrapping Up

In this unit, you learned how to create an asynchronous web server with the ESP32 to control its outputs using toggle switches. Whenever you open the web page, it shows the updated GPIO states.

6.9 - Build an ESP32 Web Server using Files from Filesystem (LittleFS)

In [Units 3.2 and 3.3](#), we've covered how to upload and manipulate files using the ESP32 filesystem. In this Unit, we'll show you how to build a web server that serves HTML and CSS files stored on the filesystem.



The web server we'll build in this Unit is similar to the web server from [Unit 6.2](#).

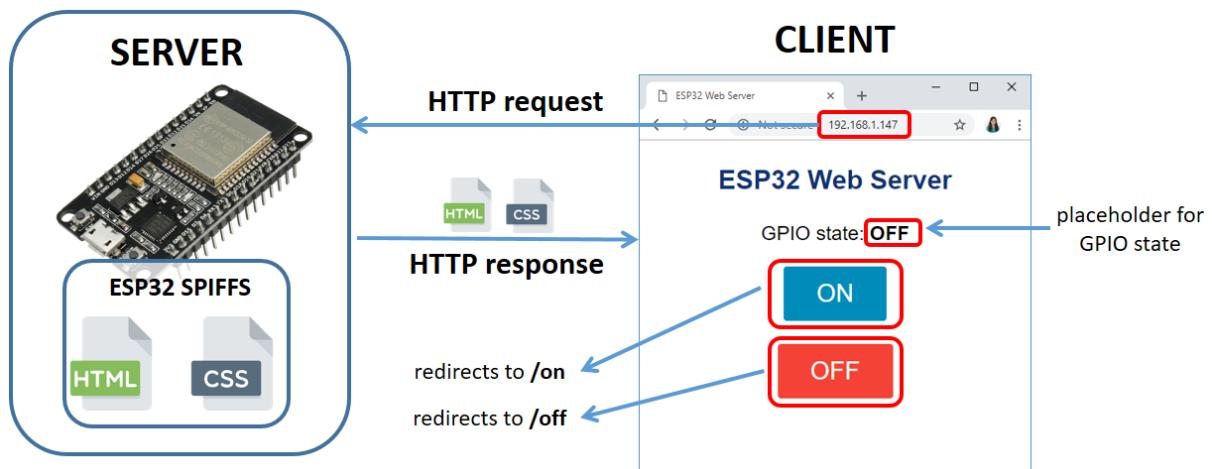
To continue with this Unit, you should have the ESP32 Filesystem Uploader plugin installed in your Arduino IDE. If you haven't, [follow Unit 3.2 to install it first](#).

Project Overview

Before going straight to the project, it's important to outline what our web server will do, so that it is easier to understand.



- The web server you'll build controls an LED connected to the ESP32 GPIO 2. This is the ESP32 on-board LED. You can control any other GPIO;
- The web server page shows two buttons: ON and OFF – to turn GPIO 2 on and off;
- The web server page also shows the current GPIO state.
- The following figure shows a simplified diagram to demonstrate how everything works.



- The ESP32 runs a web server code based on the [ESPAsyncWebServer library](#);

- The HTML and CSS files are stored on the ESP32 filesystem (LittleFS);
- When you make a request on a specific URL using your browser, the ESP32 responds with the requested files;
- When you click the ON button, you are redirected to the root URL followed by /on and the LED is turned on;
- When you click the OFF button, you are redirected to the root URL followed by /off and the LED is turned off;
- On the web page, there is a placeholder for the GPIO state. The placeholder for the GPIO state is written directly in the HTML file between % signs, for example %STATE%.

Installing Libraries

Until now, we've created the HTML and CSS for the web server as a String directly on the Arduino sketch. With the LittleFS filesystem, you can write the HTML and CSS in separate files and save them on the ESP32 filesystem.

One of the easiest ways to build a web server using files from the filesystem is by using the `ESPAsyncWebServer` library. The `ESPAsyncWebServer` library is well documented on its GitHub page. For more information about that library, check the following link:

- <https://github.com/me-no-dev/ESPAsyncWebServer>

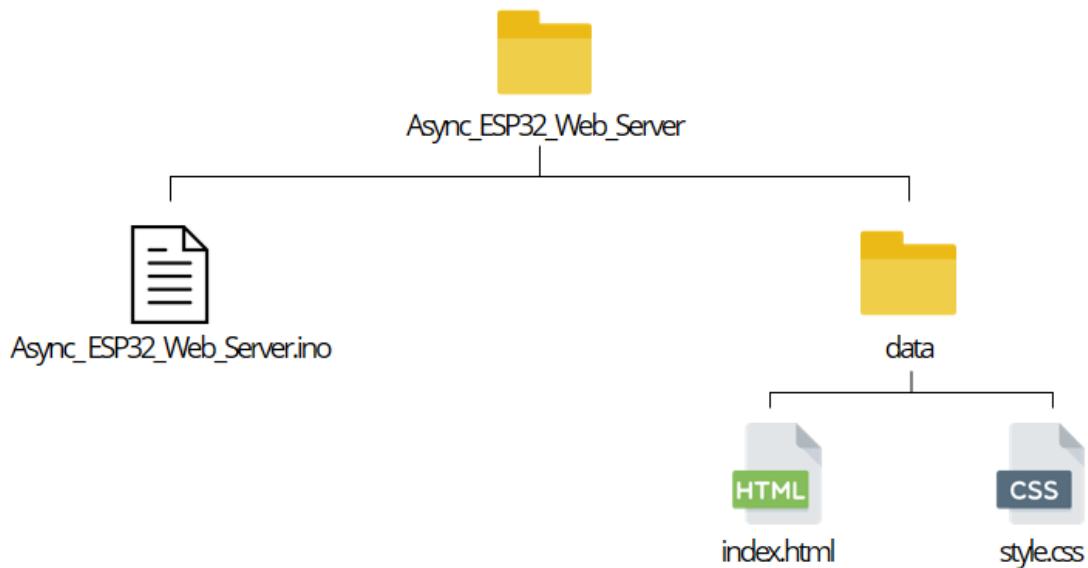
The `ESPAsyncWebServer` and `AsyncTCP` libraries aren't available to install through the Arduino Library Manager. So, you need to click the following links to download the library files.

- [Click here to download the `ESPAsyncWebServer` library.](#)
- [Click here to download the `Async TCP` library.](#)

In your Arduino IDE, go to **Sketch > Include Library > Add .zip Library** and select the libraries you've just downloaded.

Organizing your Files

To build the web server, you need three different files. The Arduino sketch, the HTML file and the CSS file. The HTML and CSS files should be saved inside a folder called **data** inside the Arduino sketch folder, as shown below:



Creating the HTML File

The HTML for this project is very simple. We just need to create a heading for the web page, a paragraph to display the GPIO state and two buttons.

Create an *index.html* file with the following content or [download all the project files here](#):

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,">
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  <h1>ESP32 Web Server</h1>
  <p>GPIO state: <strong> %STATE% </strong></p>
  <p><a href="/on"><button class="button">ON</button></a></p>
  <p><a href="/off"><button class="button button2">OFF</button></a></p>
</body>
</html>
```

Note: for an introduction to HTML and CSS, [check Unit 6.3](#).

Because we're using CSS and HTML in different files, we need to reference the CSS file on the HTML text. The following line should be added between the `<head>` and `</head>` tags:

```
<link rel="stylesheet" type="text/css" href="style.css">
```

The `<link>` tag tells the HTML file that you're using an external style sheet to format how the page looks. The `rel` attribute specifies the nature of the external file, in this case that it is a stylesheet—the CSS file—that will be used to alter the appearance of the page.

The `type` attribute is set to `"text/css"` to indicate that you're using a CSS file for the styles. The `href` attribute indicates the file location; since both the CSS and HTML files will be in the same folder, you just need to reference the filename: `style.css`.

In the following line, we write the first heading of our web page. In this case we have “ESP32 Web Server”. You can change the heading to any text you want:

```
<title>ESP32 Web Server</title>
```

Then, we add a paragraph with the text `“GPIO state:”` followed by the GPIO state. Because the GPIO state changes accordingly to the state of the GPIO, we can add a placeholder that will then be replaced for whatever value we set on the Arduino sketch.

To add placeholder, we use `%` signs. To create a placeholder for the state, we can use `%STATE%`, for example.

```
<p>GPIO state: <strong> %STATE%</strong></p>
```

Attributing a value to the `STATE` placeholder is done in the Arduino sketch.

Then, we create an ON and an OFF button. When you click the on button, we redirect the web page to the root followed by `/on` URL. When you click the off button, you are redirected to the `/off` URL.

```
<p><a href="/on"><button class="button">ON</button></a></p>
<p><a href="/off"><button class="button button2">OFF</button></a></p>
```

Creating the CSS file

Create the `style.css` file with the following content or [download all the project files here](#):

```
html {
    font-family: Helvetica;
    display: inline-block;
    margin: 0px auto;
    text-align: center;
}
h1{
    color: #0F3376;
    padding: 2vh;
}
p{
    font-size: 1.5rem;
}
.button {
    display: inline-block;
    background-color: #008CBA;
    border: none;
    border-radius: 4px;
    color: white;
    padding: 16px 40px;
    text-decoration: none;
    font-size: 30px;
    margin: 2px;
    cursor: pointer;
}
.button2 {
    background-color: #f44336;
}
```

This is just a basic CSS file to set the font size, style and color of the buttons and align the page. We won't explain how CSS works. A good place to learn about CSS is the [W3Schools website](#).

Arduino Sketch

Copy the following code to the Arduino IDE. Then, you need to type your network credentials (SSID and password) to make it work.

- [Click here to download the code.](#)

```
// Import required libraries
#include "WiFi.h"
#include "ESPAsyncWebServer.h"
#include "LittleFS.h"

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

```

const char* password = "REPLACE_WITH_YOUR_PASSWORD";

// Set LED GPIO
const int ledPin = 2;
// Stores LED state
String ledState;

// Create AsyncWebServer object on port 80
AsyncWebServer server(80);

// Replaces placeholder with LED state value
String processor(const String& var){
    Serial.println(var);
    if(var == "STATE"){
        if(digitalRead(ledPin)){
            ledState = "ON";
        }
        else{
            ledState = "OFF";
        }
        Serial.print(ledState);
        return ledState;
    }
    return String();
}

void setup(){
    // Serial port for debugging purposes
    Serial.begin(115200);
    pinMode(ledPin, OUTPUT);

    // Initialize LittleFS
    if(!LittleFS.begin(true)){
        Serial.println("An Error has occurred while mounting LittleFS");
        return;
    }

    // Connect to Wi-Fi
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.println("Connecting to WiFi..");
    }

    // Print ESP32 Local IP Address
    Serial.println(WiFi.localIP());

    // Route for root / web page
    server.on("/", HTTP_GET, [] (AsyncWebRequest *request){
        request->send(LittleFS, "/index.html", String(), false, processor);
    });

    // Route to load style.css file
    server.on("/style.css", HTTP_GET, [] (AsyncWebRequest *request){
        request->send(LittleFS, "/style.css", "text/css");
    });

    // Route to set GPIO to HIGH
    server.on("/on", HTTP_GET, [] (AsyncWebRequest *request){
        digitalWrite(ledPin, HIGH);
        request->send(LittleFS, "/index.html", String(), false, processor);
    });
}

```

```

});
```

```

// Route to set GPIO to LOW
server.on("/off", HTTP_GET, [](AsyncWebRequest *request){
    digitalWrite(ledPin, LOW);
    request->send(LittleFS, "/index.html", String(), false, processor);
});
```

```

// Start server
server.begin();
}
```

```

void loop(){
```

How the Code Works

First, include the necessary libraries:

```
#include "WiFi.h"
#include "ESPAsyncWebServer.h"
#include "LittleFS.h"
```

You need to type your network credentials in the following variables:

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

Next, create a variable that refers to GPIO 2 called `ledPin`, and a String variable to hold the led state: `ledState`.

```
// Set LED GPIO
const int ledPin = 2;
// Stores LED state
String ledState;
```

Create an `AsynWebServer` object called `server` that is listening on port 80.

```
AsyncWebServer server(80);
```

processor()

The `processor()` function is what will attribute a value to the placeholder we've created on the HTML file. It accepts as an argument the placeholder and should return a String that will replace the placeholder. The `processor()` function should have the following structure:

```
// Replaces placeholder with LED state value
String processor(const String& var){
```

```
Serial.println(var);
if(var == "STATE"){
    if(digitalRead(ledPin)){
        ledState = "ON";
    }
    else{
        ledState = "OFF";
    }
    Serial.print(ledState);
    return ledState;
}
return String();
```

This function first checks if the placeholder is the STATE we've created on the HTML file.

```
if(var == "STATE"){


```

If it is, then, according to the LED state, we set the `ledState` variable to either ON or OFF.

```
if(digitalRead(ledPin)){
    ledState = "ON";
}
else{
    ledState = "OFF";
}
```

Finally, we return the `ledState` variable. This replaces the placeholder with the `ledState` string value.

```
return ledState;
```

setup()

In the `setup()`, start by initializing the Serial Monitor, set the GPIO as an output, and initialize the LittleFS filesystem.

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

    // Initialize LittleFS
    if(!LittleFS.begin(true)){
        Serial.println("An Error has occurred while mounting LittleFS");
        return;
    }
}
```

Wi-Fi connection

Connect to Wi-Fi and print the ESP32 IP address:

```
// Connect to Wi-Fi
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi..");
}

// Print ESP32 Local IP Address
Serial.println(WiFi.localIP());
```

Async Web Server

The `ESPAsyncWebServer` library allows us to configure the routes where the server will be listening for incoming HTTP requests and execute functions when a request is received on that route. For that, use the `on()` method on the `server` object as follows:

```
server.on("/", HTTP_GET, [](AsyncWebRequest *request){
    request->send(LittleFS, "/index.html", String(), false, processor);
});
```

When the server receives a request on the root “`/`” URL, it will send the `index.html` file to the client. The last argument of the `send()` function is the `processor`, so that we can replace the placeholder for the value we want – in this case the `ledState`.

Because we’ve referenced the CSS file on the HTML file, the client will make a request for the CSS file. When that happens, the CSS file is sent to the client:

```
server.on("/style.css", HTTP_GET, [](AsyncWebRequest *request){
    request->send(LittleFS, "/style.css", "text/css");
});
```

Finally, you need to define what happens on the `/on` and `/off` routes. When a request is made on those routes, the LED is either turned on or off, and the ESP32 serves the HTML file.

```
// Route to set GPIO to HIGH
server.on("/on", HTTP_GET, [](AsyncWebRequest *request){
    digitalWrite(ledPin, HIGH);
    request->send(LittleFS, "/index.html", String(), false, processor);
});
```

```
// Route to set GPIO to LOW
server.on("/off", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, LOW);
    request->send(LittleFS, "/index.html", String(), false, processor);
});
```

In the end, we use the `begin()` method on the `server` object, so that the server starts listening for incoming clients.

```
server.begin();
```

Because this is an asynchronous web server, you can define all the requests in the `setup()`. Then, you can add other code to the `loop()` while the server is listening for incoming clients.

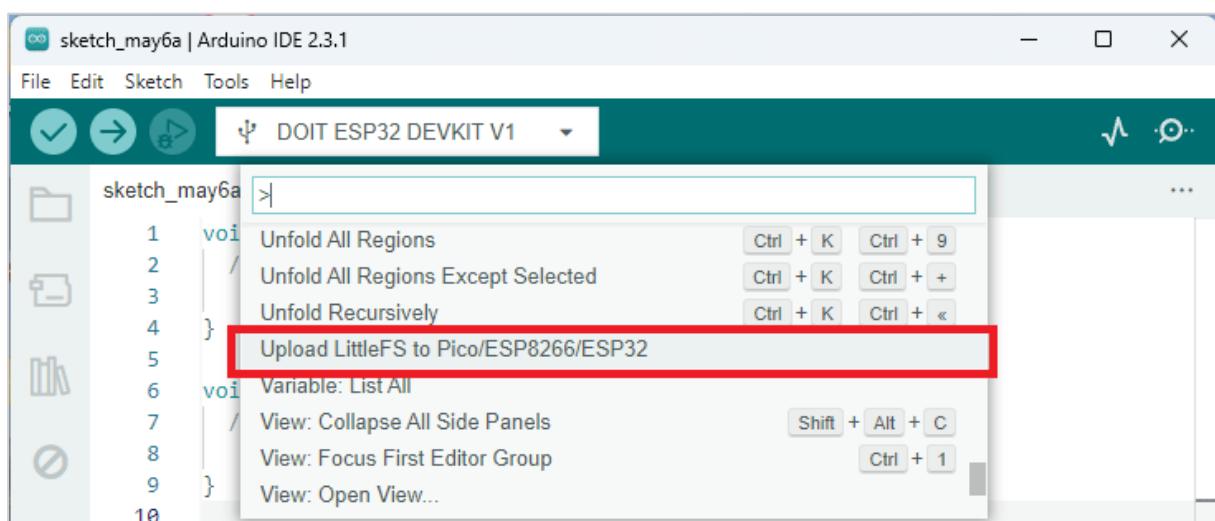
Uploading Code and Files

Save the code as **Async_ESP32_Web_Server** or [download all the project files here](#).

Go to **Sketch > Show Sketch Folder**, and create a folder called **data**. Inside that folder, you should save the HTML and CSS files.

Now, upload the files to the ESP32. Follow the next instructions:

- Press **[Ctrl] + [Shift] + [P]** to open the command palette.
- An instruction called '**Upload Little FS to Pico/ESP8266/ESP32**' should be there (just scroll down or search for the name of the instruction).



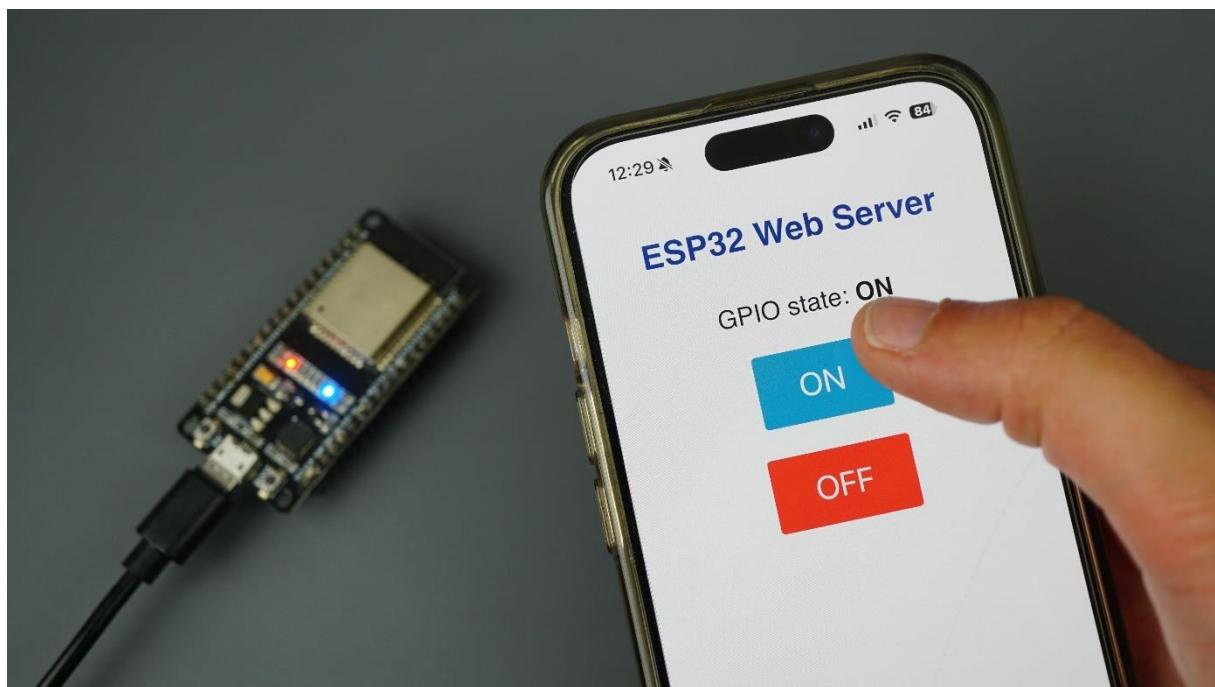
The files should be uploaded to the ESP32 after a few seconds.

Then, upload the code to your ESP32 board. Make sure you have the right board and COM port selected. Also, make sure you've added your network credentials to the code.

When everything is successfully uploaded, open the Serial Monitor at a baud rate of 115200. Press the ESP32 "ENABLE" button, and it should print the ESP32 IP address.

Demonstration

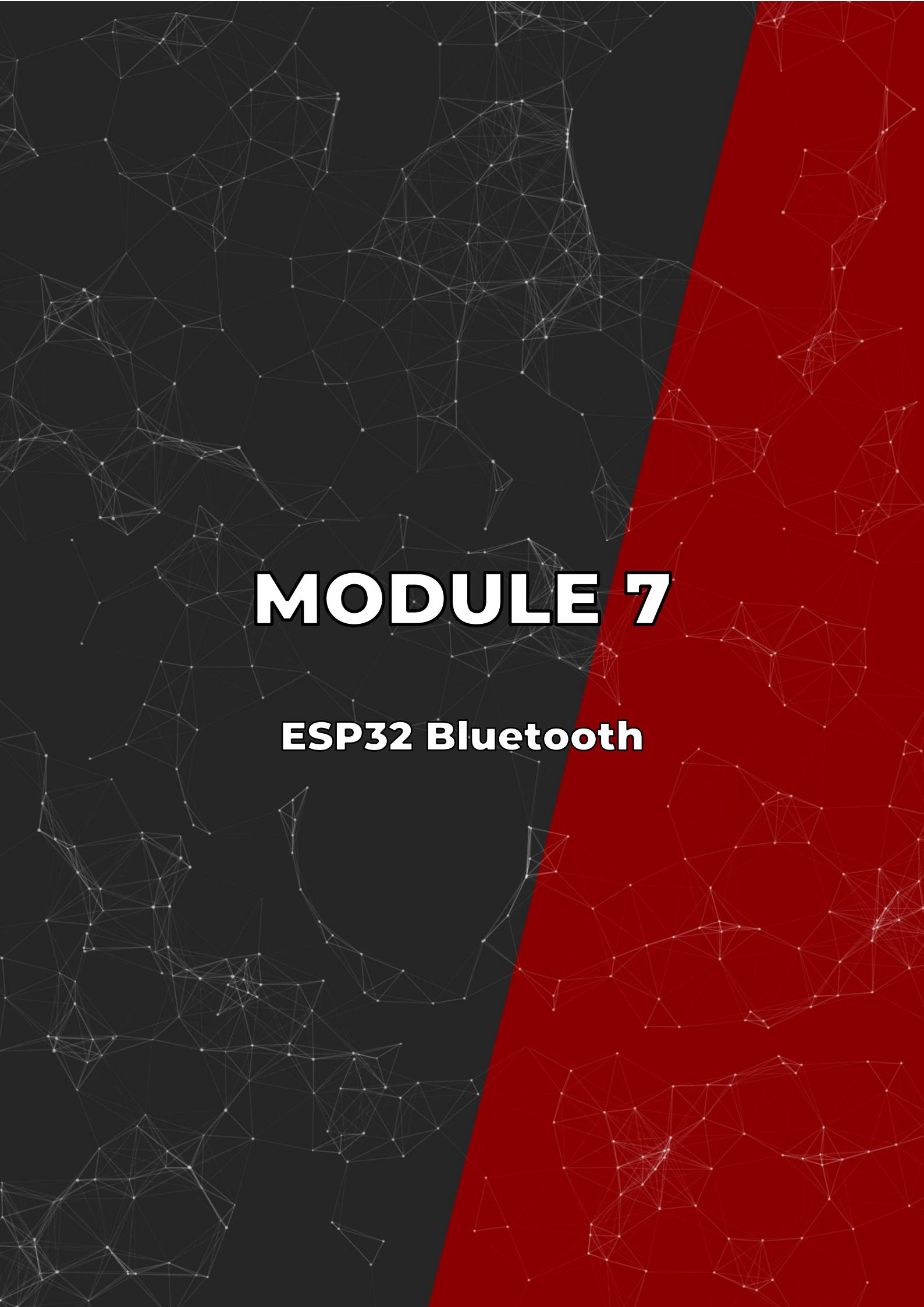
Open your browser and type the ESP32 IP address. Press the ON and OFF buttons to control the ESP32 on-board LED. Also, check that the GPIO state is being updated correctly.



Wrapping Up

Using the LittleFS filesystem is especially useful to store HTML and CSS files to serve to a client—instead of having to write all the code inside the Arduino sketch.

The ESPAsyncWebServer library allows you to build a web server by running a specific function in response to a specific request. You can also add placeholders to the HTML file that can be replaced with variables—like sensor readings, or GPIO states, for example.



MODULE 7

ESP32 Bluetooth

7.1 - ESP32 Bluetooth Low Energy (BLE): Introduction

In this Unit, we'll explore what's BLE (which stands for Bluetooth Low Energy) and its applications. The ESP32 chip comes not only with Wi-Fi, but also with Bluetooth and Bluetooth Low Energy (BLE) wireless capabilities.

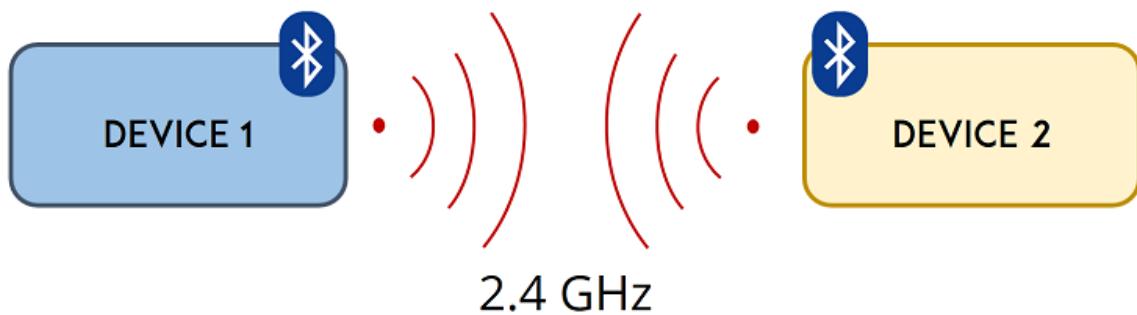


In this Unit, we'll cover:

- The Basics of Bluetooth Low Energy (BLE)
- BLE Terminology: UUID, Service, Characteristic, and Properties
- Server and Client interaction

What is Bluetooth?

Bluetooth is a wireless technology that enables devices to communicate over short distances. Just like Wi-Fi, Bluetooth also operates at 2.4GHz.

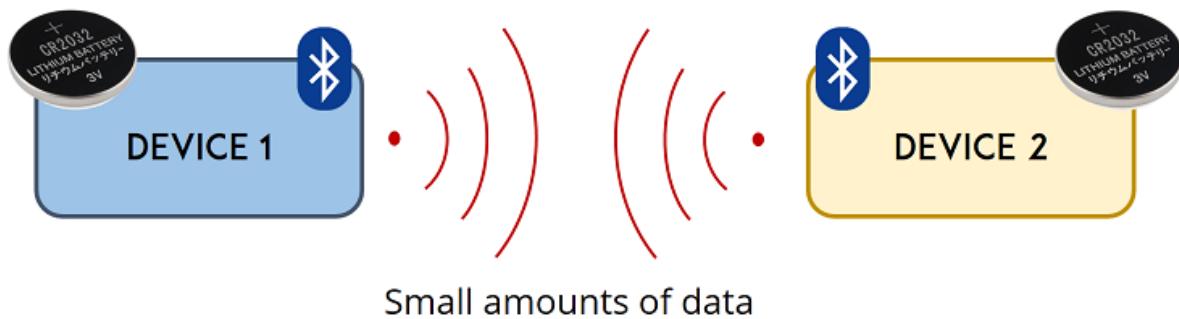


This variant of Bluetooth is also referred to as "Bluetooth Classic" or simply "Bluetooth". It was designed for high-speed data transmission and it's commonly used for connecting devices like headphones to phones, linking a keyboard or mouse to a computer, or transfer files between devices. Devices use Bluetooth in point-to-point communication that requires a continuous connection to transmit data.

What is Bluetooth Low Energy?



Bluetooth Low Energy, BLE for short (also called Bluetooth Smart), is a power-conserving variant of Bluetooth. BLE's primary application is short-distance transmission of small amounts of data (low bandwidth) and it's aimed at very low power applications running off a coin cell.



Unlike Bluetooth which is always on, BLE remains in sleep mode constantly except for when a connection is initiated. This makes it consume very little power. BLE consumes approximately 100x less power than Bluetooth (depending on the use case).

This feature is extremely useful in Machine to Machine (M2M) communication because you can have small devices powered with batteries that last for a very long time.

This makes it perfect for applications that need to periodically exchange small amounts of data, like in healthcare, fitness, tracking, beacons, security, and home automation industries.



Bluetooth Classic vs Bluetooth Low Energy

So, what are the main differences between Bluetooth Classic and Bluetooth Low Energy?

Bluetooth Classic is known for higher data transfer rates, making it suitable for applications like audio streaming and file transfer. It consumes more power, making it less ideal for battery-operated devices. Usually, Bluetooth Classic is easier to understand and implement for beginners, while Bluetooth Low Energy might take a little bit more time to understand basic concepts. That's why many people still prefer using Bluetooth Classic instead of BLE on their projects.

On the other hand, **BLE (Bluetooth Low Energy)** is designed for low power consumption, making it perfect for devices like IoT gadgets and wearables, and is also a great solution for the ESP32 in IoT and Home Automation applications. BLE operates with lower data transfer rates but is energy-efficient and works well in short-range scenarios.

Another big difference between the two versions of Bluetooth is the way used to transfer data. Bluetooth Classic uses something similar to Serial Communication (Serial Port Profile), while Bluetooth Low Energy uses a client-server model, where it employs the GATT (Generic Attribute Profile) to structure data.

You can check in [more detail the main differences between Bluetooth and Bluetooth Low Energy here.](#)

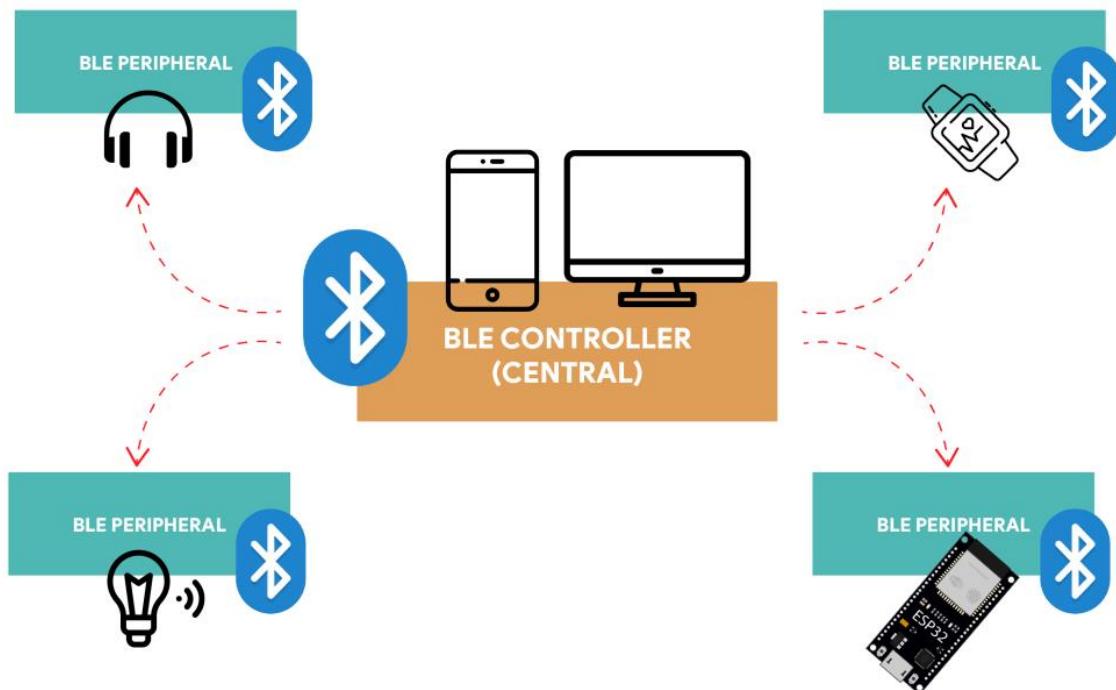
	Bluetooth Classic	Bluetooth Low Energy (BLE)
Power Consumption	Higher power consumption	Low power consumption
Data Transfer Rate	Higher data transfer rates	Lower data transfer rates
Range	Longer range	Shorter range
Applications	Audio streaming, file transfer	IoT devices, wearables, smart home
Data Transfer	Serial Port Profile (SPP)	Generic Attribute Profile (GATT)

Bluetooth Low Energy Basic Concepts

Before proceeding, it's important to get familiar with some basic BLE concepts

BLE Peripheral and Controller (Central Device)

When using Bluetooth Low Energy (BLE), it's important to understand the roles of BLE Peripheral and BLE Controller (also referred to as the Central Device).



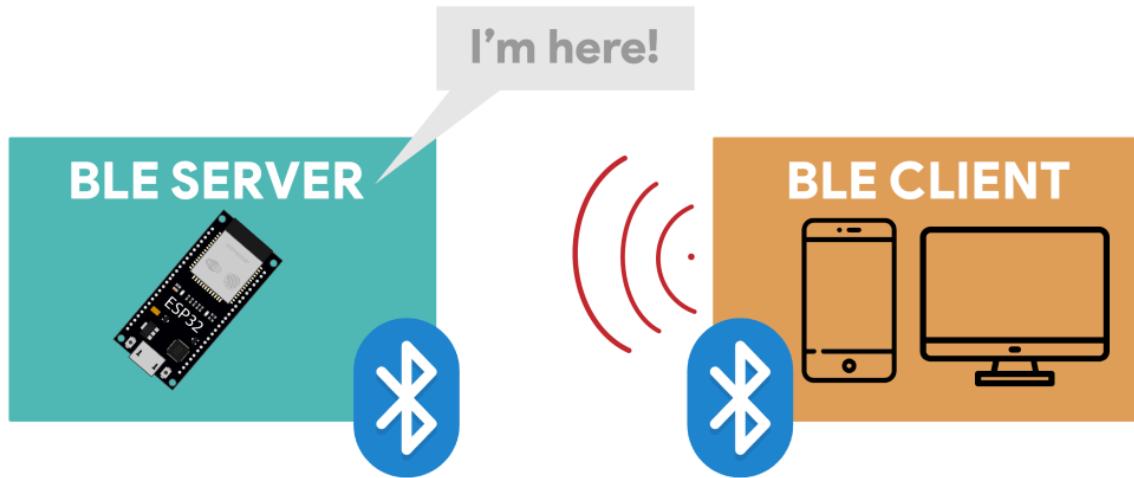
The ESP32 can act either as a Peripheral or as a central device. When it acts as a peripheral it sets up a GATT profile and advertises its service with characteristics that the central devices can read. On the other hand, when it is set as a central device, it can connect to other BLE devices to read or interact with their profiles and read their characteristics.

In the above diagram, the ESP32 takes the role of the BLE Peripheral, serving as the device that provides data or services. Your smartphone or computer acts as the BLE Controller, managing the connection and communication with the ESP32.

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. In the picture below it

acts as a server, exposing its GATT structure containing data. The BLE Server acts as a provider of data or services, while the BLE Client consumes or uses these services.

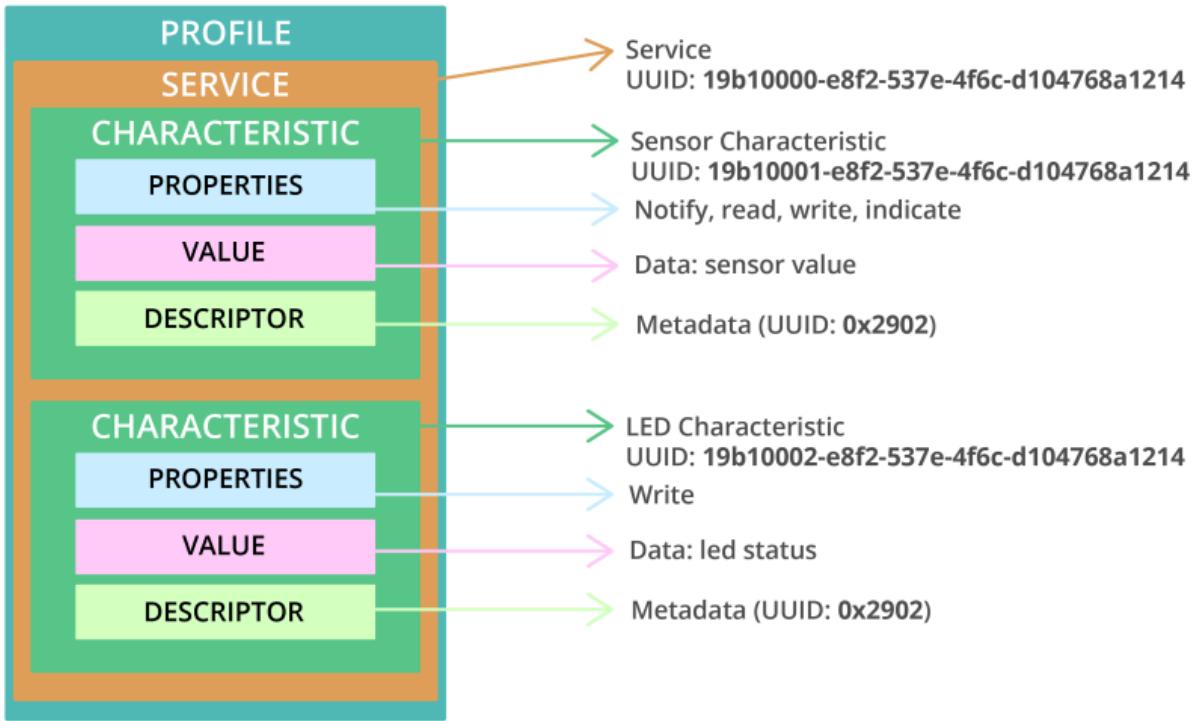


The server advertises its existence, so it can be found by other devices and contains data that the client can read or interact with. The client scans the nearby devices, and when it finds the server it is looking for, it establishes a connection and can interact with that device by reading or writing on its characteristics.

The BLE server is basically the BLE peripheral before establishing a connection. The BLE Client is the BLE controller before establishing a connection. Many times, these terms are used interchangeably.

GATT

GATT, which stands for Generic Attribute Profile, is a fundamental concept in Bluetooth Low Energy (BLE) technology. Essentially, it serves as a blueprint for how BLE devices communicate with each other. Think of it as a structured language that two BLE devices use to exchange information seamlessly.



- **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.

Let's take a more in-depth detail at the BLE Service and Characteristics.

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, like battery service and heart rate service.

Every service contains at least one characteristic. 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, Environmental Sensing, etc. You can check on the following link predefined services:

- <https://www.bluetooth.com/specifications/assigned-numbers/>

UUID

A UUID is a unique digital identifier used in BLE and GATT to distinguish and locate services, characteristics, and descriptors. It's like a distinct label that ensures every component in a Bluetooth device has a unique name.

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 and default UUIDs for services, and characteristics specified in the [SIG \(Bluetooth Special Interest Group\)](#). This means, that if you have a BLE device that uses the default UUIDs for its services and characteristics, you'll know exactly how to interact with that device to get or interact with the information you're looking for.

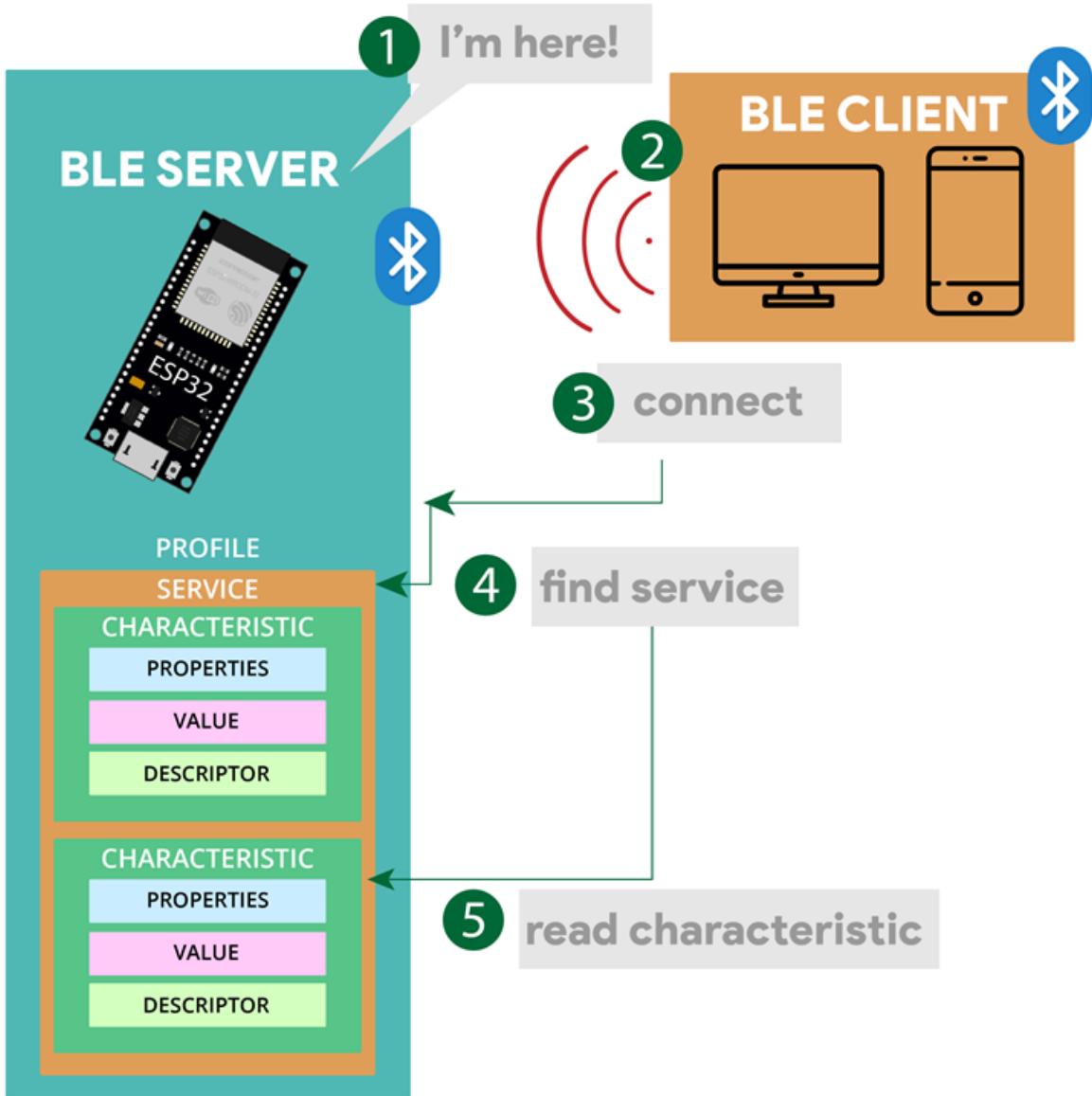
You can also generate your own custom UUIDs if you don't want to stick with predefined values or if the data, you're exchanging doesn't fit in any of the categories.

You can generate custom UUIDs using this [UUID generator website](#).

Communication between BLE Devices

Here are the usual steps that describe the communication between BLE Devices.

1. The BLE Peripheral (server) advertises its existence.
2. The BLE Central Device (client) scans for BLE devices.
3. When the central device finds the peripheral, it is looking for, it establishes a connection with it.
4. After connecting, it reads the GATT profile of the peripheral and searches for the service it is looking for (for example: *environmental sensing*).
5. If it finds the service, it can now interact with the characteristics. For example, reading the temperature value.



Wrapping Up

In this Unit, we covered the basic theory behind a BLE device and the interactions between a client and server and some of the standards you need to pay attention to when it comes to BLE.

It's important to note that we've only touched on the basics of BLE, and there are many more concepts and features to explore. We've focused on the aspects most relevant to our projects.

7.2 - Bluetooth Low Energy: BLE Server, Scanner, and Notify

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.

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 (client) to find that server. We'll also create an ESP32 BLE server that sends notifications. We'll use and explain the examples that come with the BLE library.

ESP32 BLE Server

To create the ESP32 BLE Server, open your Arduino IDE and go to **File > Examples > BLE** and select the **Server** example, or copy the code from the link below.

- [Click here to download the code.](#)

```
#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-c5c9c331914b"
#define CHARACTERISTIC_UUID    "beb5483e-36e1-4688-b7f5-ea07361b26a8"

void setup() {
  Serial.begin(115200);
  Serial.println("Starting BLE work!");

  BLEDevice::init("MyESP32");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  BLECharacteristic *pCharacteristic =
    pService->createCharacteristic(CHARACTERISTIC_UUID,
                                     BLECharacteristic::PROPERTY_READ |
                                     BLECharacteristic::PROPERTY_WRITE);

  pCharacteristic->setValue("Hello World says Neil");
  pService->start();
  // this still is working for backward compatibility
  // BLEAdvertising *pAdvertising = pServer->getAdvertising();
```

```

BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
pAdvertising->setScanResponse(true);
// functions that help with iPhone connections issue
pAdvertising->setMinPreferred(0x06);
pAdvertising->setMinPreferred(0x12);
BLEDevice::startAdvertising();
Serial.println("Characteristic defined! Now you can read it in your phone!");
}

void loop() {
    // put your main code here, to run repeatedly:
    delay(2000);
}

```

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 Does the Code Work?

Let's take a quick look at how the BLE server example code works.

It starts by importing the necessary libraries to use BLE with the ESP32.

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
```

Then, you need to define a UUID for the Service and another one for the 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, we create a BLE device called "MyESP32". You can change this name to whatever you like.

```
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. You also use the UUID defined earlier for the characteristic. You need to pass as arguments the characteristic's properties. In this case, it's: READ and WRITE.

```
BLECharacteristic *pCharacteristic =
    pService->createCharacteristic(CHARACTERISTIC_UUID,
        BLECharacteristic::PROPERTY_READ |  

        BLECharacteristic::PROPERTY_WRITE
    );
```

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 you like. In future projects, this text can be a sensor reading, or the state of a lamp, for example.

Then, we start the service. `pService` is a pointer to a BLE service that we created previously. The `start()` function begins that service, making it available for clients (such as a smartphone app) to discover and interact with.

```
pService->start();
```

BLE devices use advertising to broadcast their services to nearby devices. We create a pointer to an advertising object as follows:

```
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
```

Then, add the Service UUID to the advertising packet. This will let other BLE devices know that this service is available.

```
pAdvertising->addServiceUUID(SERVICE_UUID);
```

Set the `setScanResponse` to `true` to enable a scan response that allows additional data to be sent in response to a scan request from another device.

```
pAdvertising->setScanResponse(true);
```

Then, the following lines with `setMinPreferred(0x06)` and `setMinPreferred(0x12)` adjust the preferred connection interval for the BLE connection. These values help address connection issues, particularly with iPhones, ensuring more stable and reliable connections.

```
// functions that help with iPhone connections issue
pAdvertising->setMinPreferred(0x06);
pAdvertising->setMinPreferred(0x12);
```

The connection interval is the time between consecutive connection events, with smaller intervals allowing for more frequent communication. These values are hex representations that correspond to specific time intervals.

Finally, start the BLE advertising process. `startAdvertising()` begins the BLE advertising process. This makes the ESP32 discoverable to other BLE devices, allowing them to see the service and its characteristics.

```
BLEDevice::startAdvertising();
```

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 (we'll cover this next in the *Notify Example*).

Uploading the Code

Upload the code to your ESP32 board. Open the Serial Monitor at a baud rate of 115200. Press the ESP32 RST/EN button. The ESP32 should start the BLE advertising successfully.

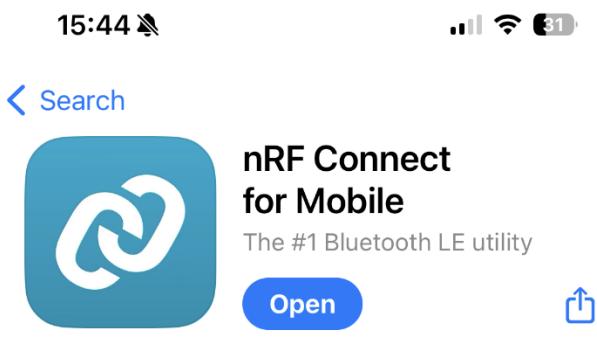


The screenshot shows the Arduino Serial Monitor window. The top bar has tabs for "Output" and "Serial Monitor". The main area displays the message: "Starting BLE work! Characteristic defined! Now you can read it in your phone!". Below the message, there are buttons for "New Line" and "115200 baud".

Testing the ESP32 BLE Server with Your Smartphone

Most modern smartphones should have BLE capabilities. 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.

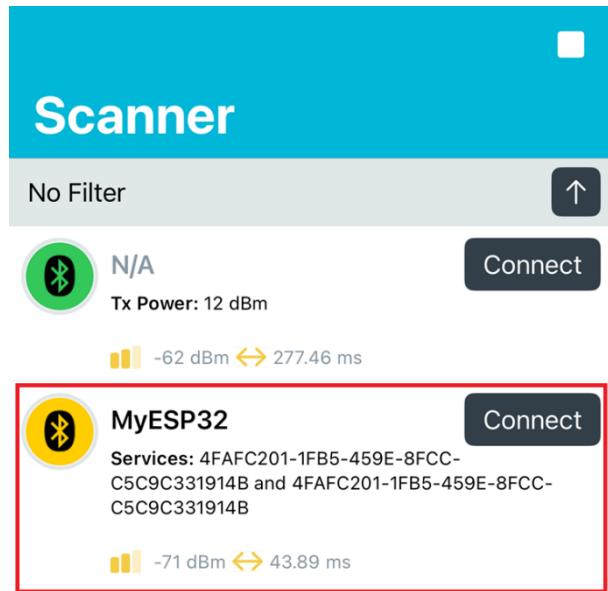


Open the *nRF Connect for Mobile* app on your smartphone. Make sure you have Bluetooth enabled.

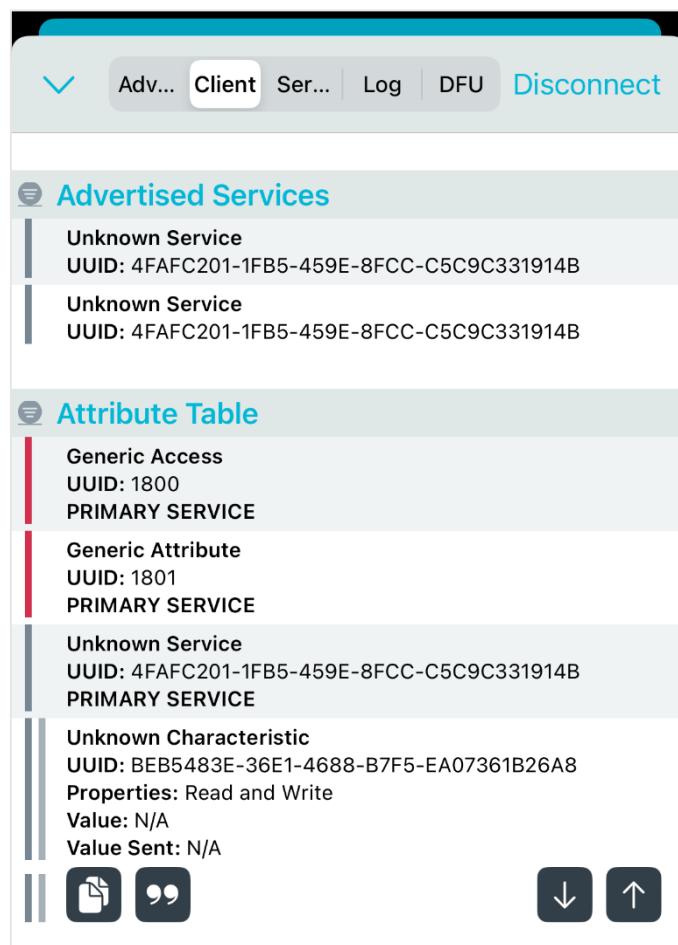
Once everything is ready in your smartphone and the ESP32 is running the BLE server sketch, in the app, tap the scan button to scan for nearby devices (or it will automatically start scanning new devices).

The app is slightly different on iOS and Android.

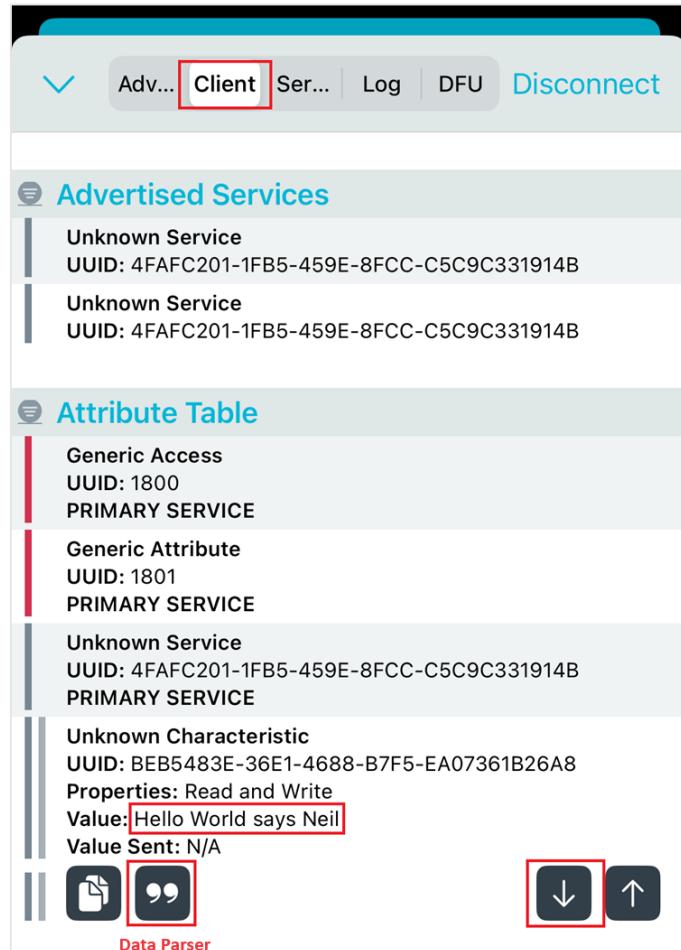
You should find an ESP32 with the name “MyESP32”.



Click the “Connect” button to connect to the ESP32. It should have an unknown service and characteristic with the UUIDs that you’ve defined earlier. Click on the arrow that is facing down to read the characteristic.



On iPhones, you'll need to change the format of the received characteristic value by clicking on the second symbol at the left (data parser) and then selecting UTF-8. After that, you should be able to read the ESP32 characteristic you defined earlier in the code.



Note that this characteristic has the READ and WRITE properties exactly like you've defined previously defined in the BLE server sketch. So, everything is working fine.

BLE Scanner

Keep the ESP32 board running the previous BLE sketch, but disconnected from the smartphone. We'll now run a scanner sketch on another ESP32 board to see if it finds the other one.

Open the code available in the Arduino IDE at: **File > Examples > BLE > Scan** or copy the code provided in the link below.

- [Click here to download the code.](#)

The following sketch scans for nearby devices. When it finds a device, it displays its info on the serial monitor.

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

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

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

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

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

void loop() {
    // put your main code here, to run repeatedly:
    BLEScanResults *foundDevices = pBLEScan->start(scanTime, false);
    Serial.print("Devices found: ");
    Serial.println(foundDevices->getCount());
    Serial.println("Scan done!");
    pBLEScan->clearResults(); // delete results fromBLEScan buffer to release memory
    delay(2000);
}
```

Testing the BLE Scanner

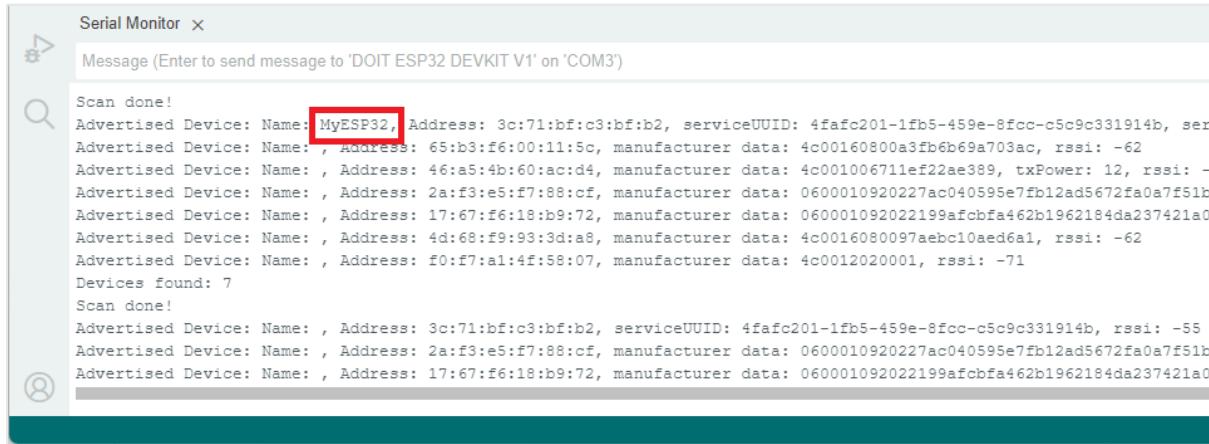
Make sure you have the right board and COM port selected for this new ESP32.

Once the code is uploaded, you should have the two ESP32 boards powered on:

- 1) One ESP32 with the “Server” sketch
- 2) Other with ESP32 “Scan” sketch

Go to the Serial Monitor with the ESP32 running the “Scan” example, press the ESP32 (with the “Scan” sketch) RESET/EN button to restart, and wait a few seconds while it scans.

After a few seconds, it should find a device called “MyESP32” or whatever name you’ve given to your device—it is the ESP32 is running the “Server” sketch.



```
Serial Monitor < X >
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'COM3')

Scan done!
Advertised Device: Name: MyESP32, Address: 3c:71:bf:c3:bf:b2, serviceUUID: 4fafc201-1fb5-459e-8fcc-c5c9c331914b, ser
Advertised Device: Name: , Address: 65:b3:f6:00:11:5c, manufacturer data: 4c00160800a3fb6b69a703ac, rssi: -62
Advertised Device: Name: , Address: 46:a5:4b:60:ac:d4, manufacturer data: 4c00100671lef2ae389, txPower: 12, rssi: -
Advertised Device: Name: , Address: 2a:f3:e5:f7:88:cf, manufacturer data: 0600010920227ac040595e7fb12ad5672fa0a7f51b
Advertised Device: Name: , Address: 17:67:f6:18:b9:72, manufacturer data: 060001092022199afcbfa462b1962184da237421a0
Advertised Device: Name: , Address: 4d:68:f9:93:3d:a8, manufacturer data: 4c0016080097aebc10aed6a1, rssi: -62
Advertised Device: Name: , Address: f0:f7:a1:4f:58:07, manufacturer data: 4c0012020001, rssi: -71
Devices found: 7
Scan done!
```

The “Scan” sketch also provides additional information, such as address and transmit power level (txPower).

Notify Example

In this example, you’ll set the ESP32 as a BLE server that sends a notify message with a counter value when a client is connected. We’ll use the example that comes with the [ESP32 BLE library for Arduino IDE](#).

We’ll use the example available at **File > Examples > BLE > Notify**. You can copy from the download link provided below.

- [Click here to download the code.](#)

```

#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <BLE2901.h>

BLEServer *pServer = NULL;
BLECharacteristic *pCharacteristic = NULL;
BLE2901 *descriptor_2901 = NULL;

bool deviceConnected = false;
bool oldDeviceConnected = false;
uint32_t value = 0;

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

#define SERVICE_UUID      "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

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

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

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

    // Create the BLE Device
    BLEDevice::init("ESP32");

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

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

    // Create a BLE Characteristic
    pCharacteristic = pService->createCharacteristic(
        CHARACTERISTIC_UUID,
        BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_WRITE |
            BLECharacteristic::PROPERTY_NOTIFY |
            BLECharacteristic::PROPERTY_INDICATE
    );

    // Creates BLE Descriptor 0x2902: Client Characteristic Configuration Descriptor (CCCD)
    pCharacteristic->addDescriptor(new BLE2902());
    // Adds also the Characteristic User Description - 0x2901 descriptor
    descriptor_2901 = new BLE2901();
    descriptor_2901->setDescription("My own description for this characteristic.");
    // enforce read only - default is Read|Write
    descriptor_2901->setAccessPermissions(ESP_GATT_PERM_READ);
    pCharacteristic->addDescriptor(descriptor_2901);

    // Start the service
}

```

```

pService->start();

// Start advertising
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
pAdvertising->setScanResponse(false);
pAdvertising->setMinPreferred(0x0); // set value to 0x00 to not advertise this parameter
BLEDevice::startAdvertising();
Serial.println("Waiting a client connection to notify...");
}

void loop() {
    // notify changed value
    if (deviceConnected) {
        pCharacteristic->setValue((uint8_t *)&value, 4);
        pCharacteristic->notify();
        value++;
        delay(500);
    }
    // disconnecting
    if (!deviceConnected && oldDeviceConnected) {
        delay(500); // give the bluetooth stack the chance to get things ready
        pServer->startAdvertising(); // restart advertising
        Serial.println("start advertising");
        oldDeviceConnected = deviceConnected;
    }
    // connecting
    if (deviceConnected && !oldDeviceConnected) {
        // do stuff here on connecting
        oldDeviceConnected = deviceConnected;
    }
}

```

How Does the Code Work?

Let's take a look at this example.

It starts by importing the necessary libraries for BLE.

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <BLE2901.h>
```

BLE2902 and BLE2901

The `BLE2902.h` is used to handle the Client Characteristic Configuration Descriptor (CCCD), which has a UUID of 0x2902. This is a special descriptor that allows the client (typically a smartphone or other BLE-enabled device) to enable or disable notifications or indications from a characteristic.

Notification: The client receives updates automatically whenever the characteristic's value changes.

Indication: Similar to notifications but requires an acknowledgment from the client.

The `BLE2901.h` is used to manage the Characteristic User Description Descriptor (CUDD), which has a UUID of 0x2901. It provides a human-readable description of the characteristic. This descriptor is mainly for user interfaces—it allows connected devices (like a smartphone app) to display a more meaningful name for the characteristic, rather than just showing a UUID.

Then, create pointers to a BLE server, BLE characteristic and the BLE2901 descriptor.

```
BLEServer *pServer = NULL;  
BLECharacteristic *pCharacteristic = NULL;  
BLE2901 *descriptor_2901 = NULL;
```

Then, create some variables to handle the connection between the client and server:

```
bool deviceConnected = false;  
bool oldDeviceConnected = false;  
uint32_t value = 0;
```

The `value` variable is used to set the Characteristic value. This is the value that we'll send to the client.

```
uint32_t value = 0;
```

Then, the following lines defines 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.

setup()

Scroll down to the `setup()` function.

You start the serial communication at a baud rate of 115200.

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

Then, you create a BLE device called "ESP32". You can change this to whatever you like.

```
BLEDevice::init("ESP32"); // Create the BLE Device
```

In the following line, you set the BLE device as a server.

```
pServer = BLEDevice::createServer();
```

Assign a callback function to the server (MyServerCallbacks()).

```
pServer->setCallbacks(new MyServerCallbacks());
```

Basically, upon a successful connection, the boolean variable deviceConnected changes to true, and when a client disconnects it changes the boolean variable to false.

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

After that, you create a BLE service for the server that has the UUID defined earlier in the code.

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

Then, you set the characteristic for that service. You use the UUID defined earlier for the characteristic. You also need to pass as arguments the properties for this characteristic. In this case, it's: READ, WRITE, NOTIFY, and INDICATE.

```
// Create a BLE Characteristic
pCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_UUID,
    BLECharacteristic::PROPERTY_READ |
    BLECharacteristic::PROPERTY_WRITE |
    BLECharacteristic::PROPERTY_NOTIFY |
    BLECharacteristic::PROPERTY_INDICATE);
```

You also create a BLE descriptor for the characteristic.

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

This is the Client Characteristic Configuration Descriptor (CCCD), which has a UUID of 0x2902. This is a special descriptor that allows the client (typically a smartphone or other BLE-enabled device) to enable or disable notifications or indications from a characteristic.

We also add a user description for the characteristic using the 0x2901 descriptor.

```
descriptor_2901 = new BLE2901();
descriptor_2901->setDescription("My own description for this characteristic.");
```

The following line enforces that the descriptor description is read-only.

```
// enforce read only - default is Read|Write
descriptor_2901->setAccessPermissions(ESP_GATT_PERM_READ);
```

Then, we add the descriptor to our characteristic.

```
pCharacteristic->addDescriptor(descriptor_2901);
```

Finally, you can start the service, and the advertising, so other BLE devices can scan and find this BLE device, like we did in the previous example.

```
// Start advertising
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
pAdvertising->setScanResponse(false);
pAdvertising->setMinPreferred(0x0); // set value to 0x00 to not advertise this parameter
BLEDevice::startAdvertising();
Serial.println("Waiting a client connection to notify...");
```

loop()

In the `loop()`, you check if a device is connected or not and act accordingly.

If a device is connected, set a new value for the characteristic, notify the client, and then, increment the value.

```
if (deviceConnected) {
    pCharacteristic->setValue((uint8_t *)&value, 4);
    pCharacteristic->notify();
    value++;
    delay(500);
}
```

If the BLE Client has just disconnected from the server, it restarts BLE advertising, making the server discoverable again to other BLE devices. This allows a new client to connect.

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

Finally, this last block is executed when a BLE client has just connected to the server. It allows you to perform any necessary actions that should happen right when the connection is established.

```
if (deviceConnected && !oldDeviceConnected) {  
    // do stuff here on connecting  
    oldDeviceConnected = deviceConnected;  
}
```

Testing the Notify Example

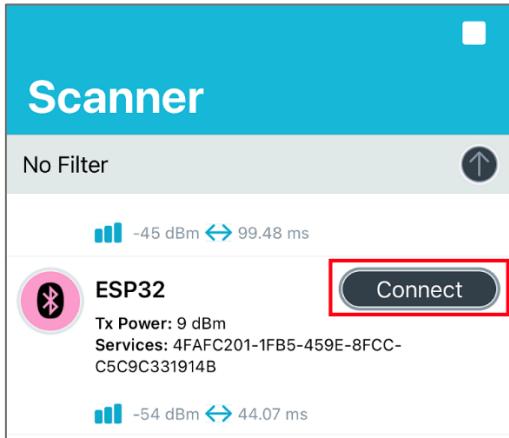
Upload the previous code to your ESP32. Make sure you have the right board and COM port selected. Once the code is uploaded, open the Serial Monitor at a baud rate of 115200.



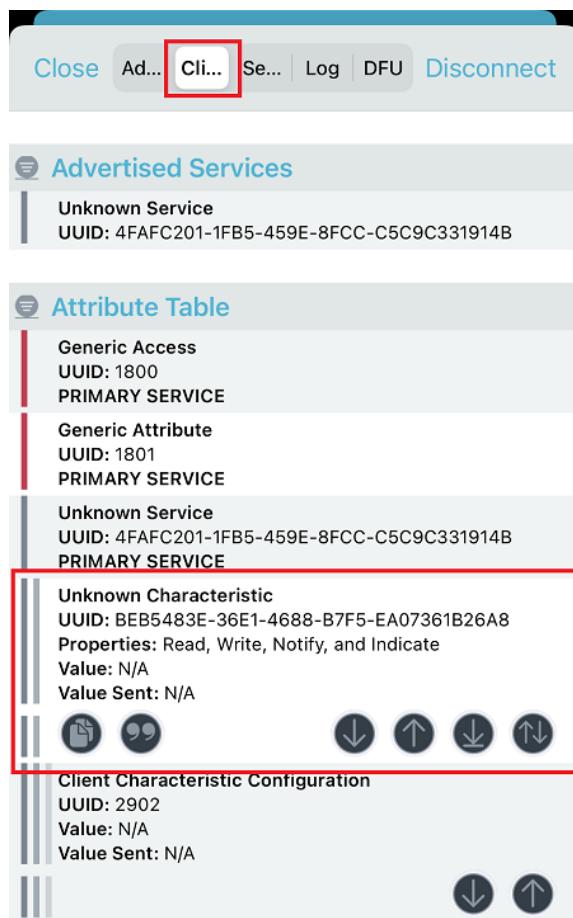
Leave the Serial Monitor window open...

Connecting With Your Smartphone

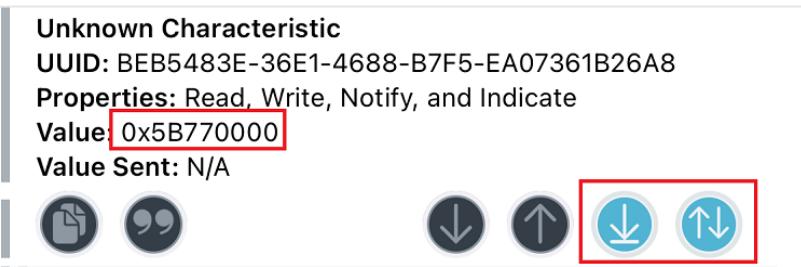
Open the *nRF Connect for mobile* app on your smartphone and scan for nearby devices. It should find an ESP32 with the name “**ESP32**” or whatever name you’ve defined for your device.



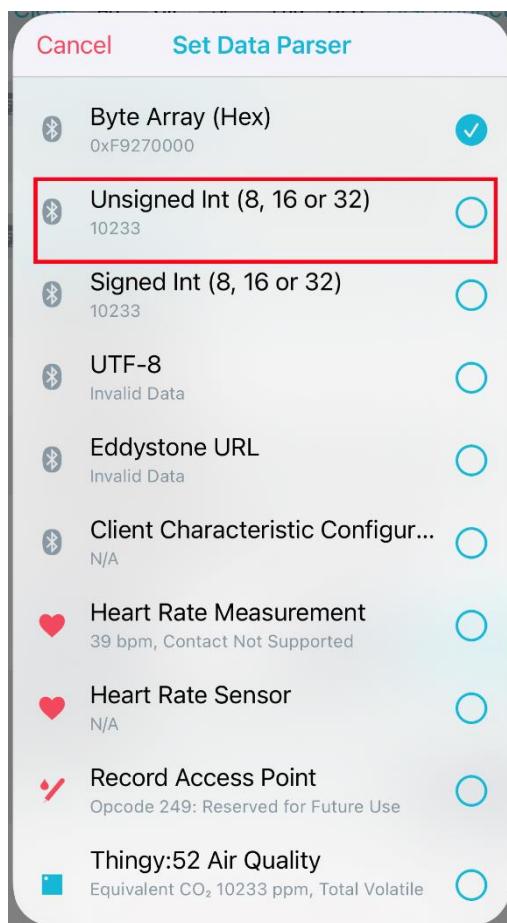
Click the “**Connect**” button. Then, click on the “**Client**” tab. You should see the ESP32 BLE profile with its service and characteristic with the UUID that you’ve defined in the code. If you tap the service, it will expand the menu.



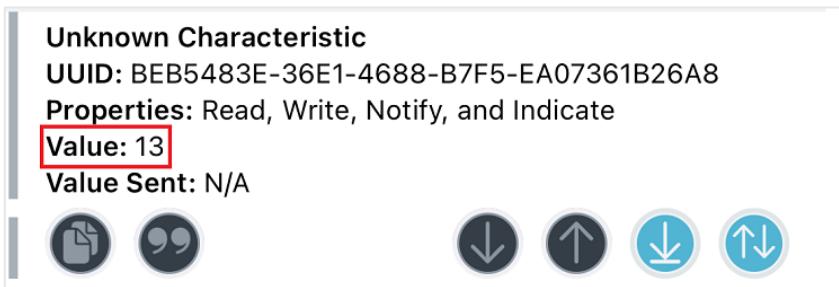
As you can see, the characteristic has 4 properties: **Indicate**, **Notify**, **Read**, and **Write**. The four buttons allow you to read the current characteristic value, write a new characteristic value, enable the notify property, or enable the indicate property. Click on the notify button to activate the notifications.



You'll start receiving new values on the **Value** field. Those values are in hexadecimal. To change to decimal click on the data parser button (second button at the left). Then, change the data parser to unsigned int.



Then, you'll receive the values in decimal.



If you disconnect your smartphone, the ESP32 will print in the serial monitor the message "start advertising" so that other BLE devices can connect.



Output Serial Monitor X

Message (Enter to send message to 'DOIT ESP32 DEVKIT V1')

New Line 115200 baud

```
Waiting a client connection to notify...
start advertising
```

Ln 110, Col 4 DOIT ESP32 DEVKIT V1 on COM3 2 404

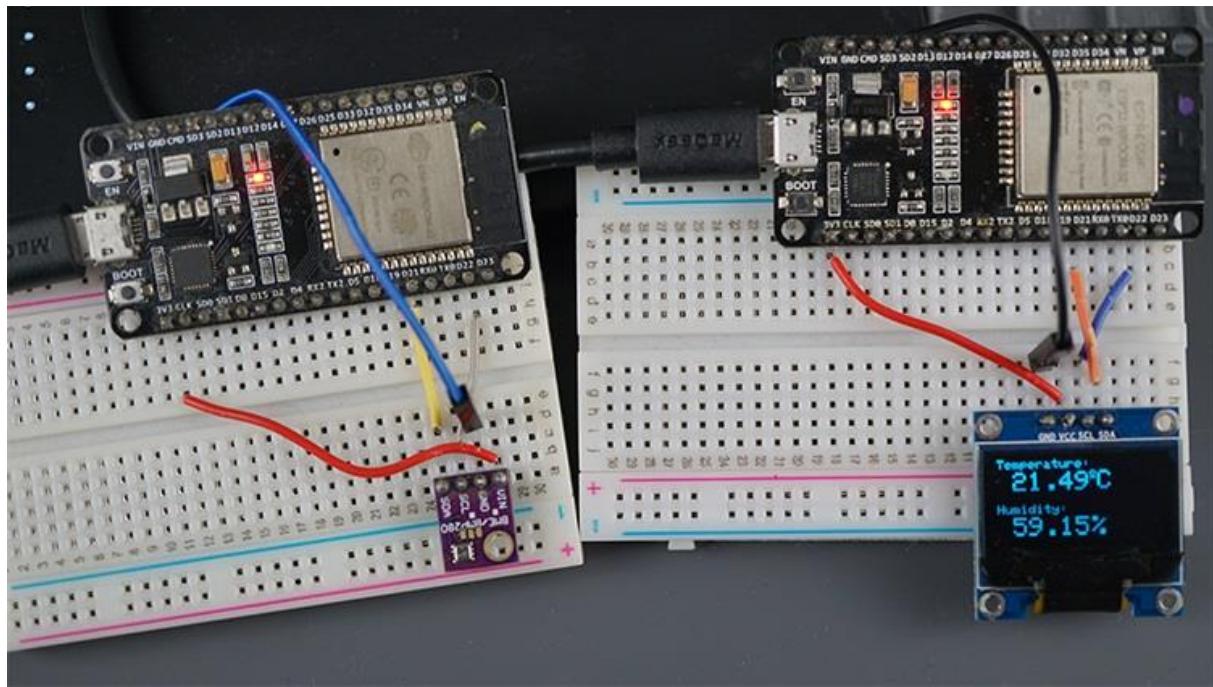
Wrapping Up

In this Unit we've taken a look at some of the basic BLE examples for the ESP32.

Instead of using your smartphone app to read the characteristic of the ESP32, you can set another ESP32 as a BLE client to read the values advertised by the Notify example. We'll cover something similar in the next two units. We'll set up an ESP32 as a client and another ESP32 as a server so that they exchange data via BLE.

7.3 - ESP32 BLE Server and Client (Part 1/2)

In this Unit (and following), you're going to learn how to make a Bluetooth connection between two ESP32 boards. One ESP32 is going to be the server, and the other ESP32 will be the 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** – Prepare the BLE server
- **Part 2** – Prepare the BLE client

The 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.

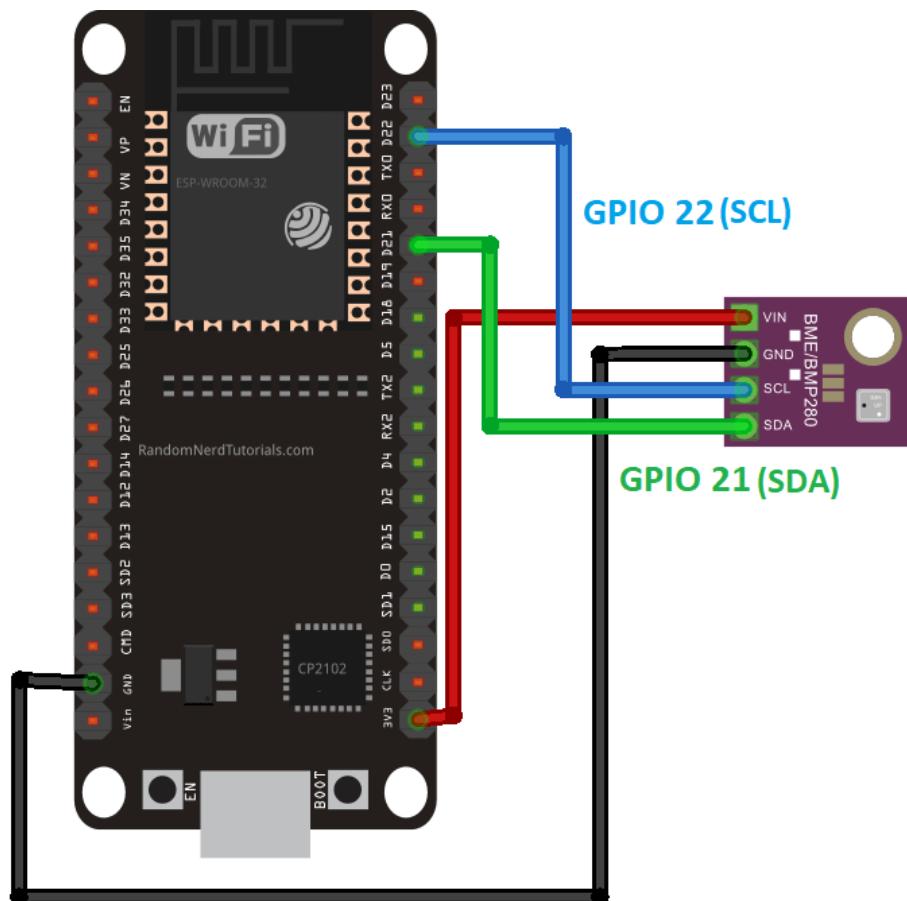
BLE Server - Parts Required

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [DHT11/DHT22](#) temperature and humidity sensor
- [4.7k Ohm resistor](#)
- [Jumper wires](#)
- [Breadboard](#)
- Smartphone with Bluetooth

Wiring the Circuit

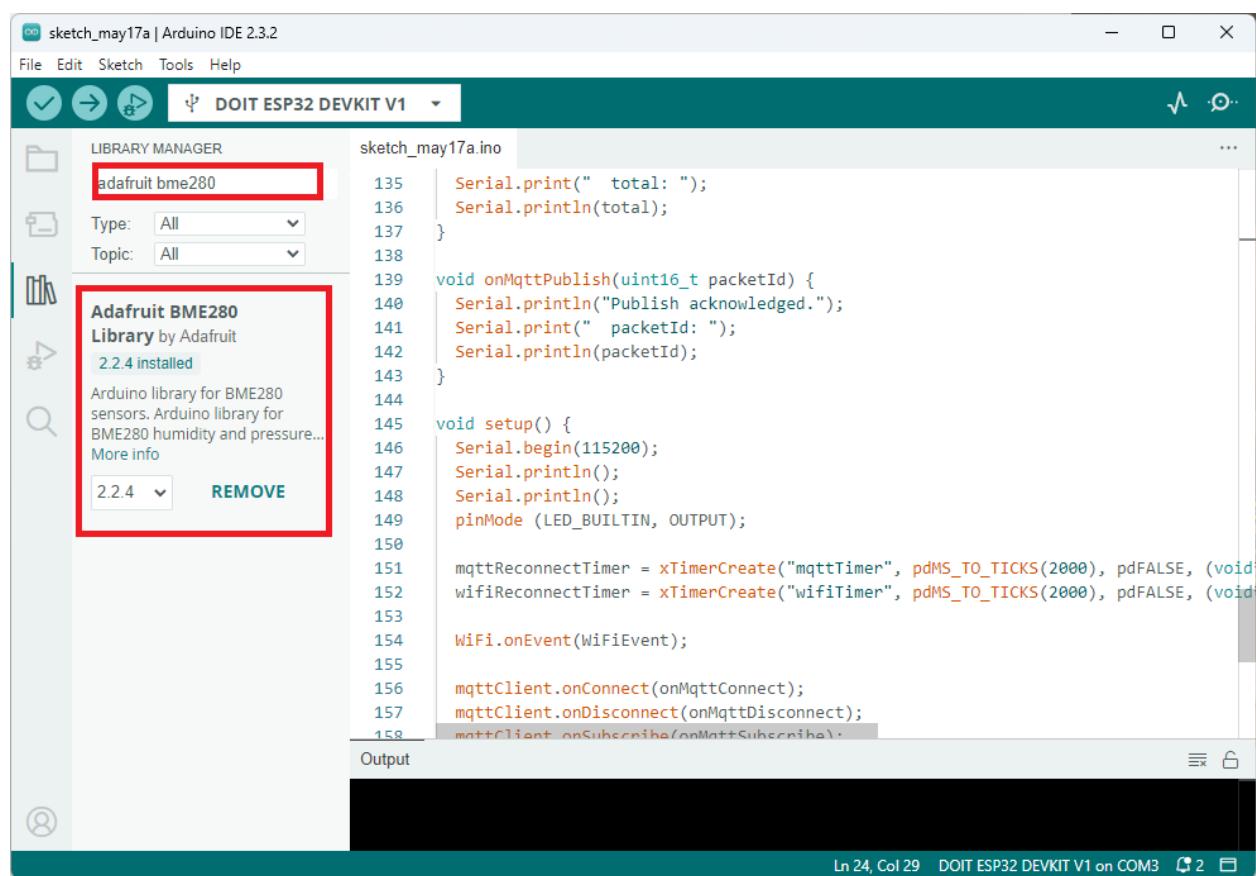
The ESP32 BLE server will advertise characteristics with temperature and humidity from a BME280 sensor. You can use any other sensor as long as you add the required lines in the code.



Installing the BME280 Sensor Library

To read from the BME280 sensor, we'll use the [Adafruit BME280 library](#) to get data from the BME280. Follow the next steps to install the library.

- 1) In the Arduino IDE, go to **Sketch > Include Library > Manage Libraries**.
The Library Manager should open.
- 2) Search for **adafruit bme280** on the search box and install the library. Also install any dependencies currently not installed (usually the Adafruit Bus IO and the Adafruit Unified Sensor libraries).



- 3) After installing the libraries, restart your Arduino IDE.

ESP32 Server Sketch

With the circuit ready and the needed libraries installed, open your Arduino IDE and copy the code provided in the link below.

- [Click here to download the code.](#)

```

#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;
float hum;

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;

bool deviceConnected = false;

// See the following for generating UUIDs:
// https://www.uuidgenerator.net/
#define SERVICE_UUID "91bad492-b950-4226-aa2b-4ede9fa42f59"

// Temperature Characteristic and Descriptor
#ifndef temperatureCelsius
    BLECharacteristic bmeTemperatureCelsiusCharacteristics(
        "cba1d466-344c-4be3-ab3f-189f80dd7518",
        BLECharacteristic::PROPERTY_READ |
        BLECharacteristic::PROPERTY_NOTIFY);
    BLEDescriptor bmeTemperatureCelsiusDescriptor(BLEUUID((uint16_t)0x2902));
#else
    BLECharacteristic bmeTemperatureFahrenheitCharacteristics(
        "f78ebbf-f-c8b7-4107-93de-889a6a06d408",
        BLECharacteristic::PROPERTY_READ |
        BLECharacteristic::PROPERTY_NOTIFY);
    BLEDescriptor bmeTemperatureFahrenheitDescriptor(BLEUUID((uint16_t)0x2902));
#endif

// Humidity Characteristic and Descriptor
BLECharacteristic bmeHumidityCharacteristics(
    "ca73b3ba-39f6-4ab3-91ae-186dc9577d99",
    BLECharacteristic::PROPERTY_READ |
    BLECharacteristic::PROPERTY_NOTIFY);
BLEDescriptor bmeHumidityDescriptor(BLEUUID((uint16_t)0x2902));

// Setup callbacks onConnect and onDisconnect
class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
        deviceConnected = true;
    };
    void onDisconnect(BLEServer* pServer) {
        deviceConnected = false;
    }
}

```

```

};

void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}

void setup() {
    // Start serial communication
    Serial.begin(115200);

    // Init BME Sensor
    initBME();

    // Create the BLE Device
    BLEDevice::init(bleServerName);

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

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

    // Create BLE Characteristics and Create a BLE Descriptor
    // Temperature
    #ifdef temperatureCelsius
        bmeService->addCharacteristic(&bmeTemperatureCelsiusCharacteristics);
        bmeTemperatureCelsiusCharacteristics.addDescriptor(&bmeTemperatureCelsiusDescriptor);
    #else
        bmeService->addCharacteristic(&bmeTemperatureFahrenheitCharacteristics);
        bmeTemperatureFahrenheitCharacteristics.addDescriptor(&bmeTemperatureFahrenheitDescriptor);
    #endif

    // Humidity
    bmeService->addCharacteristic(&bmeHumidityCharacteristics);
    bmeHumidityCharacteristics.addDescriptor(&bmeHumidityDescriptor);

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

    // Start advertising
    BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
    pAdvertising->addServiceUUID(SERVICE_UUID);
    pAdvertising->setScanResponse(true);
    pAdvertising->setMinPreferred(0x06); // functions that help with iPhone connections issue
    pAdvertising->setMinPreferred(0x12);

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

void loop() {
    if (deviceConnected) {
        if ((millis() - lastTime) > timerDelay) {
            // Read temperature as Celsius (the default)
            temp = bme.readTemperature();
            // Fahrenheit
            tempF = 1.8 * temp + 32;
        }
    }
}

```

```

// Read humidity
hum = bme.readHumidity();

// Notify temperature reading from BME sensor
#ifndef temperatureCelsius
    static char temperatureCTemp[6];
    dtostrf(temp, 6, 2, temperatureCTemp);
    // Set temperature Characteristic value and notify connected client
    bmeTemperatureCelsiusCharacteristics.setValue(temperatureCTemp);
    bmeTemperatureCelsiusCharacteristics.notify();
    Serial.print("Temperature Celsius: ");
    Serial.print(temp);
    Serial.print(" °C");
#else
    static char temperatureFTemp[6];
    dtostrf(tempF, 6, 2, temperatureFTemp);
    // Set temperature Characteristic value and notify connected client
    bmeTemperatureFahrenheitCharacteristics.setValue(temperatureFTemp);
    bmeTemperatureFahrenheitCharacteristics.notify();
    Serial.print("Temperature Fahrenheit: ");
    Serial.print(tempF);
    Serial.print(" °F");
#endif

// Notify humidity reading from BME
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(" %");

lastTime = millis();
}
}
}

```

You can upload the code, and it will work straight away, reporting the temperature in Celsius. But if you want to know how the code works, continue reading.

How Does the Code Work?

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.

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's 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 to any other value.

```
// 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. We set the characteristics properties to `READ` and `NOTIFY`. We also set the BLE descriptors for our characteristics.

```
// See the following for generating UUIDs:  
// https://www.uuidgenerator.net/  
#define SERVICE_UUID "91bad492-b950-4226-aa2b-4ede9fa42f59"  
  
// Temperature Characteristic and Descriptor  
#ifdef temperatureCelsius  
    BLECharacteristic bmeTemperatureCelsiusCharacteristics(  
        "cba1d466-344c-4be3-ab3f-189f80dd7518",  
        BLECharacteristic::PROPERTY_READ |  
        BLECharacteristic::PROPERTY_NOTIFY);  
    BLEDescriptor bmeTemperatureCelsiusDescriptor(BLEUUID((uint16_t)0x2902));  
#else  
    BLECharacteristic bmeTemperatureFahrenheitCharacteristics(  
        "f78ebbf-f-c8b7-4107-93de-889a6a06d408",  
        BLECharacteristic::PROPERTY_READ |  
        BLECharacteristic::PROPERTY_NOTIFY);  
    BLEDescriptor bmeTemperatureFahrenheitDescriptor(BLEUUID((uint16_t)0x2902));  
#endif  
  
// Humidity Characteristic and Descriptor  
BLECharacteristic bmeHumidityCharacteristics(  
    "ca73b3ba-39f6-4ab3-91ae-186dc9577d99",  
    BLECharacteristic::PROPERTY_READ |  
    BLECharacteristic::PROPERTY_NOTIFY);  
BLEDescriptor bmeHumidityDescriptor(BLEUUID((uint16_t)0x2902));
```

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.

A **BLE descriptor** provides additional information about a characteristic and how it should be accessed or interpreted. Our descriptors have the default UUID `0x2902` and it represents the “Client Characteristic Configuration” descriptor.

The “Client Characteristic Configuration” descriptor, with UUID 0x2902, is commonly used in BLE to configure how notifications and indications are handled by the client (usually a central device like a smartphone) when it subscribes to a characteristic’s notifications.

This descriptor allows the client to configure how it wants to receive updates (e.g., notifications) from the associated characteristic, giving more control over the communication between the ESP32 and the connected BLE devices.

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(&bmeTemperatureCelsiusCharacteristics);
    bmeTemperatureCelsiusCharacteristics.addDescriptor(&bmeTemperatureCelsiusDescriptor);
```

Otherwise, it sets the Fahrenheit characteristic:

```
#else
    bmeService->addCharacteristic(&bmeTemperatureFahrenheitCharacteristics);
    bmeTemperatureFahrenheitCharacteristics.addDescriptor(&bmeTemperatureFahrenheitDescriptor);
#endif
```

After that, it adds the humidity characteristic to the service and sets its descriptor.

```
bmeService->addCharacteristic(&bmeHumidityCharacteristics);
bmeHumidityCharacteristics.addDescriptor(&bmeHumidityDescriptor);
```

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);
pAdvertising->setScanResponse(true);
pAdvertising->setMinPreferred(0x06); // functions that help with iPhone connections issue
pAdvertising->setMinPreferred(0x12);

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 = 1.8 * temp + 32;
```

```
// Read humidity  
hum = bme.readHumidity();
```

If you're using temperature in Celsius it runs the following code section.

```
// Notify temperature reading from BME sensor  
#ifdef temperatureCelsius  
    static char temperatureCTemp[6];  
    dtostrf(temp, 6, 2, temperatureCTemp);  
    // Set temperature Characteristic value and notify connected client  
    bmeTemperatureCelsiusCharacteristics.setValue(temperatureCTemp);  
    bmeTemperatureCelsiusCharacteristics.notify();  
    Serial.print("Temperature Celsius: ");  
    Serial.print(temp);  
    Serial.print(" °C");
```

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 client  
bmeTemperatureCelsiusCharacteristics.setValue(temperatureCTemp);  
bmeTemperatureCelsiusCharacteristics.notify();
```

We follow a similar procedure for the Temperature in Fahrenheit.

```
#else  
    static char temperatureFTemp[6];  
    dtostrf(tempF, 6, 2, temperatureFTemp);  
    // Set temperature Characteristic value and notify connected client  
    bmeTemperatureFahrenheitCharacteristics.setValue(temperatureFTemp);  
    bmeTemperatureFahrenheitCharacteristics.notify();  
    Serial.print("Temperature Fahrenheit: ");  
    Serial.print(tempF);  
    Serial.print(" °F");  
#endif
```

Setting the humidity characteristic value also uses the same process.

```
// Notify humidity reading from BME  
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(" %");
```

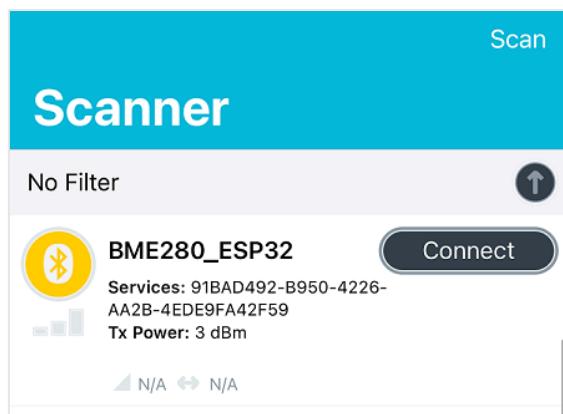
Testing the ESP32 BLE Server

Upload the code to your board and then, open the Serial Monitor. It will display a message as shown below.



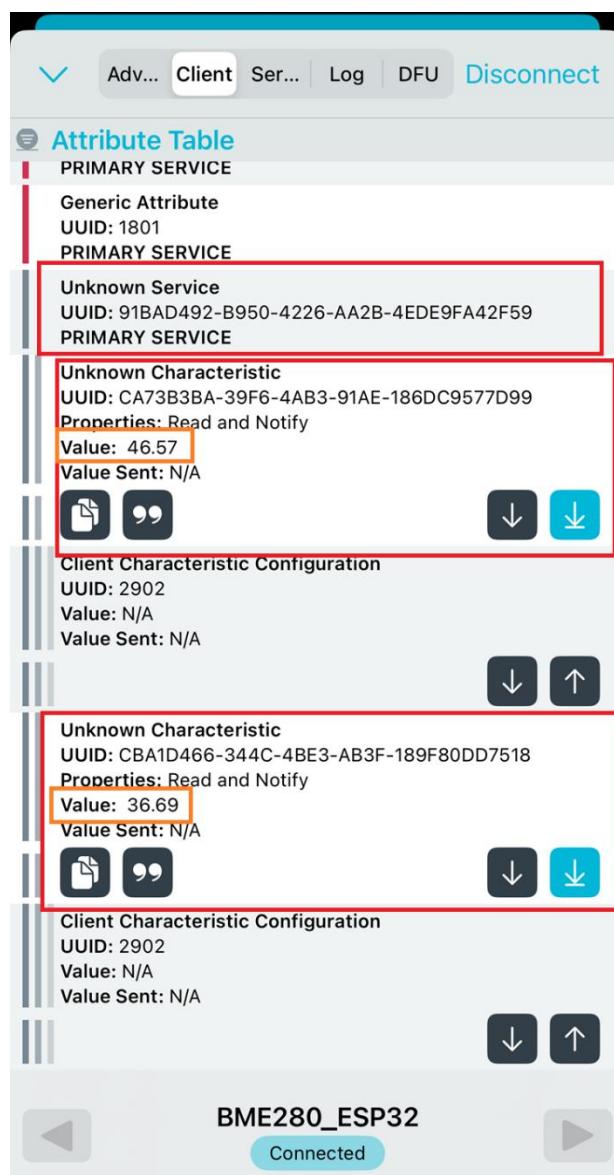
Then, you can test if the BLE server is working as expected by using the *nRF Connect for mobile* app as you did in the previous unit.

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).

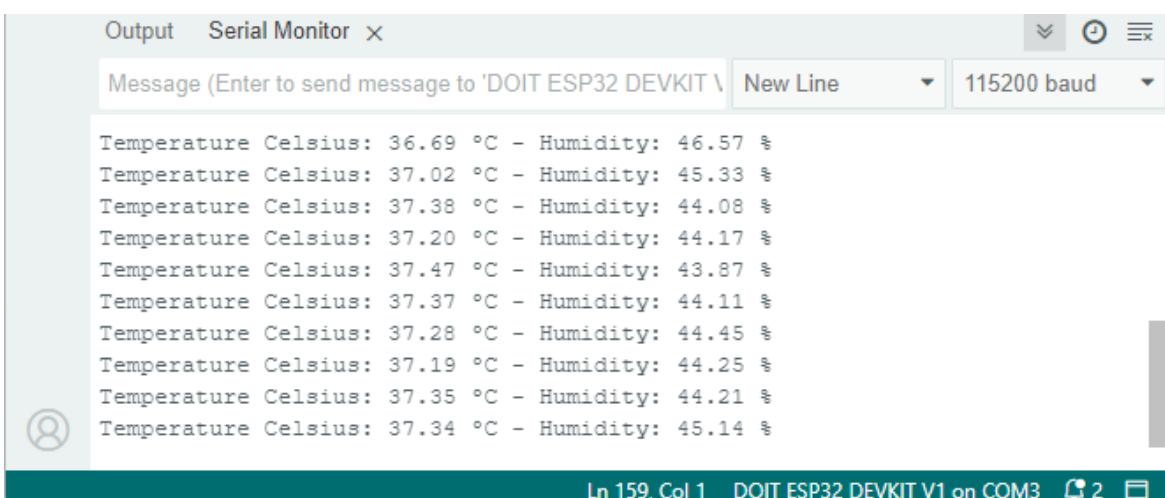


Connect to your **BME280_ESP32** device and then, select the client tab (the interface might be slightly different). You can check that it advertises the service with the UUID we defined in the code, as well as the temperature and humidity

characteristics. Notice that those characteristics have the Notify property. You may need to change the data parser to be able to read the characteristic values.



At the same time, you should also get new temperature and humidity values on the Serial Monitor upon connection with the smartphone.



Your ESP32 BLE Server is ready!

Go to the next section to create an ESP32 client that connects to this server to read the temperature and humidity characteristics and display them on an OLED display.

7.4 - ESP32 BLE Server and Client (Part 2/2)

With the ESP32 BLE server ready from the previous Unit, let's create the ESP32 BLE client that will establish a connection and display the readings on an OLED display. If you don't have the ESP32 BLE server ready yet, go back to the previous Unit.

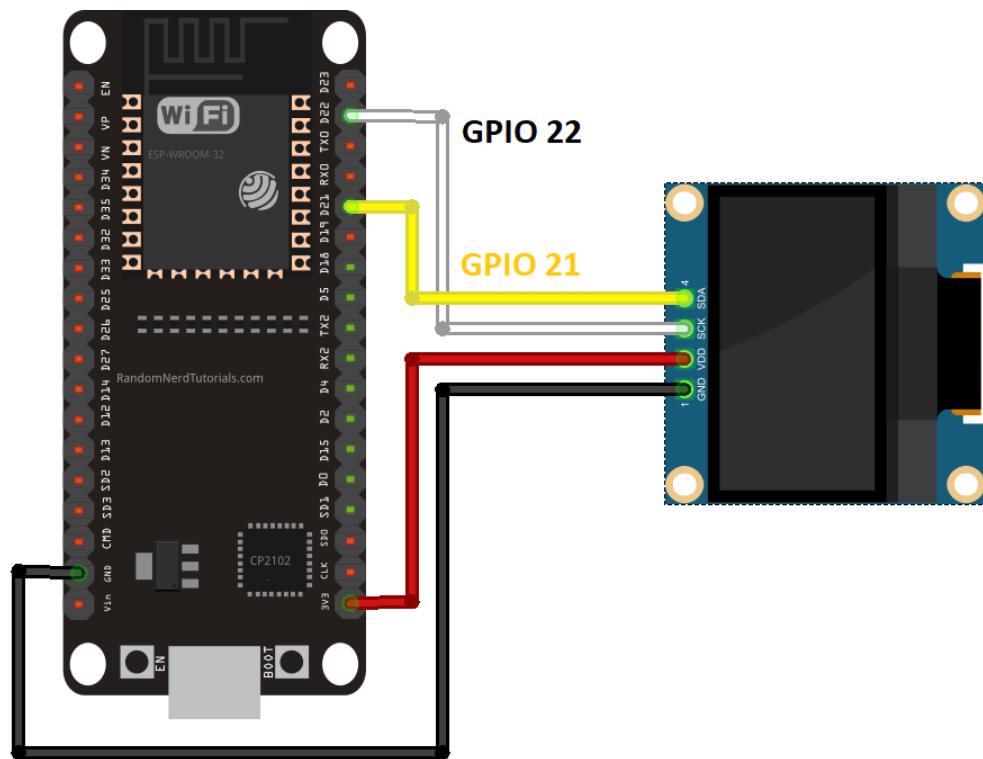
Wiring the Circuit

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 OLED requires 3.3V, the SCL connects to GPIO 22, and SDA to GPIO 21.

Here's a list of parts you need to complete this circuit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [OLED display](#)
- [Jumper wires](#)
- [Breadboard](#)



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.

Client Sketch

Copy the BLE client Sketch to your Arduino IDE. You can find the code on the link below.

- [Click here to download the code.](#)

```
#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 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-4ede9fa42f59");

// BLE Characteristics
#ifndef temperatureCelsius
// Temperature Celsius Characteristic
static BLEUUID temperatureCharacteristicUUID("cba1d466-344c-4be3-ab3f-189f80dd7518");
#else
// Temperature Fahrenheit Characteristic
static BLEUUID temperatureCharacteristicUUID("f78ebbff-c8b7-4107-93de-889a6a06d408");
#endif

// Humidity Characteristic
static BLEUUID humidityCharacteristicUUID("ca73b3ba-39f6-4ab3-91ae-186dc9577d99");

// Flags stating if should begin connecting and if the connection is up
static boolean doConnect = false;
static boolean connected = false;

// Address of the peripheral device. Address will be found during scanning...
```

```

static BLEAddress *pServerAddress;

// Characteristics that we want to read
static BLERemoteCharacteristic* temperatureCharacteristic;
static BLERemoteCharacteristic* humidityCharacteristic;

// Activate notify
const uint8_t notificationOn[] = {0x1, 0x0};
const uint8_t notificationOff[] = {0x0, 0x0};

#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels

// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

// Variables to store temperature and humidity
char temperatureChar[8];
char humidityChar[8];

// Flags to check whether new temperature and humidity readings are available
boolean newTemperature = false;
boolean newHumidity = false;

// Connect to the BLE Server that has the name, Service, and Characteristics
bool connectToServer(BLEAddress pAddress) {
    BLEClient* pClient = BLEDevice::createClient();

    // Connect to the remote BLE Server.
    pClient->connect(pAddress);
    Serial.println(" - Connected to server");

    // Obtain a reference to the service we are after in the remote BLE server.
    BLERemoteService* pRemoteService = pClient->getService(bmeServiceUUID);
    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 the remote BLE server
    temperatureCharacteristic = pRemoteService->getCharacteristic(temperatureCharacteristicUUID);
    humidityCharacteristic = pRemoteService->getCharacteristic(humidityCharacteristicUUID);

    if (temperatureCharacteristic == nullptr || humidityCharacteristic == nullptr) {
        Serial.print("Failed to find our characteristic UUID");
        return false;
    }
    Serial.println(" - Found our characteristics");

    // Assign callback functions for the Characteristics
    temperatureCharacteristic->registerForNotify(temperatureNotifyCallback);
    humidityCharacteristic->registerForNotify(humidityNotifyCallback);
    return true;
}

// Callback function that gets called, when another device's advertisement has been received
class MyAdvertisedDeviceCallbacks : public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice) {
        // Check if the name of the advertiser matches
        if (advertisedDevice.getName() == bleServerName) {

```

```

        // Scan can be stopped, we found what we are looking for
        advertisedDevice.getScan()->stop();
        // Address of advertiser is the one we need
        pServerAddress = new BLEAddress(advertisedDevice.getAddress());
        doConnect = true; // Set indicator, stating that we are ready to connect
        Serial.println("Device found. Connecting!");
    }
}
};

// When the BLE Server sends a new temperature reading with the notify property
static void temperatureNotifyCallback(BLERemoteCharacteristic*
                                      pBLERemoteCharacteristic,
                                      uint8_t* pData, size_t length, bool isNotify) {
    // Copy the received data to the temperatureChar array
    memcpy(temperatureChar, pData, length);
    temperatureChar[length] = '\0'; // Ensure null termination
    newTemperature = true;
}

// When the BLE Server sends a new humidity reading with the notify property
static void humidityNotifyCallback(BLERemoteCharacteristic*
                                      pBLERemoteCharacteristic,
                                      uint8_t* pData, size_t length, bool isNotify) {
    // Copy the received data to the humidityChar array
    memcpy(humidityChar, pData, length);
    humidityChar[length] = '\0'; // Ensure null termination
    newHumidity = true;
}

// Function that prints the latest sensor 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.setTextSize(1);
    display.cp437(true);
    display.write(167); // Display the degree symbol
    display.setTextSize(2);
    Serial.print("Temperature: ");
    Serial.print(temperatureChar);

#ifndef 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("%");
    }

void setup() {
    // OLED display setup
    // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
    if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for 128x32
        Serial.println(F("SSD1306 allocation failed"));
        for (;;) ; // Don't proceed, loop forever
    }
    display.clearDisplay();
    display.setTextSize(2);
    display.setTextColor(WHITE, 0);
    display.setCursor(0, 25);
    display.print("BLE Client");
    display.display();

    // Start serial communication
    Serial.begin(115200);
    Serial.println("Starting Arduino BLE Client application...");

    // Init BLE device
    BLEDevice::init("");

    // Retrieve a Scanner and set the callback we want to use to be informed when we
    // have detected a new device. Specify that we want active scanning and start the
    // scan to run for 30 seconds.
    BLEScan* pBLEScan = BLEDevice::getScan();
    pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
    pBLEScan->setActiveScan(true);
    pBLEScan->start(30);
}

void loop() {
    // If the flag "doConnect" is true then we have scanned for and found the desired
    // BLE Server with which we wish to connect. Now we connect to it. Once we are
    // connected we set the connected flag to be true.
    if (doConnect == true) {
        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)0x2902))
                ->writeValue((uint8_t*)notificationOn, 2, true);
            humidityCharacteristic->getDescriptor(BLEUUID((uint16_t)0x2902))
                ->writeValue((uint8_t*)notificationOn, 2, true);
            connected = true;
        } else {
            Serial.println("We have failed to connect to the server;
                           Restart your device to scan for nearby BLE server again.");
        }
        doConnect = false;
    }
    // If new temperature readings are available, print them on the OLED
    if (newTemperature && newHumidity) {

```

```
    newTemperature = false;
    newHumidity = false;
    printReadings();
}
delay(1000); // Delay a second between loops.
}
```

How Does the Code Work?

Let's take a look at how this code works to connect to the ESP32 BLE server and read its characteristics' values.

Importing libraries

You start by importing the required libraries for BLE and the OLED display:

```
#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-4ede9fa42f59");

// BLE Characteristics
#ifndef temperatureCelsius
// Temperature Celsius Characteristic
static BLEUUID temperatureCharacteristicUUID("cba1d466-344c-4be3-ab3f-189f80dd7518");
#else
// Temperature Fahrenheit Characteristic
static BLEUUID temperatureCharacteristicUUID("f78ebbff-c8b7-4107-93de-889a6a06d408");
```

```
#endif

// Humidity Characteristic
static BLEUUID humidityCharacteristicUUID("ca73b3ba-39f6-4ab3-91ae-186dc9577d99");
```

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 is up
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 scanning...
static BLEAddress *pServerAddress;
```

Set the characteristics we want to read (temperature and humidity).

```
// Characteristics 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.

```
// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);
```

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[8];
char humidityChar[8];
```

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 are available
boolean newTemperature = false;
boolean newHumidity = false;
```

printReadings()

We created a function called `printReadings()` that displays the temperature and humidity readings on the OLED display.

```
// Function that prints the latest sensor 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.setTextSize(1);
    display.cp437(true);
    display.write(167); // Display the degree symbol
    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("%");
}
```

setup()

In the `setup()`, start the OLED display and print a message in the first line saying "BLE Client".

```
// OLED display setup
// SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for 128x32
    Serial.println(F("SSD1306 allocation failed"));
    for (;;) ; // Don't proceed, loop forever
}
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 informed when
// we have detected a new device. Specify that we want active scanning and start
// the scan to run for 30 seconds.
BLEScan* pBLEScan = BLEDevice::getScan();
pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
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 advertisement has been received
class MyAdvertisedDeviceCallbacks : public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice) {
        // Check if the name of the advertiser matches
```

```

    if (advertisedDevice.getName() == bleServerName) {
        // Scan can be stopped, we found what we are looking for
        advertisedDevice.getScan()->stop();
        // Address of advertiser is the one we need
        pServerAddress = new BLEAddress(advertisedDevice.getAddress());
        doConnect = true; // Set indicator, stating that we are ready to connect
        Serial.println("Device found. Connecting!");
    }
}
};

```

Connect to the server

In the `loop()`, if the `doConnect` variable is `true`, it tries to connect to the BLE server.

```

void loop() {
    // If the flag "doConnect" is true then we have scanned for and found the desired
    // BLE Server with which we wish to connect. Now we connect to it. Once we are
    // connected we set the connected flag to be true.
    if (doConnect == true) {
        if (connectToServer(*pServerAddress)) {

```

The `connectToServer()` function handles the connection between the client and the server.

```

// Connect to the BLE Server that has the name, Service, and Characteristics
bool connectToServer(BLEAddress pAddress) {
    BLEClient* pClient = BLEDevice::createClient();

    // Connect to the remote BLE Server.
    pClient->connect(pAddress);
    Serial.println(" - Connected to server");

    // Obtain a reference to the service we are after in the remote BLE server.
    BLERemoteService* pRemoteService = pClient->getService(bmeServiceUUID);
    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 the remote BLE server
    temperatureCharacteristic = pRemoteService->
                                getCharacteristic(temperatureCharacteristicUUID);
    humidityCharacteristic = pRemoteService->
                                getCharacteristic(humidityCharacteristicUUID);

    if (temperatureCharacteristic == nullptr || humidityCharacteristic == nullptr) {
        Serial.print("Failed to find our characteristic UUID");
        return false;
    }
    Serial.println(" - Found our characteristics");

    // Assign callback functions for the Characteristics
    temperatureCharacteristic->registerForNotify(temperatureNotifyCallback);
    humidityCharacteristic->registerForNotify(humidityNotifyCallback);

```

```
    return true;
}
```

It also assigns a callback function responsible for handling what happens when a new value is received.

```
// Assign callback functions for the Characteristics
temperatureCharacteristic->registerForNotify(temperatureNotifyCallback);
humidityCharacteristic->registerForNotify(humidityNotifyCallback);
```

After the BLE client is connected to the server, you need to activate 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)0x2902))
        ->writeValue((uint8_t*)notificationOn, 2, true);
    humidityCharacteristic->getDescriptor(BLEUUID((uint16_t)0x2902))
        ->writeValue((uint8_t*)notificationOn, 2, true);
    connected = true;
```

Notify new values

When the client receives a new notify value, it will call these two functions: `temperatureNotifyCallback()` and `humidityNotifyCallback()` which are responsible for retrieving the new value, updating the OLED with the new readings, and printing them on the Serial Monitor.

```
// When the BLE Server sends a new temperature reading with the notify property
static void temperatureNotifyCallback(BLERemoteCharacteristic*
    pBLERemoteCharacteristic,
    uint8_t* pData, size_t length, bool isNotify) {
    // Copy the received data to the temperatureChar array
    memcpy(temperatureChar, pData, length);
    temperatureChar[length] = '\0'; // Ensure null termination
    newTemperature = true;
}
// When the BLE Server sends a new humidity reading with the notify property
static void humidityNotifyCallback(BLERemoteCharacteristic*
    pBLERemoteCharacteristic,
    uint8_t* pData, size_t length, bool isNotify) {
    // Copy the received data to the humidityChar array
    memcpy(humidityChar, pData, length);
    humidityChar[length] = '\0'; // Ensure null termination
    newHumidity = true;
}
```

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.

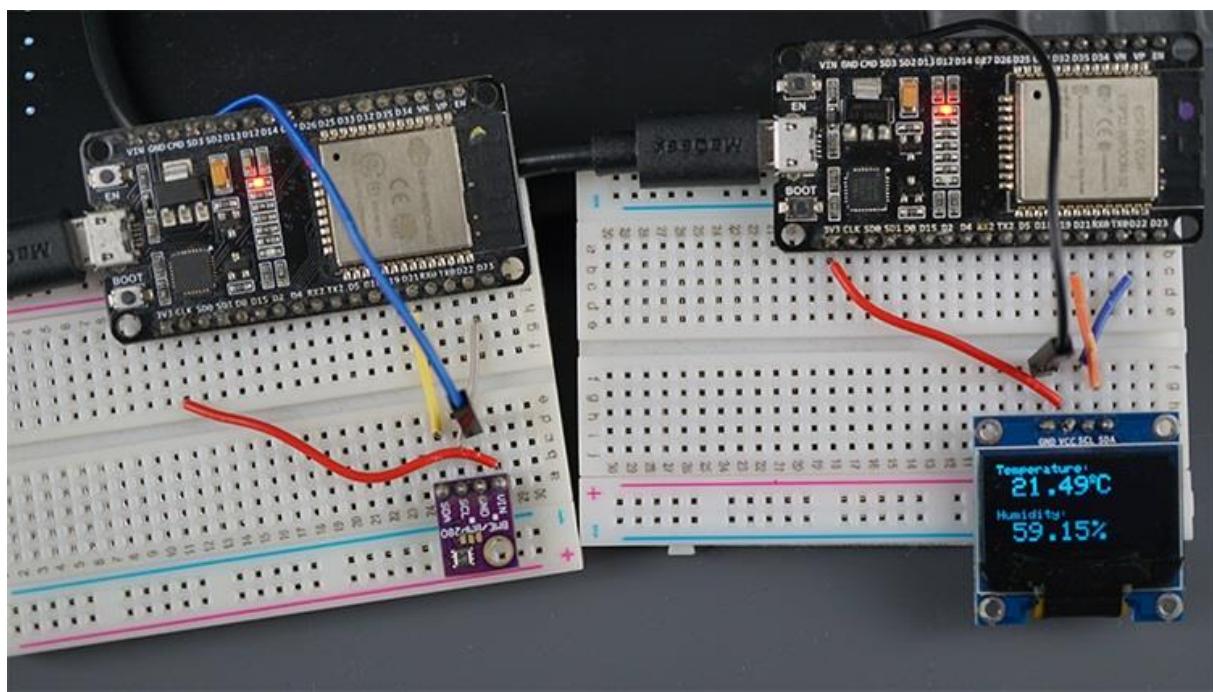
Display new temperature and humidity readings

In the `loop()`, there is an if statement that checks if new readings are available. If there are new readings, we set the `newTemperature` and `newHumidity` variables to `false`, so that we can 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 them on the OLED
if (newTemperature && newHumidity) {
    newTemperature = false;
    newHumidity = false;
    printReadings();
}
```

Testing the Project

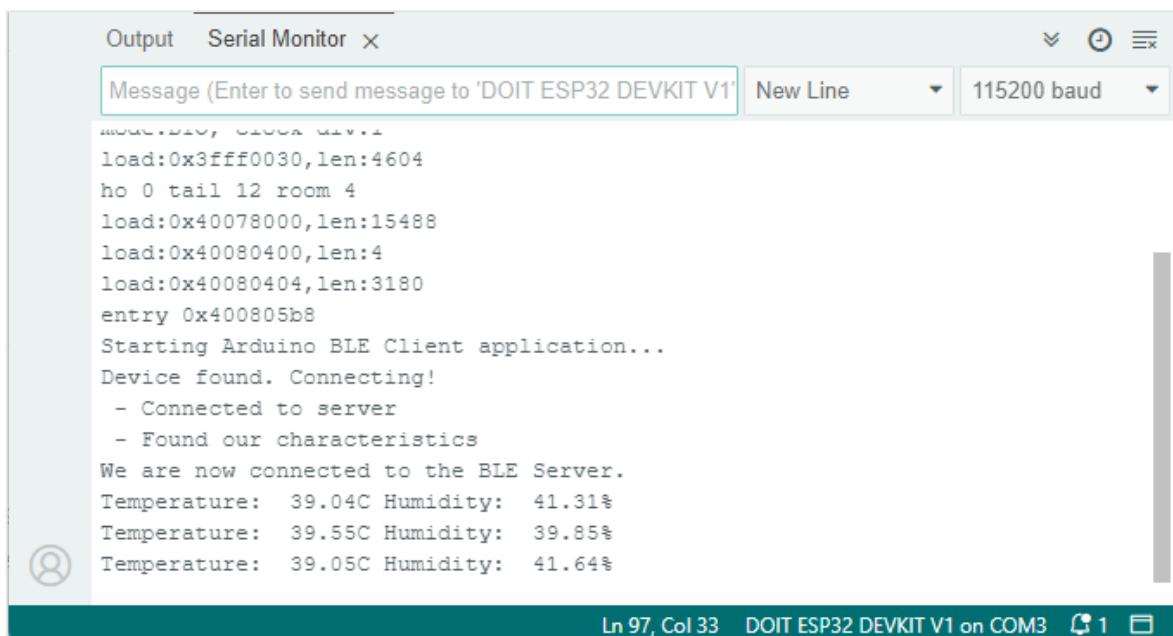
That's it for the code. You can upload it to your ESP32 board. Once the code is uploaded. Power the ESP32 BLE server, then power the ESP32 with the client sketch. The client starts scanning nearby devices, and when it finds the other ESP32, it establishes a Bluetooth connection. Every 30 seconds, it updates the display with the latest readings.



Troubleshooting tips: if you're not receiving any data on the OLED display there are two common problems:

- The BME280 is failing to read the temperature and humidity (open the serial monitor in the BLE server to see if it's printing any results);
- Your ESP32 might be failing to establish a connection with the server. Reboot the ESP32 running BLE client sketch.

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.



The screenshot shows the Arduino Serial Monitor window. The title bar says "Output" and "Serial Monitor". The main area displays the following text:

```
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1')
New Line 115200 baud
load:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Starting Arduino BLE Client application...
Device found. Connecting!
- Connected to server
- Found our characteristics
We are now connected to the BLE Server.
Temperature: 39.04C Humidity: 41.31%
Temperature: 39.55C Humidity: 39.85%
Temperature: 39.05C Humidity: 41.64%
```

At the bottom, status indicators show "Ln 97, Col 33" and "DOIT ESP32 DEVKIT V1 on COM3".

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.

7.5 - Bluetooth Classic

The ESP32 supports Wi-Fi, Bluetooth Low Energy, and Bluetooth Classic. This Unit will show you how to use Bluetooth Classic with the ESP32 and Arduino IDE to exchange data between devices. We'll control an ESP32 GPIO, and monitor sensor readings with an Android smartphone using Bluetooth Classic.



Note: this project is only compatible with [Android smartphones](#).

Bluetooth Classic is significantly easier to use compared to Bluetooth Low Energy, especially if you're already familiar with programming an Arduino board with a Bluetooth module like the HC-05. It operates using the standard serial protocol and functions, making the process more straightforward than using the GATT profile in Bluetooth Low Energy.

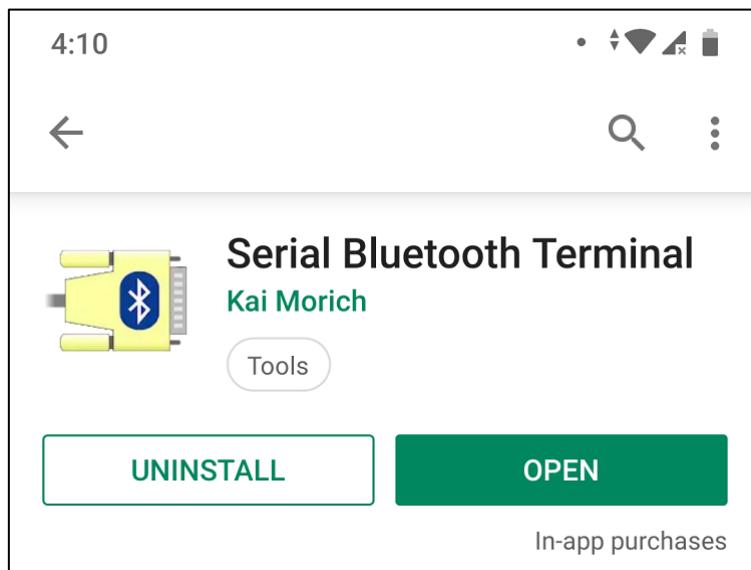
We'll start by analyzing and testing an example that comes with the Arduino IDE. Then, we'll build a simple project to exchange data between the ESP32 and your Android smartphone.

Bluetooth Terminal Application

To proceed with this tutorial, you need a Bluetooth Terminal application installed on your smartphone.

Note: this project is only compatible with Android smartphones.

We recommend using the Android app “[Serial Bluetooth Terminal](#)” available in the Play Store.



Serial to Serial Bluetooth

After installing the application, open the following example in Arduino IDE: **File > Examples > BluetoothSerial > SerialtoSerialBT**, or click on the link below to access the code.

- [Click here to download the code.](#)

```
// This example code is in the Public Domain (or CC0 licensed, at your option.)  
// By Evandro Copercini - 2018  
// This example creates a bridge between Serial and Classical Bluetooth (SPP)  
// and also demonstrate that SerialBT have the same functionalities of a normal Serial  
// Note: Pairing is authenticated automatically by this device  
  
#include "BluetoothSerial.h"  
  
String device_name = "ESP32-BT-Slave";  
  
// Check if Bluetooth is available  
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)  
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it  
#endif
```

```

// Check Serial Port Profile
#if !defined(CONFIG_BT_SPP_ENABLED)
#error Serial Port Profile for Bluetooth is not available or not enabled. It is
only available for the ESP32 chip.
#endif

BluetoothSerial SerialBT;

void setup() {
    Serial.begin(115200);
    SerialBT.begin(device_name); // Bluetooth device name
    // Uncomment this to delete paired devices; Must be called after begin
    //SerialBT.deleteAllBondedDevices();
    Serial.printf("The device with name \"%s\" is started.\n"
                  "Now you can pair it with Bluetooth!\\n", device_name.c_str());
}

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

```

How Does the Code Work?

This code establishes a two-way serial Bluetooth communication between two devices. The code starts by including the `BluetoothSerial` library.

```
#include "BluetoothSerial.h"
```

Set the name for the ESP32 Bluetooth Device.

```
String device_name = "ESP32-BT-Slave";
```

The next three lines check if Bluetooth and if the Serial Port Profile are properly enabled.

```

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

// Check Serial Port Profile
#if !defined(CONFIG_BT_SPP_ENABLED)
#error Serial Port Profile for Bluetooth is not available or not enabled. It is
only available for the ESP32 chip.
#endif

```

Then, create an instance of `BluetoothSerial` called `SerialBT`. This creates a Bluetooth Serial Port Profile to exchange data via Bluetooth using the ESP32.

```
BluetoothSerial SerialBT;
```

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.

```
SerialBT.begin(device_name); // Bluetooth device name
```

In the `loop()`, we 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. On the other hand, `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 how this sketch works during the demonstration.

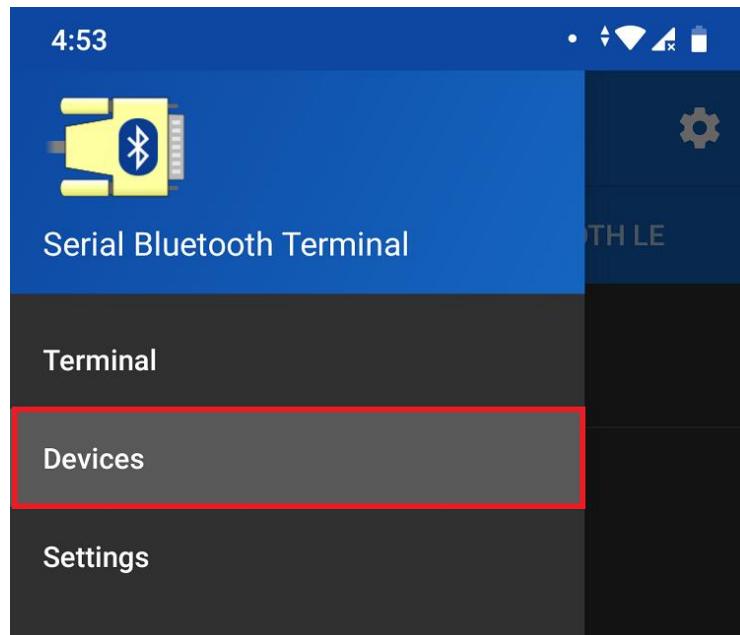
Testing the Code

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

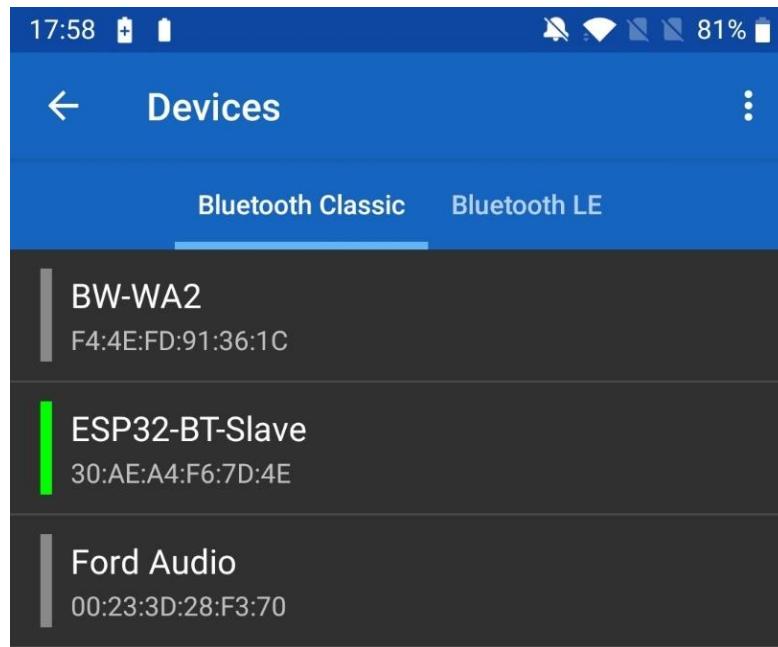
After uploading the code, open the Serial Monitor at a baud rate of 115200. Press the ESP32 Enable button.

Go to your smartphone and open the Serial **Bluetooth Terminal** app. Make sure you've enabled your smartphone's Bluetooth.

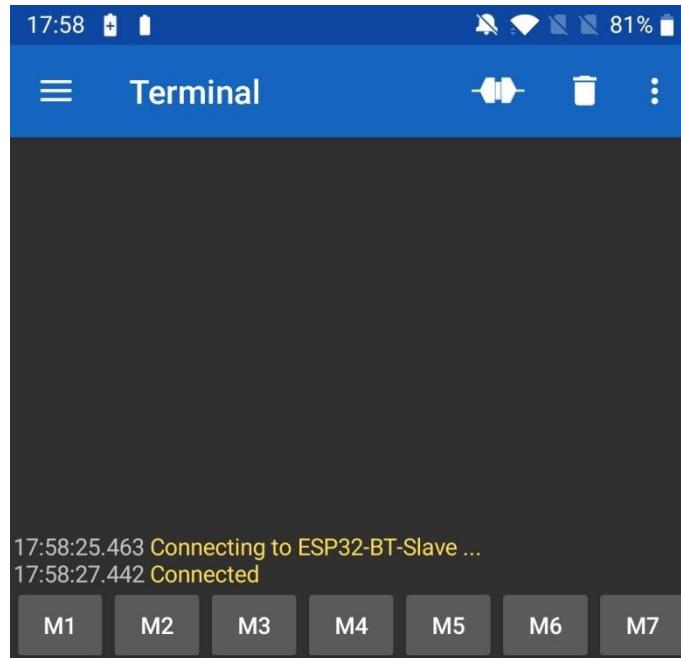
To connect to the ESP32 for the first time, you need to pair a new device. Go to **Devices**.



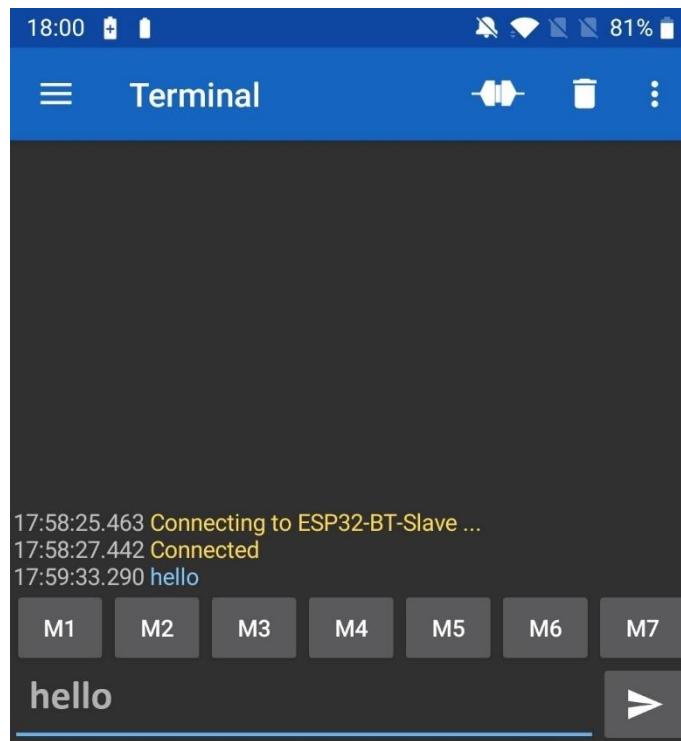
Click the **settings** icon, and select **Pair new device**. You should get a list of the available Bluetooth devices, including the **ESP32-BT-Slave device**. Pair with that device.



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.

The screenshot shows the Arduino IDE Serial Monitor window. The top bar includes tabs for 'Output' and 'Serial Monitor' (which is selected), and settings for 'New Line' and '115200 baud'. The main text area displays a configuration dump for an ESP32 DEVKIT V1 followed by a message exchange. The user has typed 'hello' into the message input field, which is highlighted with a red box. The response 'Hi. How are you?' is visible in the text area below. The status bar at the bottom indicates 'Ln 1, Col 1' and 'DOIT ESP32 DEVKIT V1 on COM3'.

```
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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
hello
```

You can also exchange data between your Serial Monitor and your smartphone. Type something in the Serial Monitor top bar then, press the Enter key to send that message.

The screenshot shows the Arduino IDE Serial Monitor window. The top bar includes tabs for 'Output' and 'Serial Monitor' (selected), and settings for 'New Line' and '115200 baud'. The main text area displays a configuration dump for an ESP32 DEVKIT V1 followed by a message exchange. The user has typed 'Hi. How are you?' into the message input field, which is highlighted with a red box. The response 'Hello' is visible in the text area below. The status bar at the bottom indicates 'Ln 1, Col 1' and 'DOIT ESP32 DEVKIT V1 on COM3'.

```
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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
hello
```

You should instantly receive the message in the Serial Bluetooth Terminal App.

The screenshot shows the Serial Bluetooth Terminal App interface. The main text area displays a log of messages. It starts with a connection sequence: '17:58:25.463 Connecting to ESP32-BT-Slave ...', '17:58:27.442 Connected', '17:59:33.290 hello', and '18:00:25.448 Hi. How are you?'. Below the text area is a control panel with seven buttons labeled M1 through M7, and a final button with a right-pointing arrow.

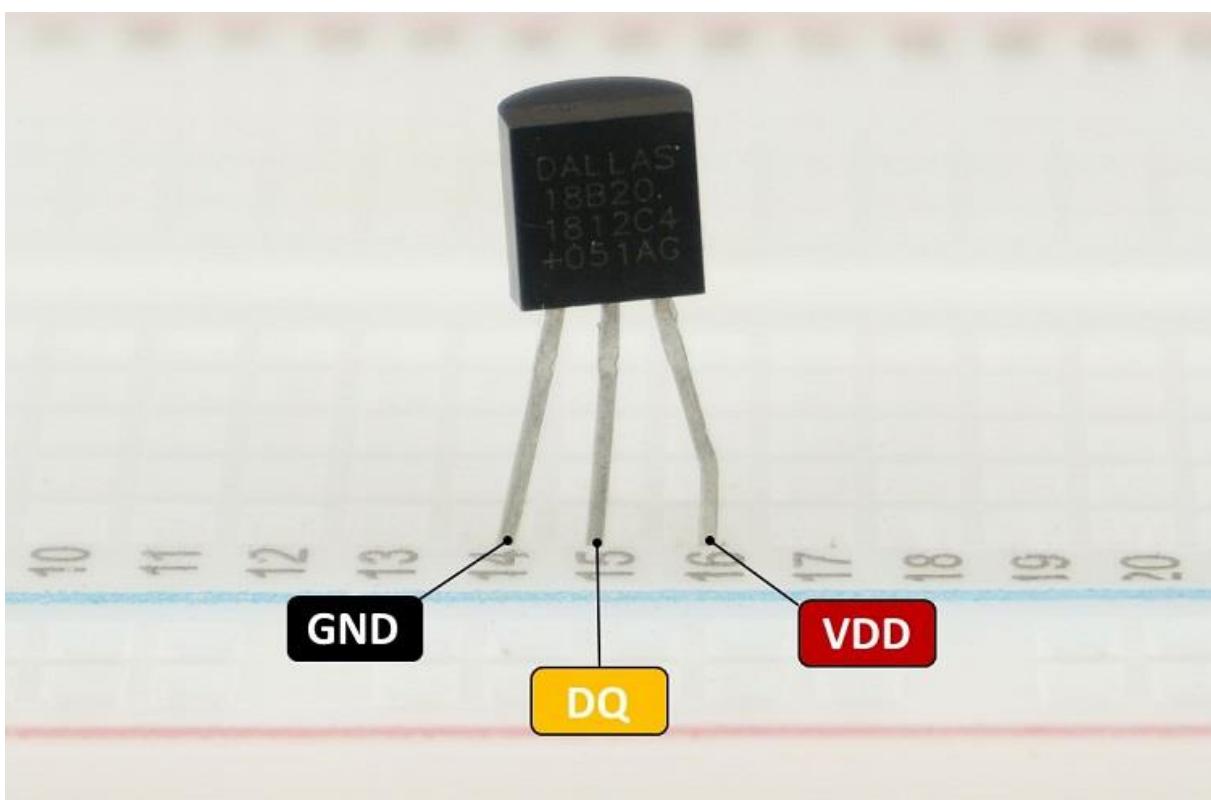
```
17:58:25.463 Connecting to ESP32-BT-Slave ...
17:58:27.442 Connected
17:59:33.290 hello
18:00:25.448 Hi. How are you?
```

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, or send data to your smartphone like sensor readings.

Project Overview

The project we'll build sends temperature readings every 10 seconds to your smartphone. We'll be using the DS18B20 temperature sensor shown in the following figure.



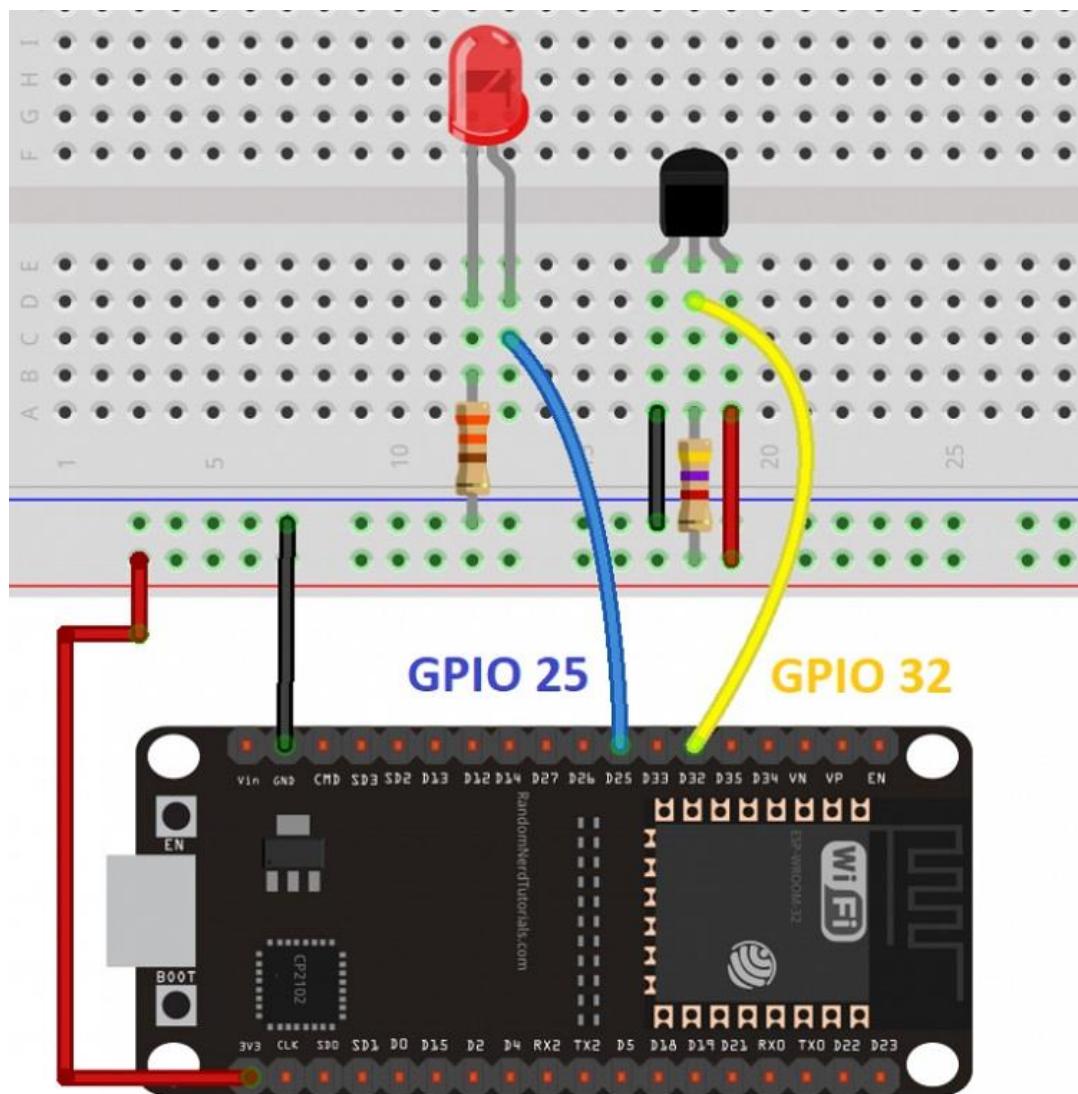
Through the Android app, we'll send messages to control an ESP32 output (an LED). When the ESP32 receives the **led_on** message, we'll turn the LED on, and when it receives the **led_off** message, we'll turn the LED off.

Wiring the Circuit

Before proceeding with this project, assemble the circuit. Connect an LED to GPIO 25, and connect the DS18B20 data pin to GPIO 32. Use the provided schematic diagram as a reference.

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [330 Ohm resistor](#)
- [DS18B20 temperature sensor](#)
- [4.7k Ohm resistor](#)
- [Jumper wires](#)
- [Breadboard](#)



Installing the DS18B20 Libraries

To work with the DS18B20 temperature sensor, you need to install the [One Wire library by Paul Stoffregen](#) and the [Dallas Temperature library](#).

In your Arduino IDE, go to **Sketch > Include Library > Manage Libraries**. Search for the libraries and install them.

- One Wire library by Paul Stoffregen
- Dallas Temperature library

Code

After assembling the circuit and installing the necessary libraries, copy the code provided below to the Arduino IDE.

- [Click here to download the code.](#)

```
// 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_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

// Check Serial Port Profile
#if !defined(CONFIG_BT_SPP_ENABLED)
#error Serial Port Profile for Bluetooth is not available or not enabled. It is
only available for the ESP32 chip.
#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 devices
OneWire oneWire(oneWireBus);
// Pass our oneWire reference to Dallas Temperature sensor
DallasTemperature sensors(&oneWire);

// Handle received and sent messages
String message = "";
char incomingChar;
String temperatureString = "";

// Timer: auxiliar variables
unsigned long previousMillis = 0; // Stores last time temperature was published
const long interval = 10000; // interval at which to publish sensor readings

void setup() {
  pinMode(ledPin, OUTPUT);
  Serial.begin(115200);
```

```

// Bluetooth device name
SerialBT.begin("ESP32");
Serial.println("The device started, now you can pair it with bluetooth!");
}

void loop() {
    unsigned long currentMillis = millis();
    // Send temperature readings
    if (currentMillis - previousMillis >= interval){
        previousMillis = currentMillis;
        sensors.requestTemperatures();
        temperatureString = String(sensors.getTempCByIndex(0))
                            + "C " + String(sensors.getTempFByIndex(0)) + "F";
        SerialBT.println(temperatureString);
    }
    // Read received messages (LED control command)
    if (SerialBT.available()){
        char incomingChar = SerialBT.read();
        if (incomingChar != '\n'){
            message += String(incomingChar);
        }
        else{
            message = "";
        }
        Serial.write(incomingChar);
    }
    // Check received message and control output accordingly
    if (message == "led_on"){
        digitalWrite(ledPin, HIGH);
    }
    else if (message == "led_off"){
        digitalWrite(ledPin, LOW);
    }
    delay(20);
}

```

How Does the Code Work?

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, it's GPIO 25.

```
const int ledPin = 25;
```

Define the DS18B20 sensor pin and create an instance of the `DallasTemperature` object called `sensor` on that GPIO.

```
// GPIO where the DS18B20 is connected to
const int oneWireBus = 32;
// Setup a OneWire instance to communicate with any OneWire devices
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 auxiliary timer variables to send readings every 10 seconds.

```
// Timer: auxiliar variables
unsigned long previousMillis = 0; // Stores last time temperature was published
const long interval = 10000; // interval at which to publish sensor readings
```

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");
```

In the `loop()`, we'll 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();
// Send temperature readings
```

```
if (currentMillis - previousMillis >= interval){
    previousMillis = currentMillis;
    sensors.requestTemperatures();
    temperatureString = String(sensors.getTempCByIndex(0))
        + "C " + String(sensors.getTempFByIndex(0)) + "F";
    SerialBT.println(temperatureString);
}
```

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.

```
// Read received messages (LED control command)
if (SerialBT.available()){
    char incomingChar = SerialBT.read();
    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.

```
else{
    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.

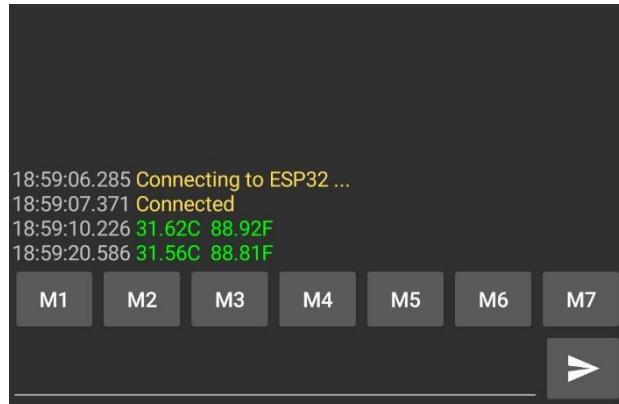
```
else if (message == "led_off"){
    digitalWrite(ledPin, LOW);
}
```

Testing the Project

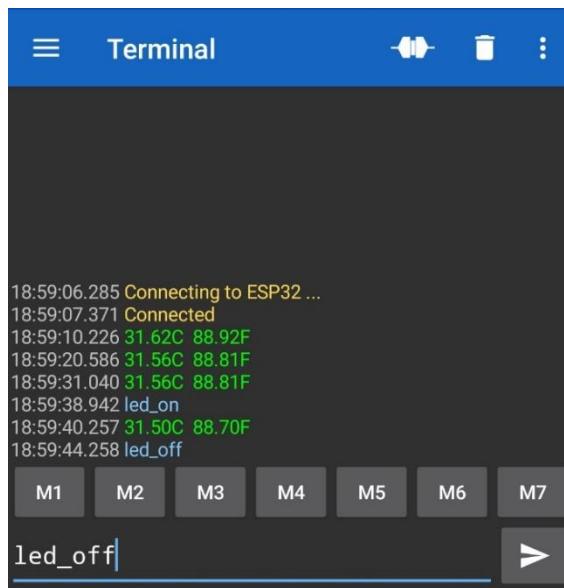
Upload the previous code to your ESP32 board.

Open the Serial Monitor, and press the ESP32 RESET/EN button to restart the ESP32. Go to your Smartphone, open the Bluetooth Terminal app, and connect to the ESP32.

You'll start receiving new temperature readings every 10 seconds.



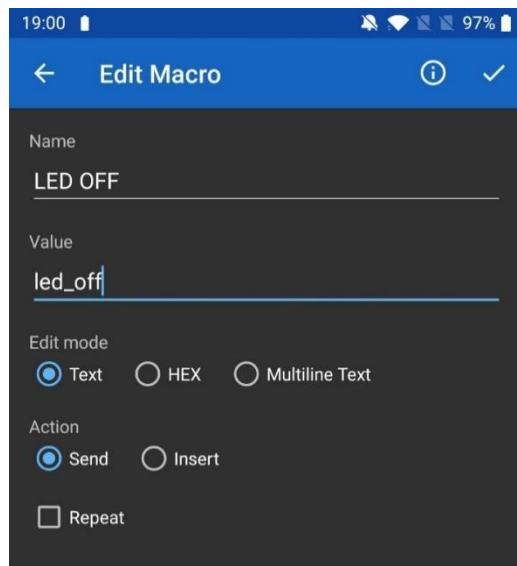
Then, you can write the “**led_on**” and “**led_off**” messages to control the LED.



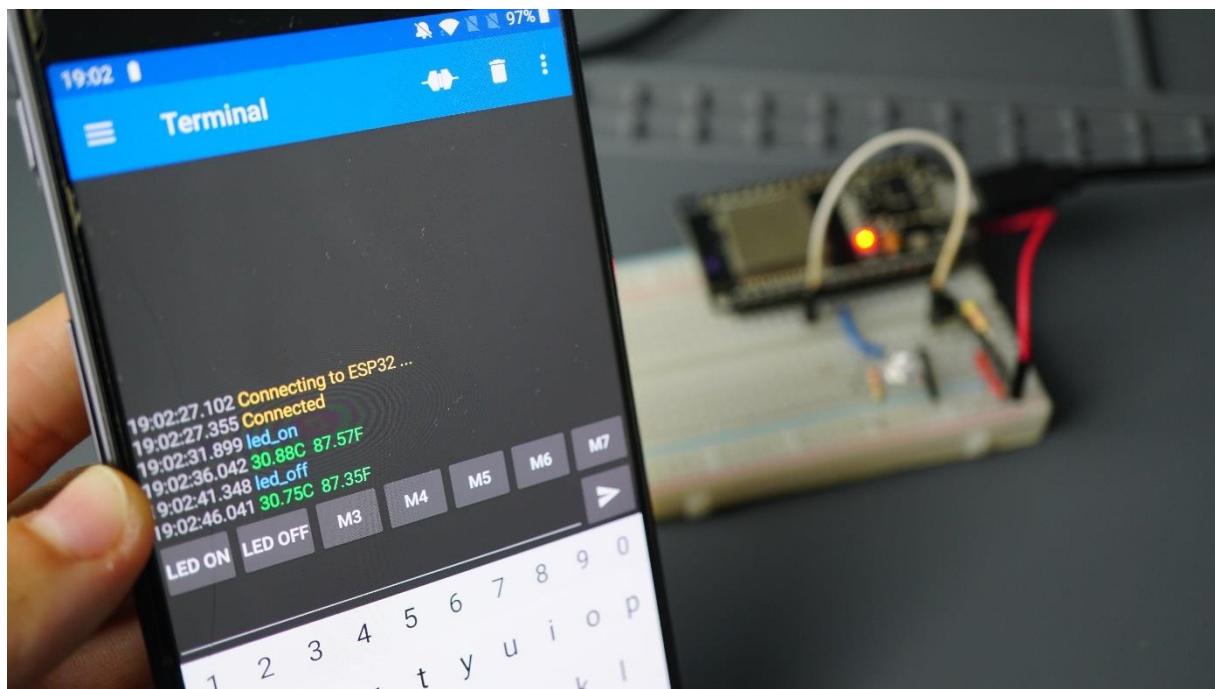
The application has several buttons in which you can save default messages. For example. Hold on to the M1 button. You can edit the button name and set a default message for that button. For example:

- name: LED ON
- message: led_on

Do the same procedure for another button. But, this time to create a button to turn off the LED.



Now, you can control the ESP32 LED using the buttons on the app.



Wrapping Up

In summary, the ESP32 supports BLE and Bluetooth Classic. Using Bluetooth Classic is as simple as using a serial communication and its functions.

MODULE 8

LoRa Technology with ESP32

8.1 - ESP32 with LoRa: Introduction

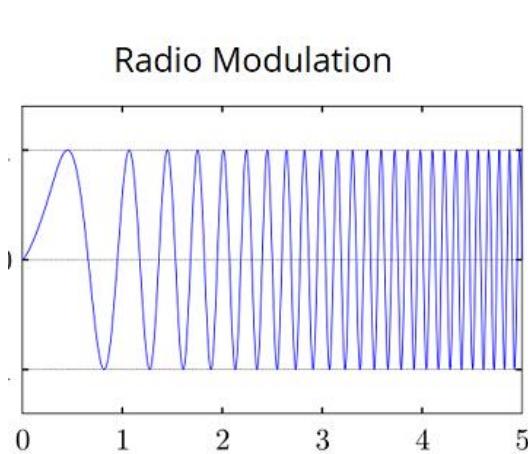
In this Unit, we'll explore the basic principles of LoRa technology, and how it can be used with the ESP32 for IoT projects.



Before getting started, note that LoRa is a vast topic with many features and applications. We'll just cover the basic concepts so that you're able to use LoRa with your ESP32 without having to go deep into a lot of theoretical stuff.

What is LoRa?

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



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.



Long distance communication



Small amounts of data (low bandwidth)



High immunity to interference



Low power consumption

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 here.](#)

LoRa Applications

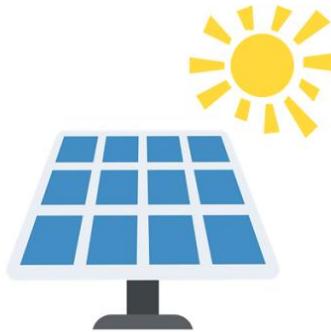
LoRa long-range and low-power features make it perfect for battery-operated sensors and low-power applications in Internet of Things (IoT), smart home, machine-to-machine communication, and much more.

IoT

SMART
HOME

M2M

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



LoRa is Not a Good Option For...

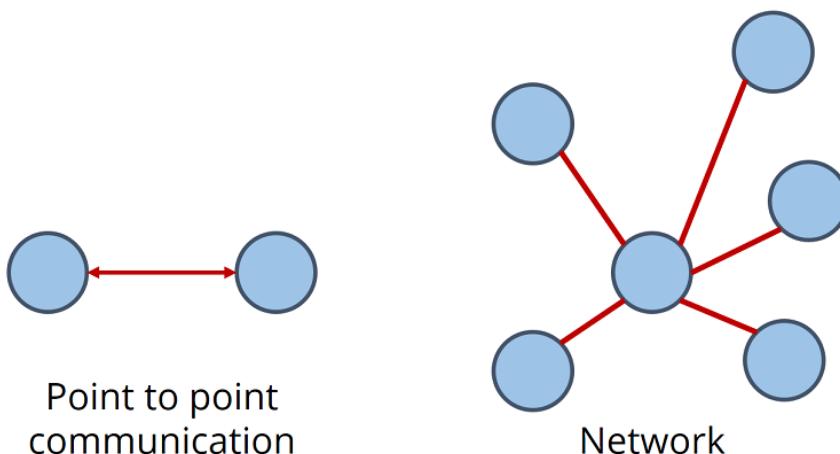


Keep in mind that LoRa is not suitable for projects that:

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

LoRa Topologies

You can use LoRa in:



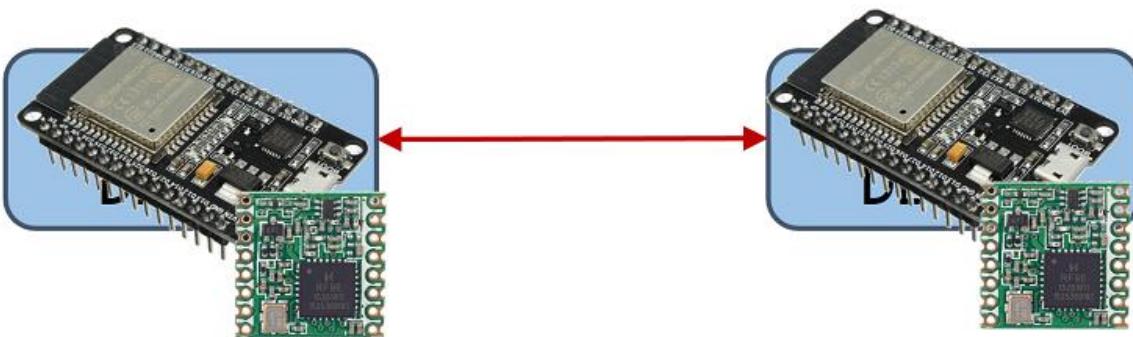
- Point to point communication
- Or build a LoRa network using LoRaWAN

Point to Point Communication

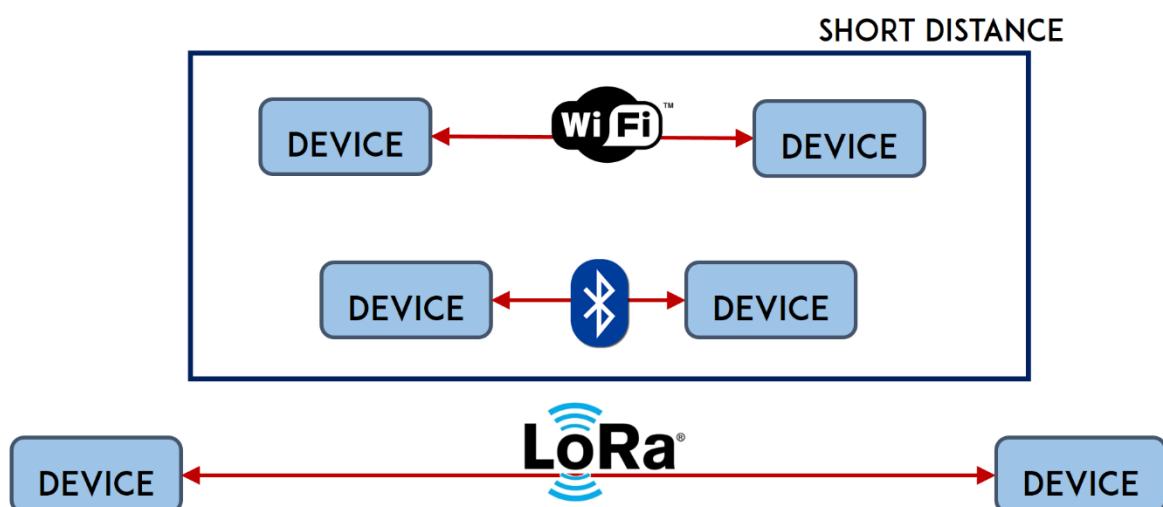
In point-to-point communication, two LoRa-enabled devices talk with each other using RF signals.



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 of distance. Other LoRa solutions can 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](#).

LoRaWan Network Architecture

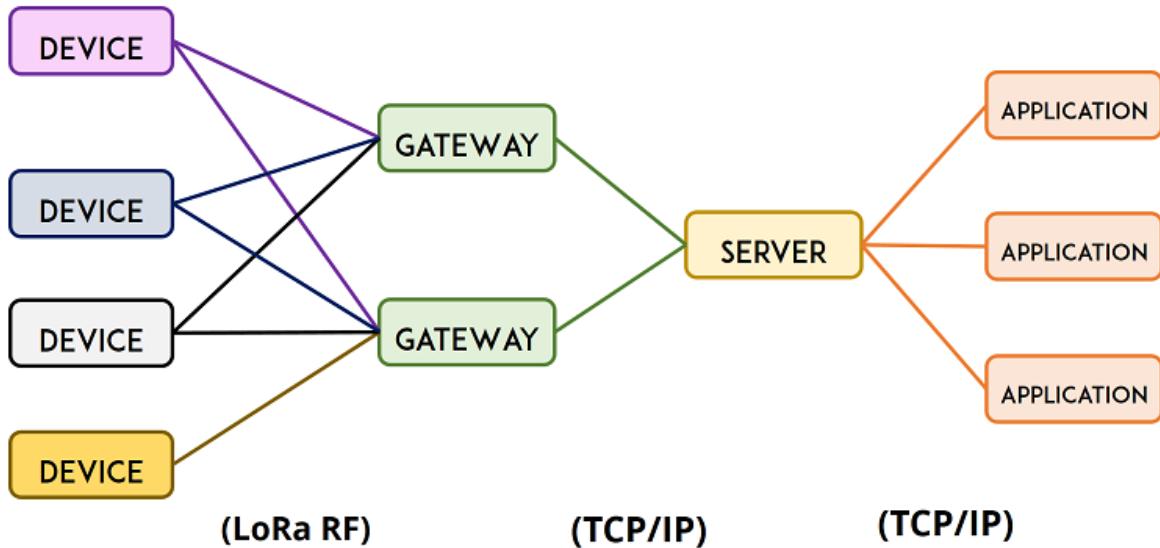
A typical LoRa network consists of four parts:

- Devices;
- Gateways;
- Network server;
- Application.



In the LoRa network, end devices transfer data to gateways. Gateways scan and capture LoRa packets. Devices are not associated with a single gateway. This means that all gateways within the range of a device receive the signal.

Then, the gateways pass the received data to a network server that will handle the packet over TCP/IP. Our application then connects to that network server to receive the data from the end devices.



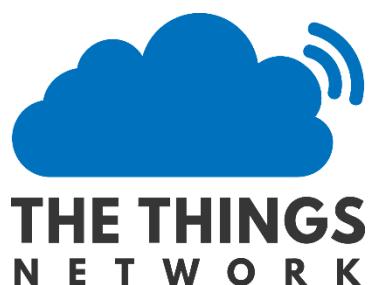
Network Options

Currently, there are two LoRa network options:

- Private network
- Public network

You can build your own private wireless sensor network by setting up your own gateway and your network server. Or you can use a LoRaWan infrastructure offered by a third party, allowing you to deploy your sensors in the field without investing in gateway technology.

Public networks can be managed by a telecom company or by a community of people. For example, you can set up and register your own gateway in The Things Network (TTN).



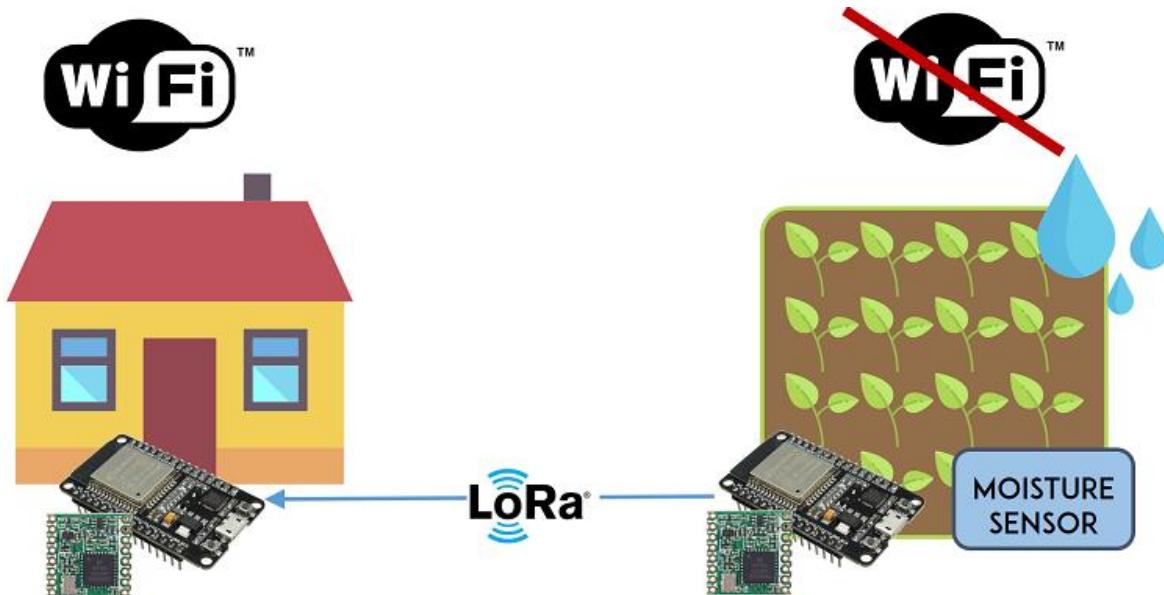
You can allow other people to use your gateway for long range connectivity, or you can use community gateways. If many people set up and share their gateways, we can cover wide areas and allow transmission of messages at longer distances using this protocol.



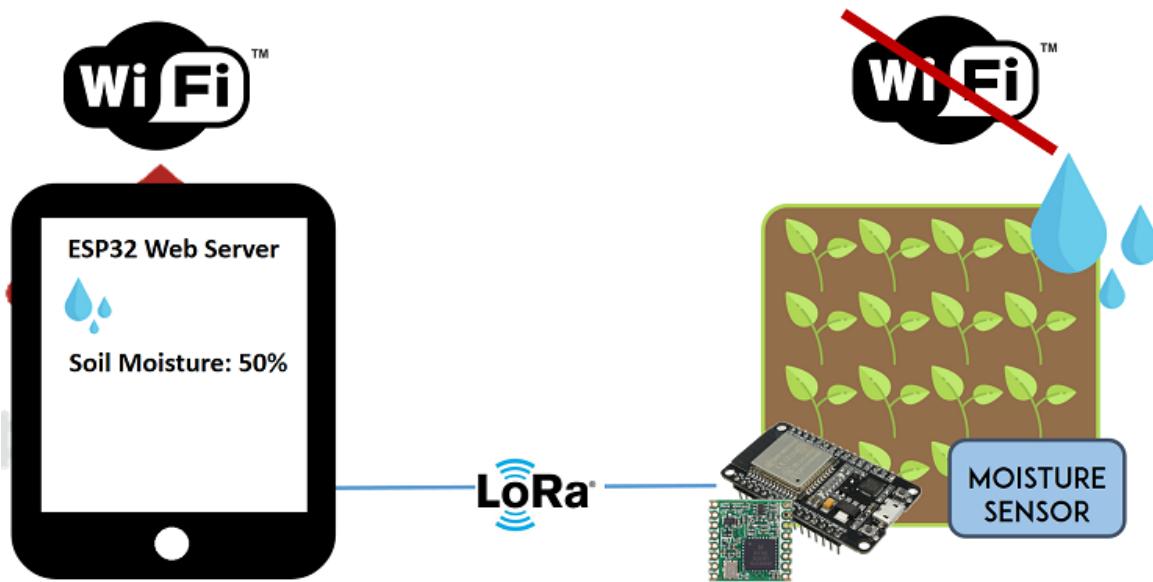
How can LoRa be useful in your home automation projects?

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.



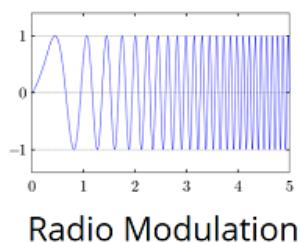
The later ESP32 has access to Wi-Fi, and it can run a web server that displays the moisture readings.



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

Wrapping Up

In summary:



Radio Modulation



Long distance communication



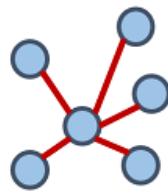
Small amounts of data



Low power consumption



Point to point communication



Network



Useful to monitor sensors without Wi-Fi coverage

- LoRa is a radio modulation technique;
- LoRa allows long-distance communication of small amounts of data and requires low power;

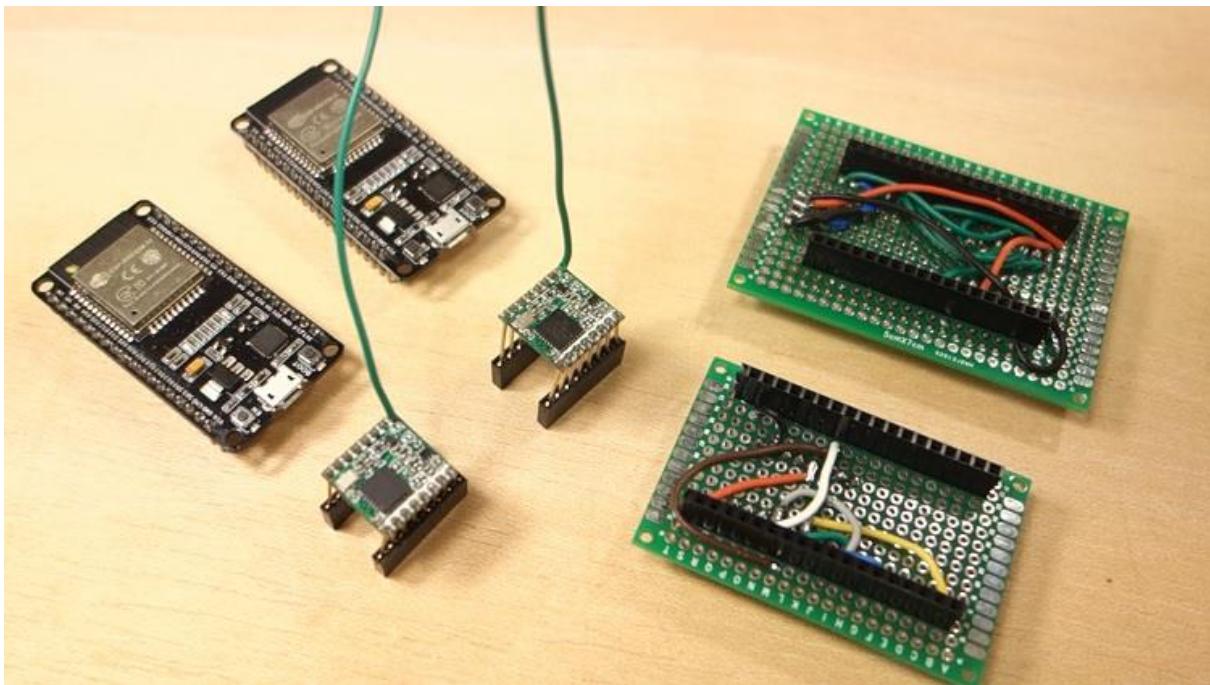
- You can use LoRa in point to point communication or in a network using LoraWAN;
- LoRa can be especially useful if you want to monitor sensors that are not covered by your Wi-Fi network.

LoRa Further Reading

As we've mentioned at the beginning of the post, LoRa is a pretty complex topic that would give a complete course itself. The following links provide more information about this subject if you want to explore it further.

- [LoRa for IoT \(epanorama website\)](#)
- [LoRa vs LoRaWAN \(libelium website\)](#)
- [The Things Network Kickstarter page](#)
- [The limits of LoRaWAN \(paper\)](#)

8.2 - ESP32 LoRa Sender and Receiver

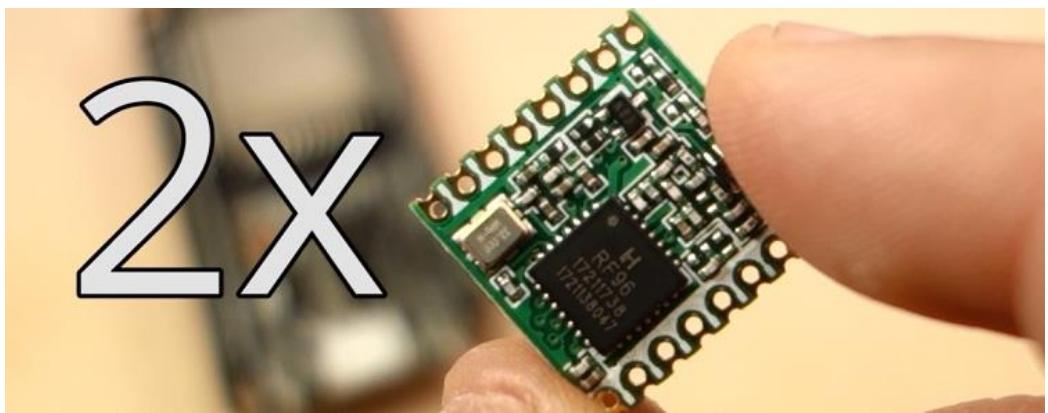


In this Unit, you'll program two ESP32 boards to exchange data using LoRa RF signals. You'll also test the module's communication range.

For this example, you'll need two ESP32 development boards, one will be the sender, and the other will be the receiver. The receiver will have an OLED display to print the received messages. We're just 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.

Getting LoRa Transceiver 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...

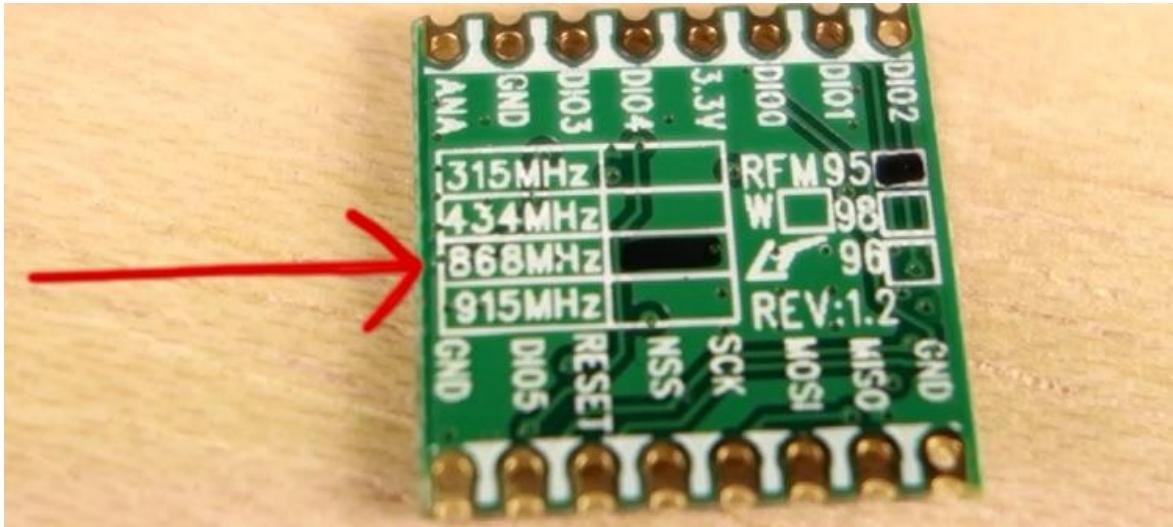
Alternatively, there are ESP32 boards with LoRa and OLED display built-in like the [ESP32 Heltec Wifi Module](#), or the [TTGO board](#).



We have a review and getting started guide for the TTGO LoRa SX1276 with built-in OLED:

- [Review](#)
- [Getting Started Guide](#)

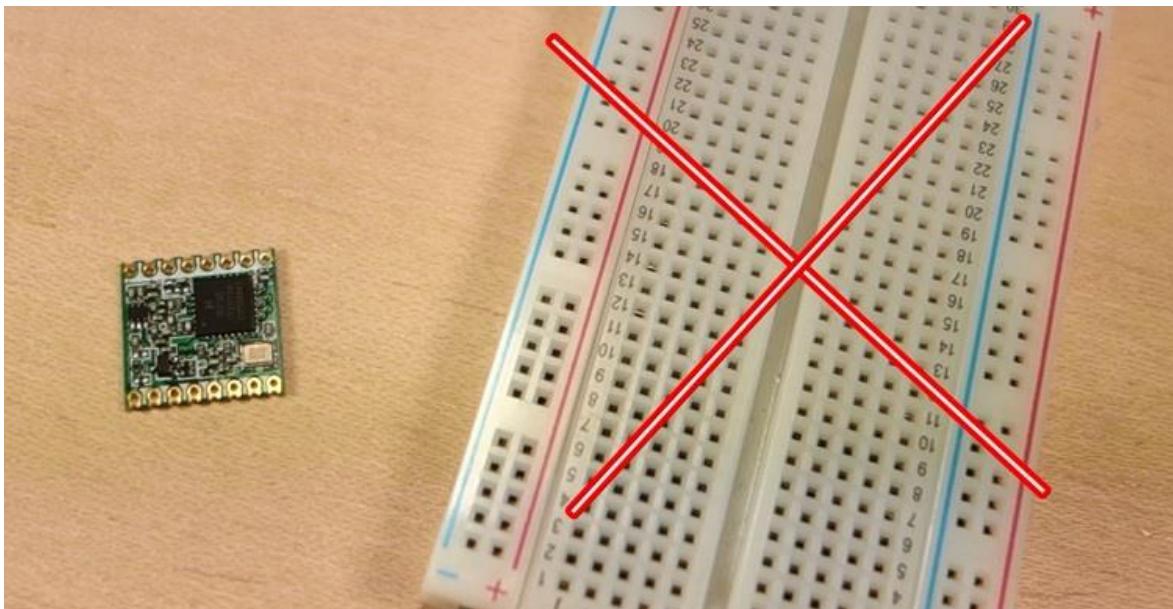
Before getting your LoRa transceiver module, make sure you check the correct frequency for your location. You can visit the following web page to learn more about [RF signals and regulations according to each country](#). For example, in Portugal, we can use a frequency between 863 and 870 MHz, or we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



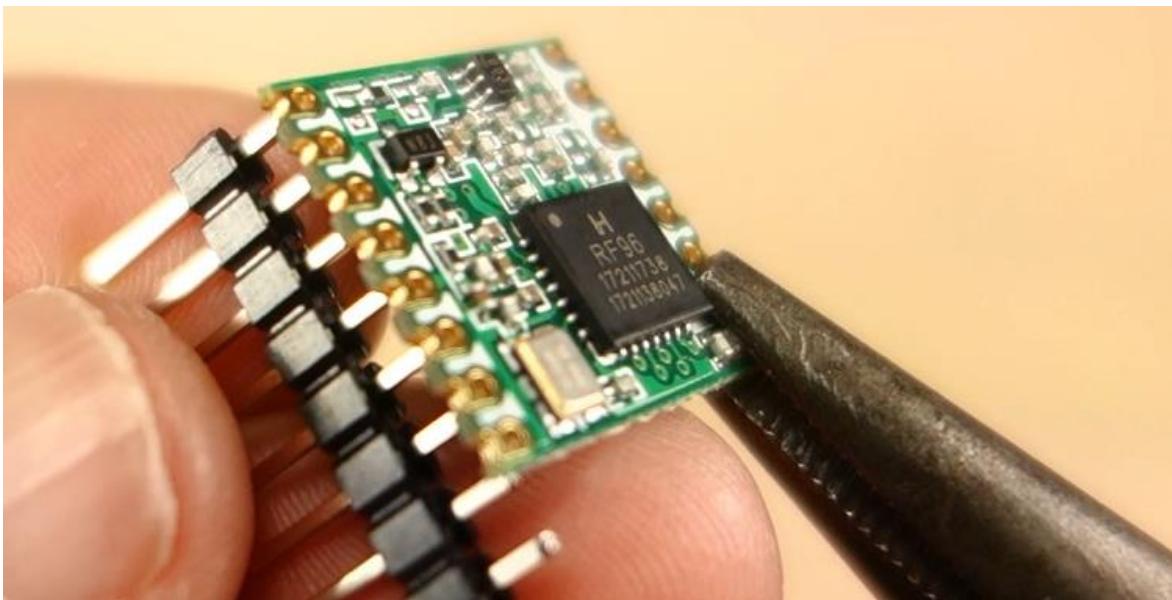
In the RFM95 module, the frequency is marked at the back, as you can see in the figure above. Make sure you are compliant with your country's regulations and buy the right board.

Preparing the RFM95 Transceiver Module

The RFM95 transceiver isn't breadboard friendly.



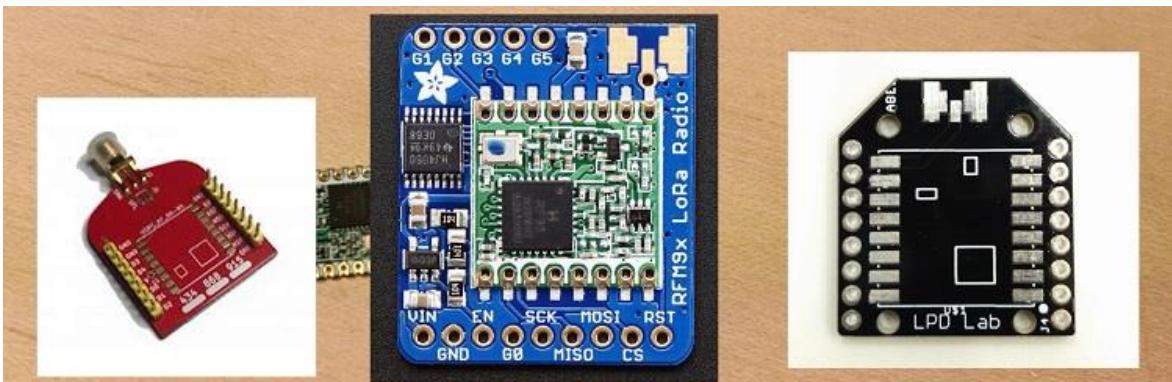
A standard row of 2.54mm header pins won't fit on the transceiver pins. The spaces between the connections are smaller than usual.



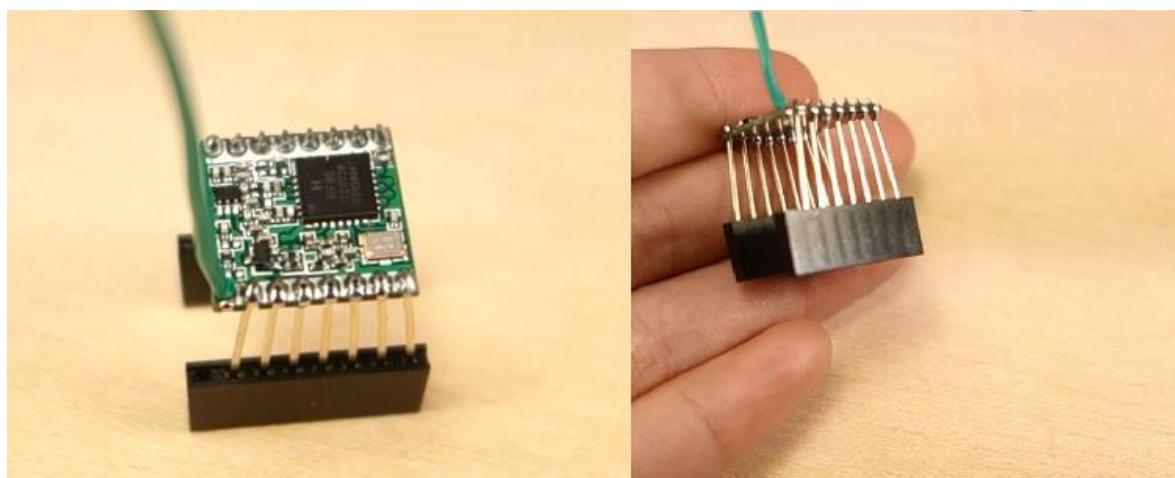
There are a few options that you can use to access the transceiver pins:

- 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.

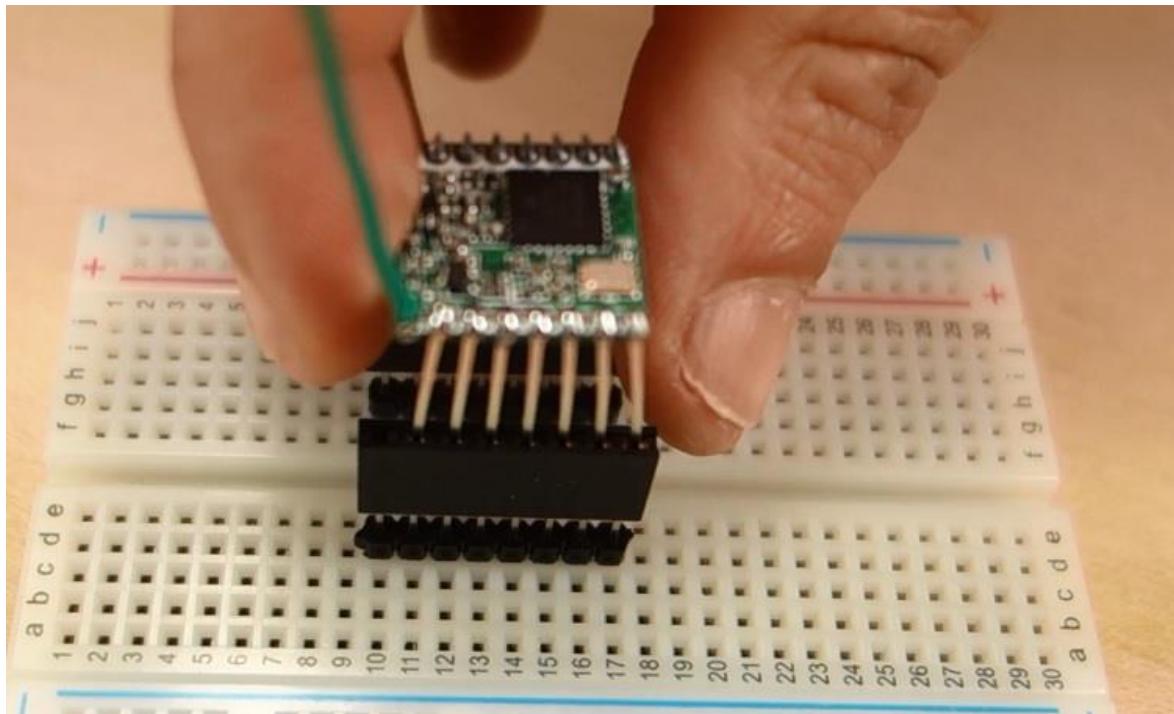
The figure below shows some examples of breakout boards for the RFM95.



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



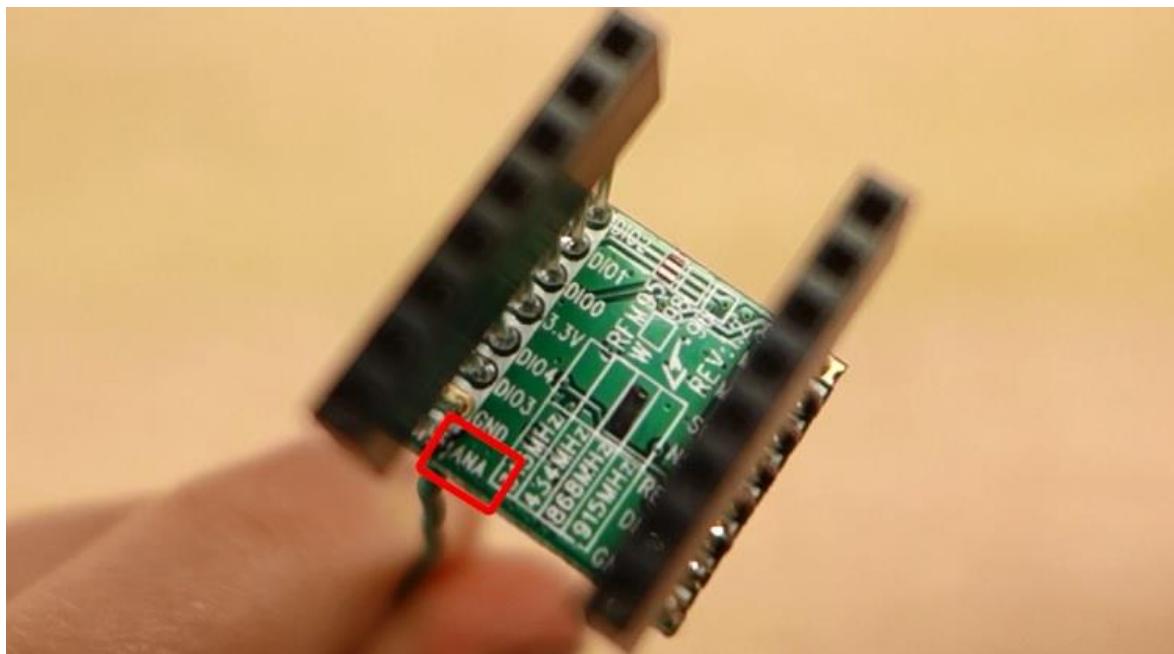
This way, you can access the module's pins with regular jumper wires, or even put some header pins to connect them directly to a stripboard or breadboard.



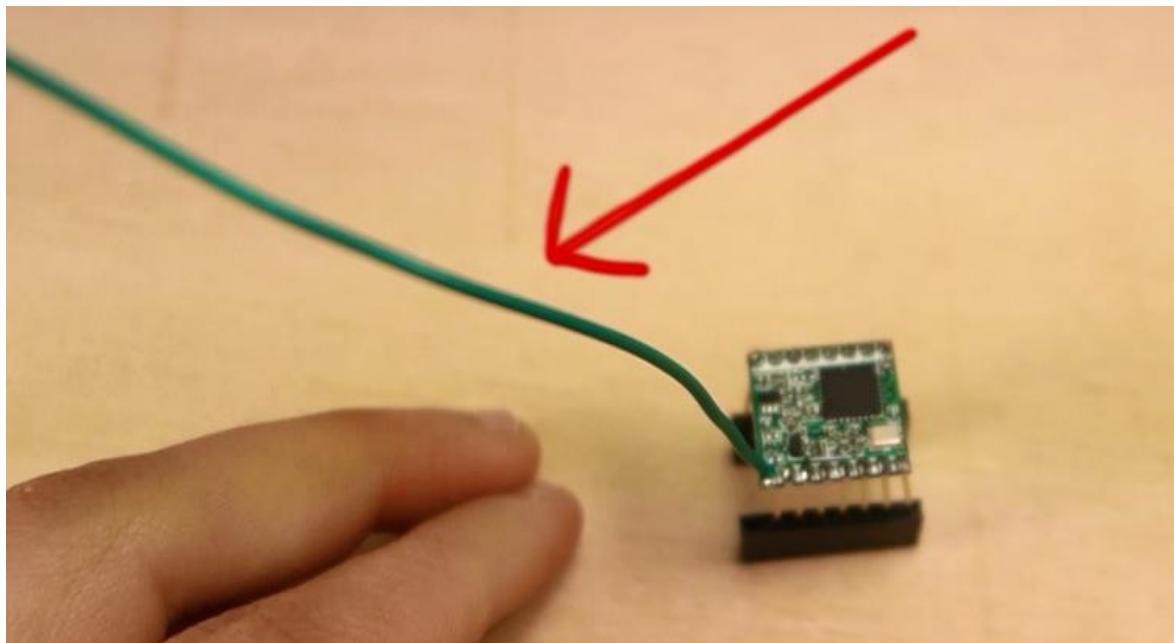
You don't need to worry about this if you're using a board with a built-in LoRa transceiver chip.

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'll need to use an 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.

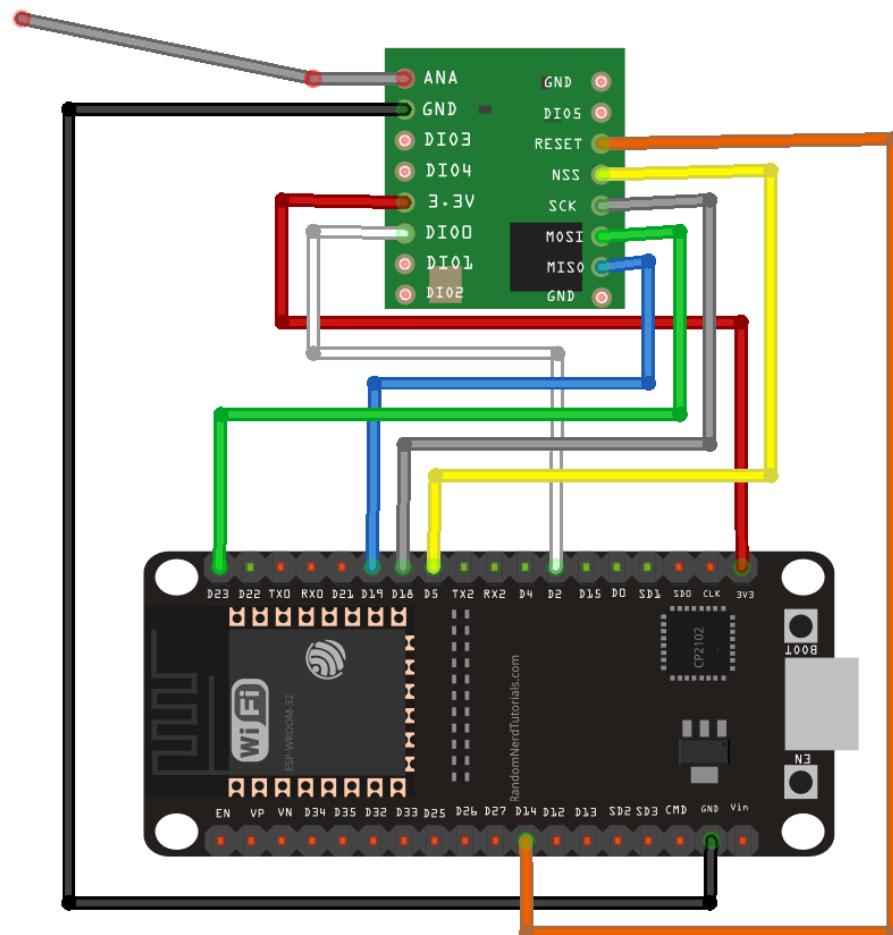
Preparing the LoRa sender

This project is divided into two parts. First, you'll set up and program the sender. Then, you'll prepare the receiver.

Wiring the Circuit

Let's build the sender circuit. With your RFM95 module with wires soldered or header pins and with an antenna, you can connect it to the ESP32 board. The module communicates with the ESP using the SPI communication protocol, so we'll use the ESP's SPI pins. Wire the ESP32 to the transceiver module as shown in the schematic diagram provided. Here's a list of parts you need to build the circuit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [LoRa Transceiver modules \(RFM95\)](#)
- RFM95 LoRa breakout board (optional)
- [Jumper wires](#)
- [Breadboard](#) or [stripboard](#)

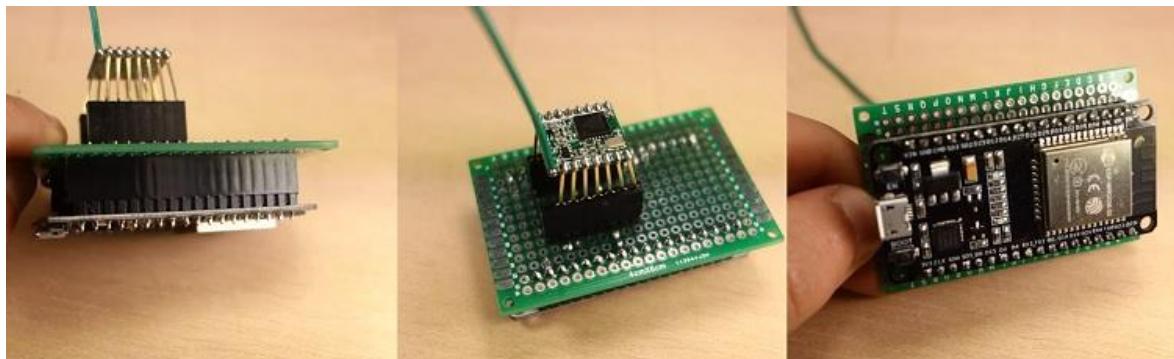


This schematic diagram is a bit confusing, so you can take a look at the following table instead.

RFM95 LoRa Transceiver Module			
ANA	Antenna	GND	-
GND	GND	DIO5	-
DIO3	-	RESET	GPIO 14
DIO4	-	NSS	GPIO 5
3.3V	3.3V	SCK	GPIO 18
DIO0	GPIO 2	MOSI	GPIO 23
DIO1	-	MISO	GPIO 19
DIO2	-	GND	-

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.

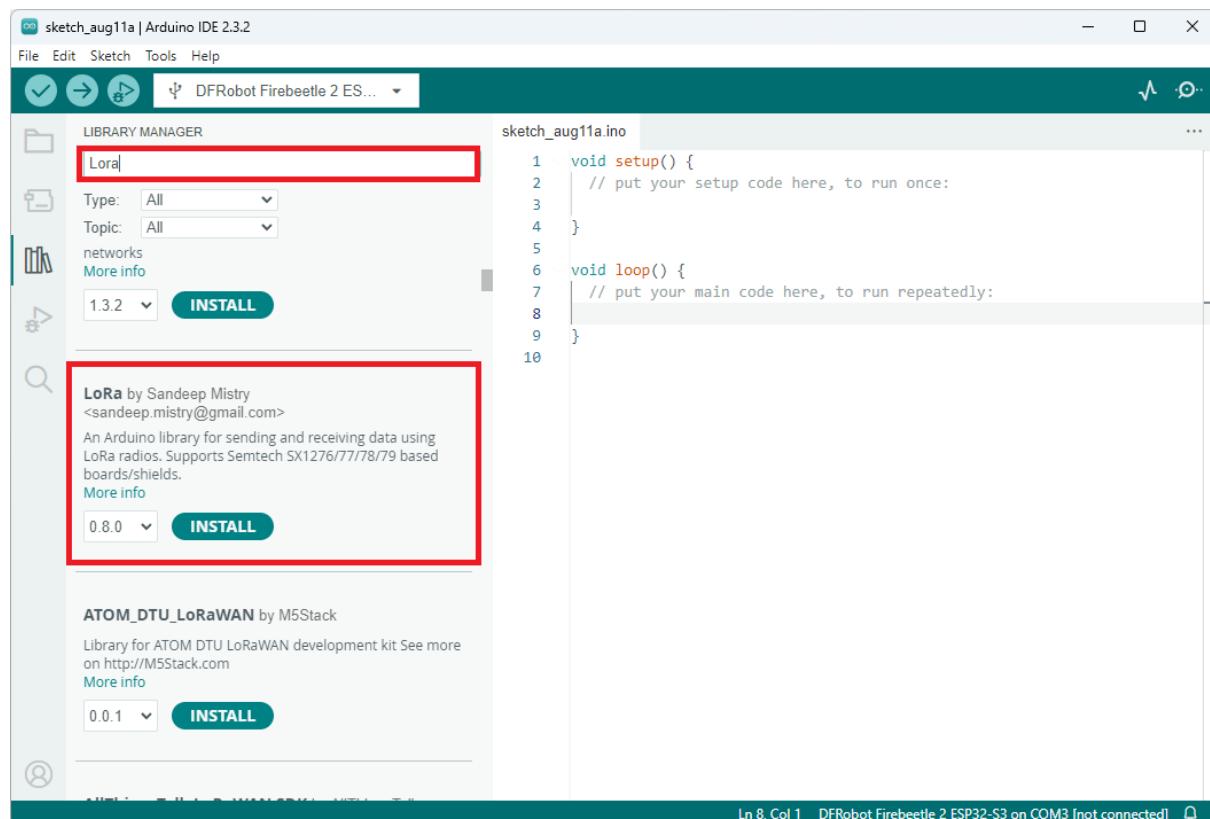


If you're using an ESP32 board with a built-in LoRa module, you don't need to wire any circuit. Just make sure you have your board pinout so that you know which pins are internally connected to the LoRa module.

Installing the LoRa Library

Before uploading any code to your ESP32, you need to install a LoRa library. There are several libraries available to easily send and receive LoRa packets with the ESP32. In this example, we'll use the [arduino-LoRa library by sandeep mistry](#).

Open your Arduino IDE, and go to **Sketch > Include Library > Manage Libraries** and search for “**LoRa**”. Scroll down and install the library developed by Sandeep Mistry.



The Arduino LoRa library is well documented on [GitHub](#).

The Sender Sketch

After installing the LoRa library, copy the Sender Sketch provided below to your Arduino IDE. 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.

- [Click here to download the code.](#)

```

#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 frequency
    //433E6 for Asia
    //866E6 for Europe
    //915E6 for North America
    if (!LoRa.begin(866E6)) {
        Serial.println("Starting LoRa failed!");
        while (1);
    }
    Serial.println("LoRa Initializing OK!");
}

void loop() {
    Serial.print("Sending packet: ");
    Serial.println(counter);

    // Send LoRa packet to receiver
    LoRa.beginPacket();
    LoRa.print("hello ");
    LoRa.print(counter);
    LoRa.endPacket();

    counter++;

    delay(10000);
}

```

Note: if you're using an ESP32 with a built-in LoRa module, please check the following tutorial instead where we set custom SPI pins for the module. Make sure you have the pinout for the board you're using.

- [TTGO LoRa32 SX1276 OLED Board: Getting Started with Arduino IDE](#)

How Does the Code Work?

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

Initialize the `counter` variable that starts at 0;

```
int counter = 0;
```

setup()

In the `setup()`, you initialize the Serial Monitor at a baud rate of 115200.

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

Set the pins for the LoRa module.

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

And initialize the transceiver module with a specified frequency.

```
if (!LoRa.begin(866E6)) {
  Serial.println("Starting LoRa failed!");
  while (1);
}
```

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

- 433E6
- 866E6
- 915E6

loop()

Next, in the `loop()`, you can start sending 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 Setup

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

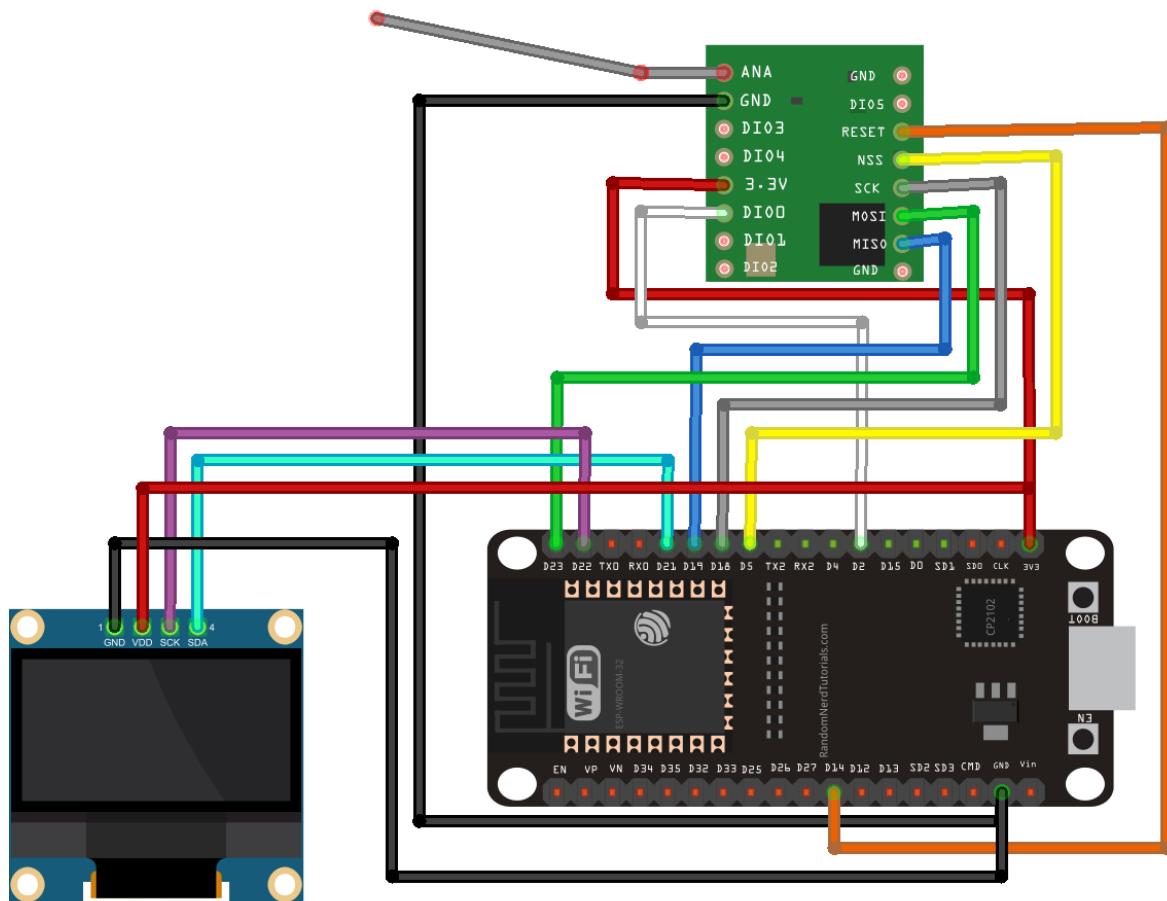
After that, open the Serial Monitor, and press the ESP32 enable button. You should see a success message, as shown in the figure below. The counter should be incremented every 10 seconds.



Preparing the LoRa Receiver

After making sure the LoRa initializes ok, let's prepare the LoRa receiver.

Wire the RFM95 transceiver module and the OLED display to the ESP32 as shown in the following schematic diagram. The OLED display we're using uses I2C communication protocol, so we'll use the ESP32 I2C pins: GPIOs 21 and 22.



You can also look at the following table to check the connections between the ESP32 and the transceiver.

RFM95 LoRa Transceiver Module

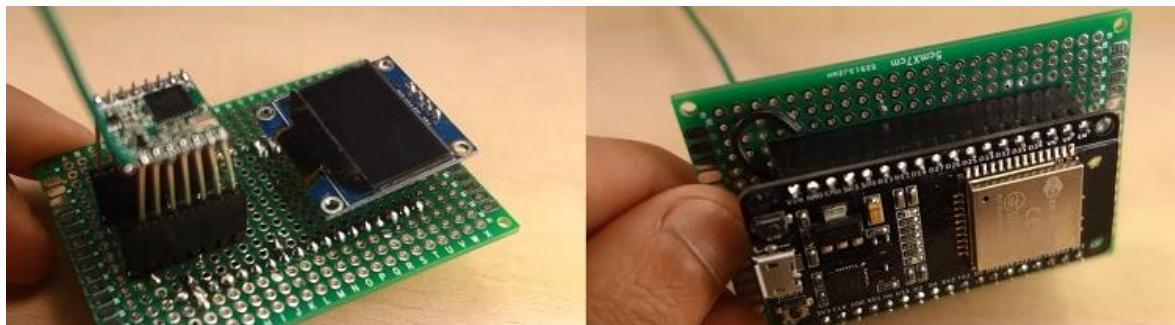
ANA	Antenna	GND	-
GND	GND	DIO5	-
DIO3	-	RESET	GPIO 14
DIO4	-	NSS	GPIO 5

3.3V	3.3V	SCK	GPIO 18
DIO0	GPIO 2	MOSI	GPIO 23
DIO1	-	MISO	GPIO 19
DIO2	-	GND	-

And the following table shows the connection of the ESP32 to the OLED display.

OLED Display	ESP32
GND	GND
VCC	3.3V
SCK	GPIO 22
SDA	GPIO 21

For practical reasons, we've made this circuit on a stripboard. The figure below shows how it looks like.

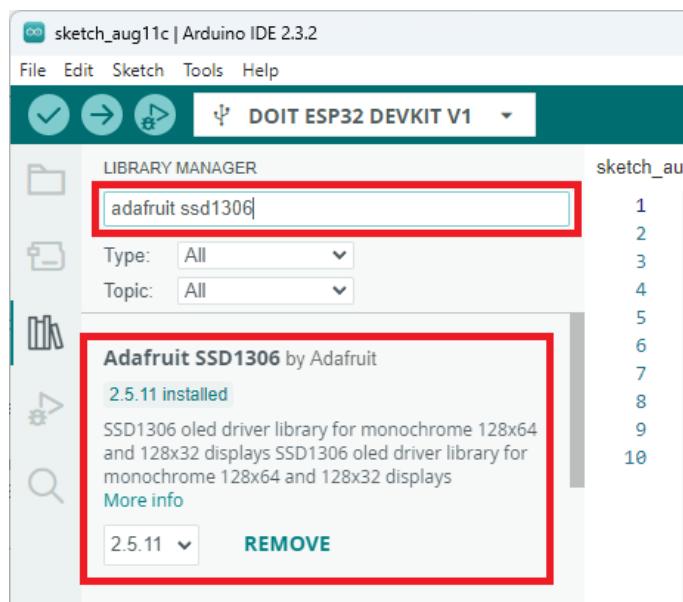


If you're using an ESP32 board with a built-in LoRa module and OLED display, you don't need to wire any circuit. Just make sure you have your board pinout so that you know which pins are internally connected to the LoRa module and to the OLED.

Installing the OLED Library

For the LoRa receiver, besides the LoRa library, you also need to install a library to write on the OLED display. We'll be using the [Adafruit SSD1306](#) library. If you don't have this library installed yet, follow the next steps to install it.

In your Arduino IDE, go to **Sketch > Include Library > Manage Libraries**. Search for "Adafruit SSD1306" and install it. Install any other required dependencies.



The Receiver Sketch

After installing the required libraries, copy the receiver sketch to your Arduino IDE. This sketch listens for LoRa packets within its range and prints the content of the packets on the OLED display and the RSSI. The RSSI measures the relative received signal strength.

- [Click here to download the code.](#)

```
#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include <Adafruit_SSD1306.h>
#include <Adafruit_GFX.h>

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

#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels
```

```

// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
#define OLED_RESET      4 // Reset pin # (or -1 if sharing Arduino reset pin)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

void setup() {
    // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
    if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for 128x32
        Serial.println(F("SSD1306 allocation failed"));
        for(;;); // Don't proceed, loop forever
    }

    display.clearDisplay();
    display.setTextSize(1);
    display.setTextColor(WHITE);
    display.setCursor(0,0);
    display.println("LoRa Receiver");
    display.display();

    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 frequency
    // note: the frequency should match the sender's frequency
    //433E6 for Asia
    //866E6 for Europe
    //915E6 for North America
    if (!LoRa.begin(866E6)) {
        Serial.println("Starting LoRa failed!");
        while (1);
    }
    Serial.println("LoRa Initializing OK!");
    display.setCursor(0,10);
    display.println("LoRa Initializing OK!");
    display.display();
}

void loop() {
    // try to parse packet
    int packetSize = LoRa.parsePacket();
    if (packetSize) {
        //received a packet
        Serial.print("Received packet ");
        display.clearDisplay();
        display.setCursor(0,0);
        display.print("Received packet ");
        display.display();

        // read packet
        while (LoRa.available()) {
            String LoRaData = LoRa.readString();
            Serial.print(LoRaData);
            display.setCursor(0,10);
            display.print(LoRaData);
            display.display();
        }
    }

    // print RSSI of packet
}

```

```

    int rssi = LoRa.packetRssi();
    Serial.print(" with RSSI ");
    Serial.println(rssi);
    display.setCursor(0,20);
    display.print("RSSI: ");
    display.setCursor(30,20);
    display.print(rssi);
    display.display();
}
}

```

How Does the Code Work?

It starts by including the needed libraries for the LoRa module and for the OLED display.

```

#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include <Adafruit_SSD1306.h>
#include <Adafruit_GFX.h>

```

As in the previous code, set the pins used by the transceiver module.

```

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

```

Define the OLED width and height:

```

#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels

```

You also need to create an object for the OLED display.

```

// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
#define OLED_RESET      4 // Reset pin # (or -1 if sharing Arduino reset pin)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

```

setup()

In the `setup()` initialize the display.

```

// SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for 128x32
  Serial.println(F("SSD1306 allocation failed"));
  for(;;); // Don't proceed, loop forever
}

```

Print the message “LoRa Receiver”.

```

display.clearDisplay();

```

```
display.setTextSize(1);
display.setTextColor(WHITE);
display.setCursor(0,0);
display.println("LoRa Receiver");
display.display();
```

You initialize the Serial Monitor and also print the message “LoRa Receiver” in your Serial monitor for debugging purposes.

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

Set the pins for the LoRa module.

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

And initialize the transceiver with a specified frequency.

```
if (!LoRa.begin(866E6)) {
  Serial.println("Starting LoRa failed!");
  while (1);
}
```

As in the previous sketch, you should change the frequency to match the one used in your location:

- 433E6
- 866E6
- 915E6

If the LoRa module is properly initialized, print a success message.

```
Serial.println("LoRa Initializing OK!");
display.setCursor(0,10);
display.println("LoRa Initializing OK!");
display.display();
```

loop()

In the `loop()` is where we'll receive the LoRa packets and print the received messages. Start by checking if there's a new packet available using the `parsePacket()` method.

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

If there's a new packet, we display a message on the OLED display as well as on the serial monitor saying "Received packet".

```
if (packetSize) {
    //received a packet
    Serial.print("Received packet ");
    display.clearDisplay();
    display.setCursor(0,0);
    display.print("Received packet ");
    display.display();
```

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

```
String LoRaData = LoRa.readString();
```

The incoming data is saved on the `LoRaData` variable and printed on the OLED display and in the serial monitor.

```
Serial.print(LoRaData);
display.setCursor(0,10);
display.print(LoRaData);
display.display();
```

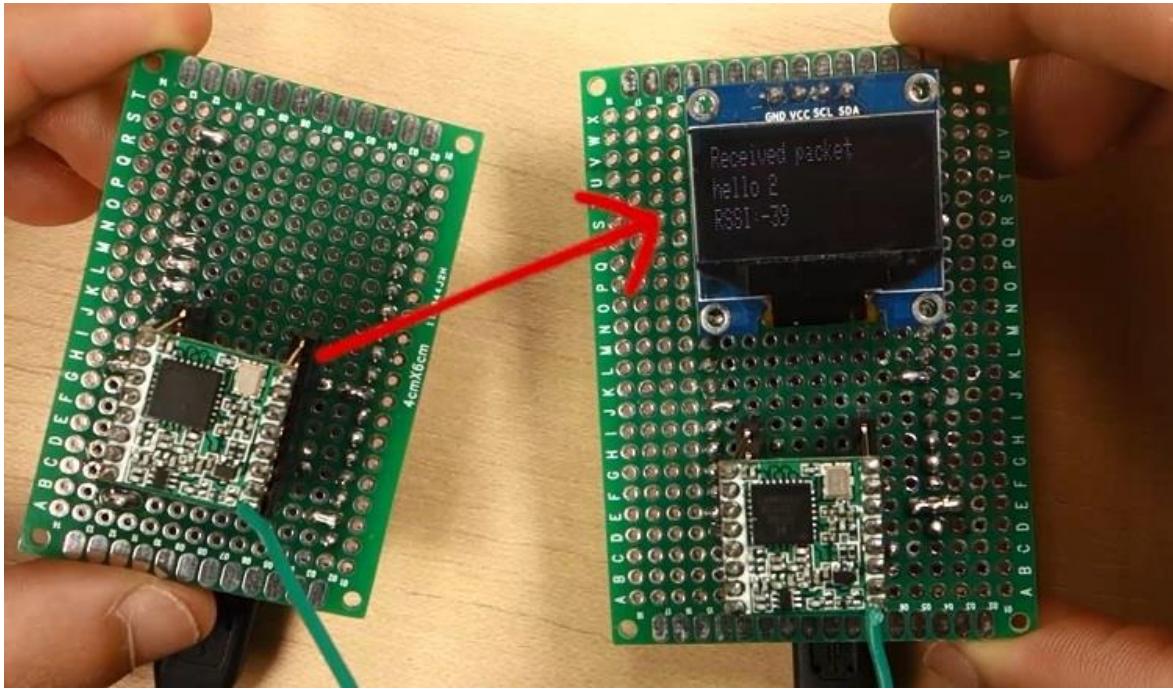
Finally, the next few lines of code print the RSSI of the received packet in dBm.

```
//print RSSI of packet
int rssi = LoRa.packetRssi();
Serial.print(" with RSSI ");
Serial.println(rssi);
display.setCursor(0,20);
display.print("RSSI: ");
display.setCursor(30,20);
display.print(rssi);
display.display();
```

Testing the Project

Upload this code to your ESP32. Now, let's test our setup!

Apply power to both ESP32 boards: the one running the receiver sketch, and the one running the sender sketch. You should start receiving messages on the ESP32 receiver board after a few seconds.

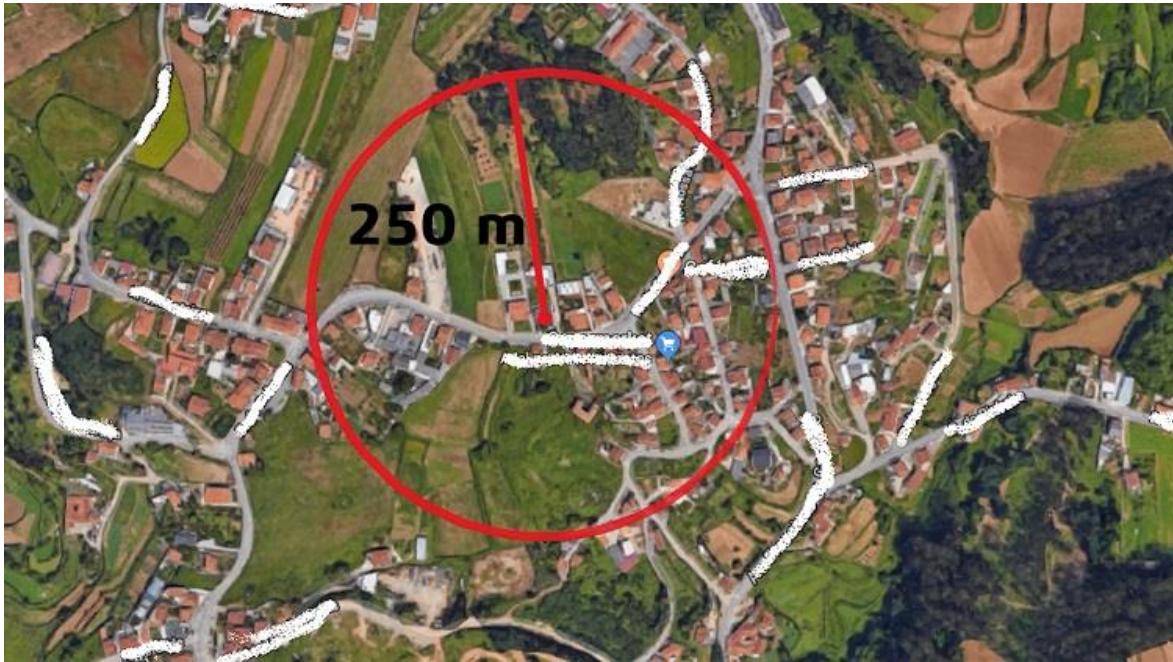


Testing the Communication Range

Now, test how far the two ESP32 can communicate using LoRa. Go for a walk with the receiver circuit powered by a portable charger, and let the other ESP32 at home.



With this setup, we got a stable communication between the two ESP32 board of up to 250 meters. This will greatly vary depending on when you are located—in an urban or rural area—if there are many buildings in between, etc.



Additionally, note that we're using the library's default definitions for LoRa, you may get a better communication range by changing some settings. We'll not explore this topic in this Unit, but there's a [discussion on GitHub on how to configure the LoRa library for better range](#). Feel free to take a look if you're interested.

Wrapping Up

In summary, this project involved sending a message using LoRa between two ESP32 boards. Keep in mind that this is just a simple example to test LoRa technology on the ESP32. There are some limitations with this setup:

- For example, the sender sends a LoRa packet that can be received by any receiver within its range, not necessarily by yours.
- The messages are not encrypted, which means that anyone can read your messages.
- The sender doesn't know whether the receiver got the package or not.
- The receiver catches any LoRa messages within its range. So, if your neighbor sends a LoRa message, your receiver will get it. So, you need something that identifies your messages.

8.3 - Further Reading about LoRa Gateways

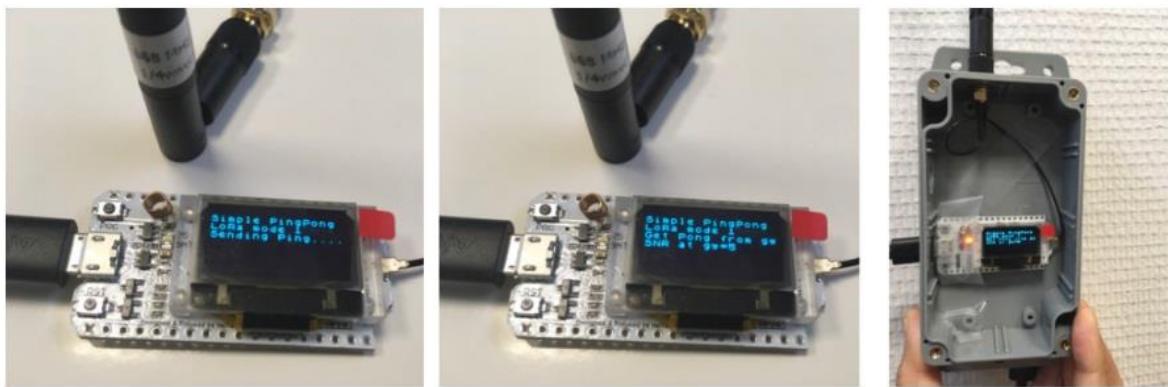
This Unit will give some extra information about LoRa gateways. We won't explore how to build a LoRa/LoRaWAN gateway in this course, but we'll provide you some information you can explore if you want to build one.

If you want to have your own gateway and contribute to [The Things Network](#) Community by sharing your gateway, there are plenty of options. You can build your own gateway, or you can buy a commercial solution (which is more expensive, but easier to set up). We'll show you some options and tutorials available online.

DIY Gateways

Many people around the world are building their own gateways using low-cost electronics components.

1. [LoRa Gateway with ESP32](#)



[View source](#)

You can build a LoRa gateway with the ESP32 using the previously mentioned RFM95 transceiver module or an ESP32 with built-in LoRa, and program it using the following repository: [LowCostLoRaGw](#).

2. Low Cost LoRa Gateway with Raspberry Pi



[View source](#)

This tutorial shows how to build a LoRa gateway with the Raspberry Pi and a low-cost LoRa transceiver module like the RFM95.

3. LoRa Gateway using the IMST iC880a board



This tutorial called “From zero to LoRaWAN in a weekend” is a tutorial on building a gateway using the IMST iC880a board with the Raspberry Pi. The iC880A is a

LoRaWAN Concentrator 868 MHz able to receive packets of different end devices and can be easily used in combination with Raspberry Pi, BeagleBone, Banana Pi, etc. The IMST iC880a board costs around \$150.

4. LoRaWAN Gateway with rak831 module and Raspberry Pi



Learn how to build a LoRaWAN gateway with the rak831 module and the Raspberry Pi. A developer kit with the rak831 module costs around \$150.

Commercial Gateways

You can also get commercial gateways and then connect them to "The Things Network"

They provide instructions for the most popular gateways:

- [LoRaWAN Gateways and Set up Instructions](#)

8.4 - LoRa: Where to Go Next?

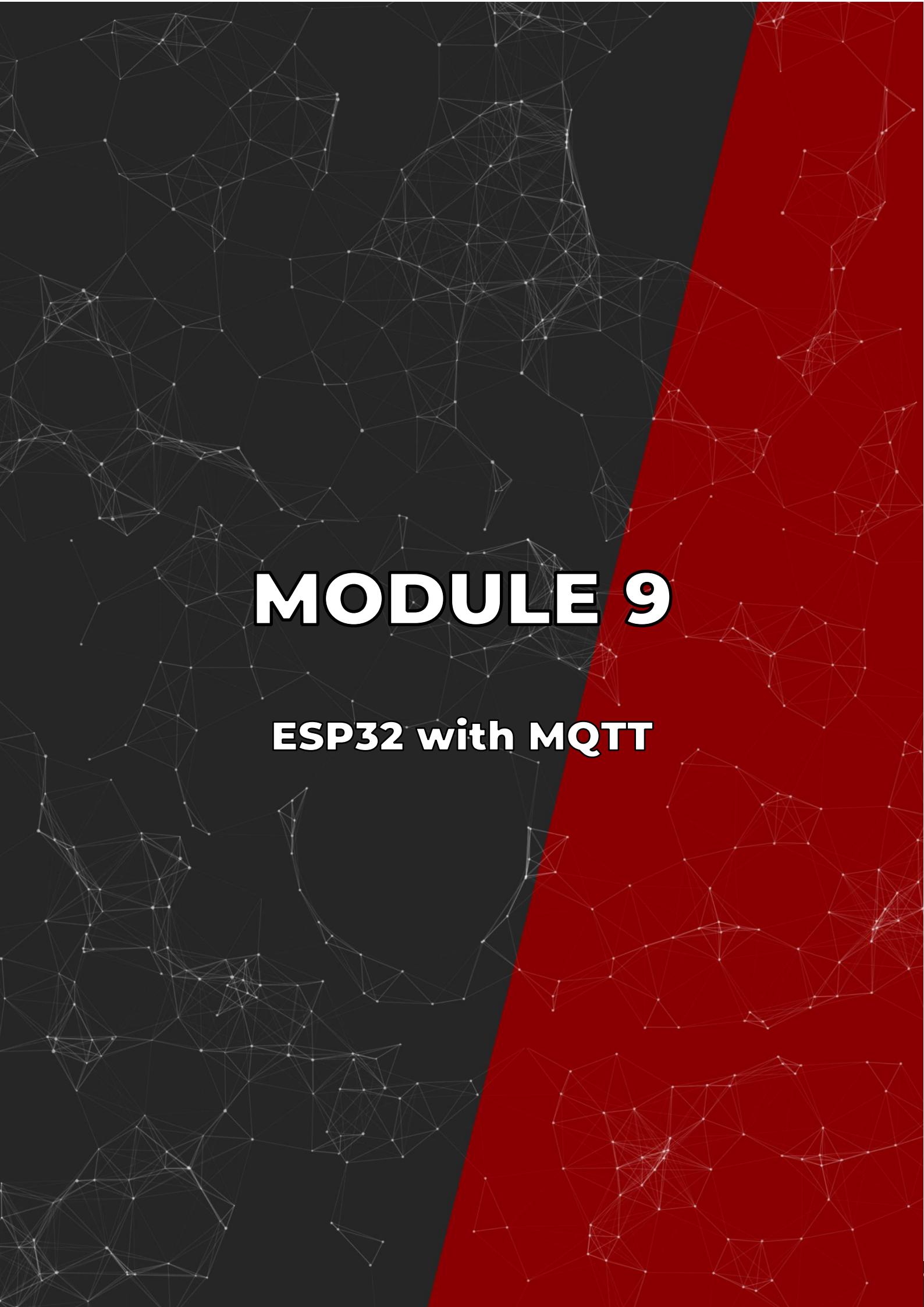
In this Module, we've taken a look at the LoRa technology. It is a powerful technology for IoT as it allows long-range communication with low power consumption.



LoRa can be especially useful in your home automation projects to monitor sensors that are not covered by your Wi-Fi network.

In a previous Unit, we've tested the LoRa technology in your ESP32 board. We've sent a LoRa packet from one ESP32 to another using LoRa RF signals. This project was a simple one to show you and test LoRa in your ESP32.

[In Project 4](#), we'll show you how to integrate LoRa in your IoT projects to get readings from sensor nodes outside your Wi-Fi network (up to 250m from your house) and publish those readings on a web server.



MODULE 9

ESP32 with MQTT

9.1 - Introducing MQTT

MQTT stands for **M**essage **Q**ueuing **T**elemetry **T**ransport. MQTT is a simple messaging protocol, designed for constrained devices with low bandwidth. So, it's the perfect solution to exchange data between multiple IoT devices.

MQTT communication works as a publish and subscribe system. Devices publish messages on a specific topic. All devices that are subscribed to that topic receive the message.



Its main applications include sending messages to control outputs, reading and publishing data from sensor nodes, and much more.

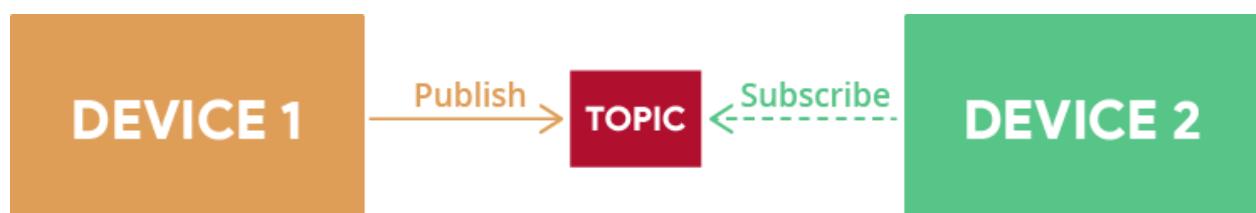
MQTT Basic Concepts

In MQTT there are a few basic concepts that you need to understand:

- Publish/Subscribe
- Messages
- Topics
- Broker

MQTT Publish/Subscribe

The first concept is the *publish/subscribe* system. In a publish and subscribe system, a device can publish a message on a topic, or it can be subscribed to a particular topic to receive messages



- 1) For example, Device 1 publishes on a topic.
- 2) Device 2 is subscribed to the same topic that Device 1 is publishing in.
- 3) So, Device 2 receives the message.

MQTT Messages

Messages are the information that you want to exchange between your devices. It can be a message like a command to control an output or data like sensor readings, for example.

MQTT Topics

Another important concept is the *topics*. Topics are the way you register interest in incoming messages or how you specify where you want to publish the message.

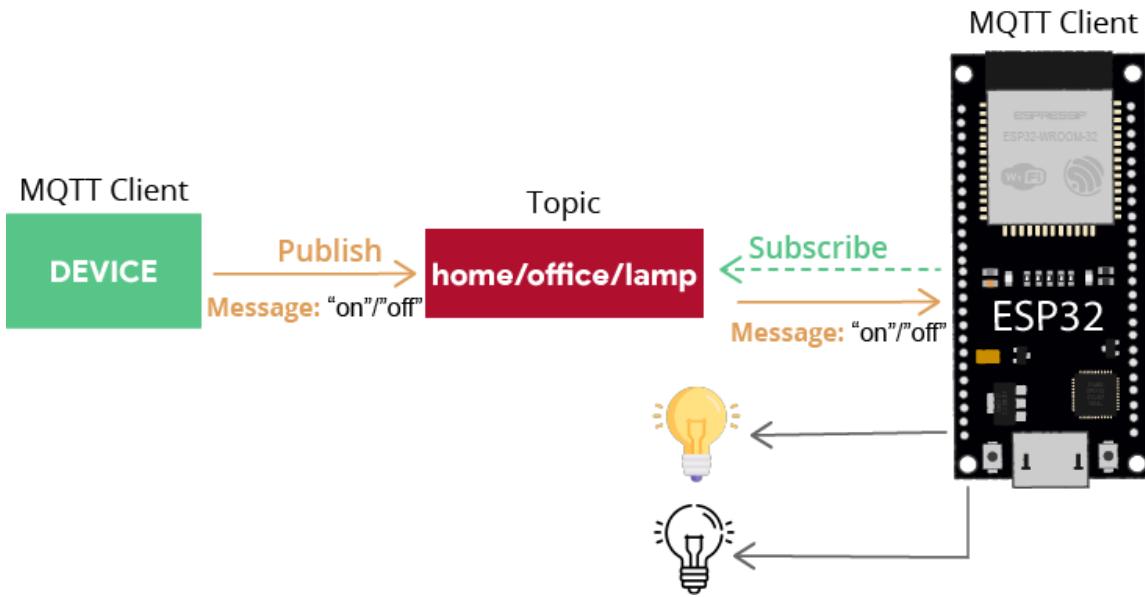
Topics are represented with strings separated by a forward slash. Each forward slash indicates a topic level. Here's an example of how you would create a topic for a lamp in your home office:



Note: topics are case-sensitive, which makes these two topics different:

home/office/lamp
≠
home/office/LAmp

If you would like to turn on a lamp in your home office using MQTT you can imagine the following scenario:



A device publishes “on” and “off” messages on the **home/office/lamp** topic.

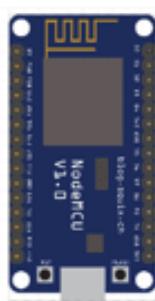
You have a device that controls a lamp (it can be an ESP32, ESP8266, or any other board or device). The ESP32 that controls your lamp, is subscribed to that same topic: **home/office/lamp**.

So, when a new message is published on that topic, the ESP32 receives the “on” or “off” messages and turns the lamp on or off.

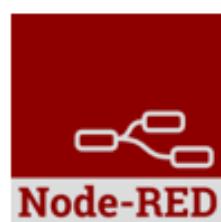
The device that is publishing the messages is another MQTT client, which can be an ESP32, an ESP8266, or a Home Automation controller platform with MQTT support like Node-RED, Home Assistant, Domoticz, or OpenHAB, for example. In our case, we'll be using Node-RED.



ESP32



ESP8266



Domoticz

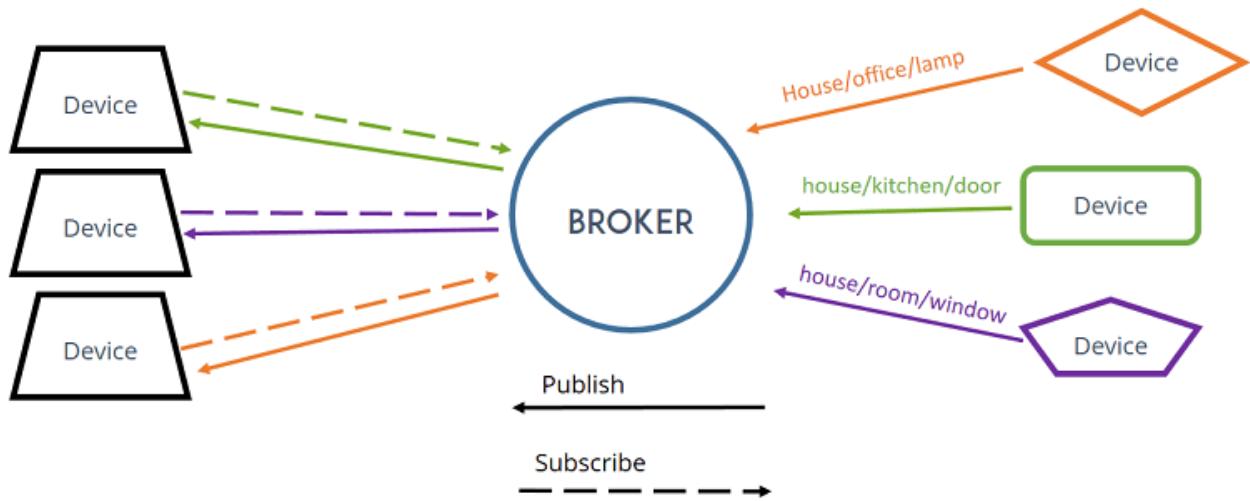


openHAB
empowering the smart home

MQTT Broker

Finally, another important concept is the *broker*.

The MQTT broker is responsible for receiving all messages, filtering the messages, deciding who is interested in them, and then publishing the message to all subscribed clients.



There are several brokers you can use. For example:

1. **Local MQTT broker:** you can install an MQTT broker locally on your computer or on your Raspberry Pi. The **Mosquitto** MQTT broker hosted on a Raspberry Pi is widely used in many hobbyist projects and it's also the solution we use more often.
2. **Cloud MQTT Broker:** as an alternative to the previous solution, you can also install MQTT broker on your own cloud server.
3. **Cloud MQTT Broker commercial solutions:** you can use commercial MQTT broker solutions like [HiveMQ](#), for example. You don't need to set up anything. You just need to create an account and you're ready to go.

In home automation projects, we mostly use the [Mosquitto Broker](#) installed on a Raspberry Pi. You can also install the Mosquitto broker on your PC (which is not as convenient as using a Raspberry Pi board, because you have to keep your computer running all the time to keep the MQTT connection between your devices).



Having the Mosquitto broker installed on a Raspberry Pi on your local network allows you to exchange data between IoT devices that are also connected to that same network.

In the next unit, we'll show you how to install the Mosquitto broker on your Raspberry Pi board.

Alternatively, you can also run Mosquitto MQTT broker in the cloud. Running the MQTT Mosquitto Broker in the cloud allows you to connect several IoT devices from anywhere using different networks as long as they have an Internet connection. We have a [tutorial about this subject here](#).

Wrapping Up

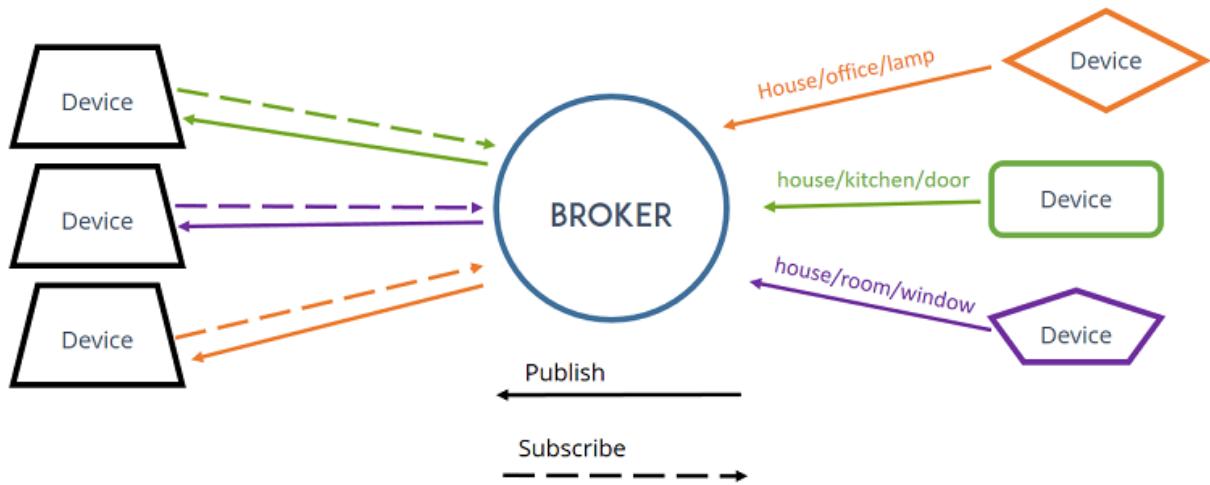
In summary:

- MQTT is a communication protocol very useful in the Internet of Things projects;
- In MQTT, devices can publish messages on specific topics and can be subscribed to other topics to receive messages;
- You need a broker when using MQTT. It receives all the messages and sends them to the subscribed devices.

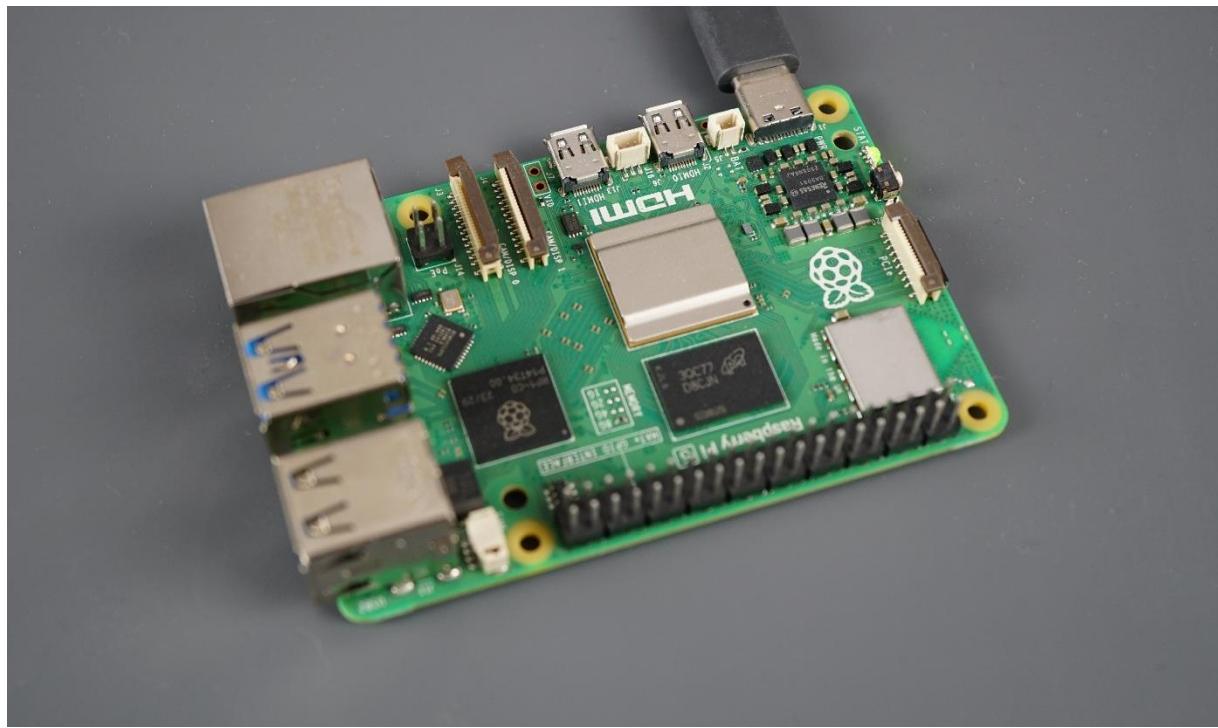
In the next three Units, you'll learn how to install an MQTT broker and exchange data between two ESP32 boards using the MQTT protocol.

9.2 - Installing Mosquitto MQTT Broker on a Raspberry Pi

In this Unit, you're going to install the Mosquitto Broker on a Raspberry Pi. The broker is primarily responsible for **receiving** all messages, **filtering** the messages, decide who is interested in them and then, **publishing** the messages to all subscribed clients.



There are several brokers you can use. For this Module, we're going to use the [Mosquitto Broker](#) installed on a Raspberry Pi.



Note: you can also use a free [Cloud MQTT](#) broker (for a maximum of 5 connected devices). Learn [how to use Cloud MQTT broker with your ESP32](#).

Prerequisites

Before continuing with this tutorial:

1. You should be familiar with the Raspberry Pi board – [read Getting Started with Raspberry Pi](#);
2. You should have the Raspberry Pi OS installed in your Raspberry Pi – read [Install Raspberry Pi OS, Set Up Wi-Fi, Enable and Connect with SSH](#);
3. You also need the following hardware:
 - [Raspberry Pi board](#) – read [Best Raspberry Pi Starter Kits](#)
 - [MicroSD Card – 32GB Class10](#)
 - [Raspberry Pi Power Supply \(5V 2.5A\)](#)

After having your Raspberry Pi board prepared with Raspberry Pi OS, you can continue with this Unit.

Installing Mosquitto Broker on the Raspberry Pi

Let's install the Mosquitto Broker.



- 1) Open a new SSH connection or a terminal window on your Raspberry Pi.

A screenshot of a terminal window titled "pi@raspberrypi: ~". The window has a dark background and a light-colored text area. At the top left, there is a small icon of a user profile. The title bar shows the session information. The main area is mostly empty, with a few small green and white characters visible at the bottom left corner, likely from a previous command's output.

2) Run the following command to upgrade and update your system:

```
sudo apt update && sudo apt upgrade
```

- 3) When asked, press **Y** and **Enter**. It will take some time to update and upgrade (in my case, it took approximately 10 minutes, but it may take longer).
- 4) To install the Mosquitto Broker and Mosquitto Clients, enter the next command:

```
sudo apt install -y mosquitto mosquitto-clients
```

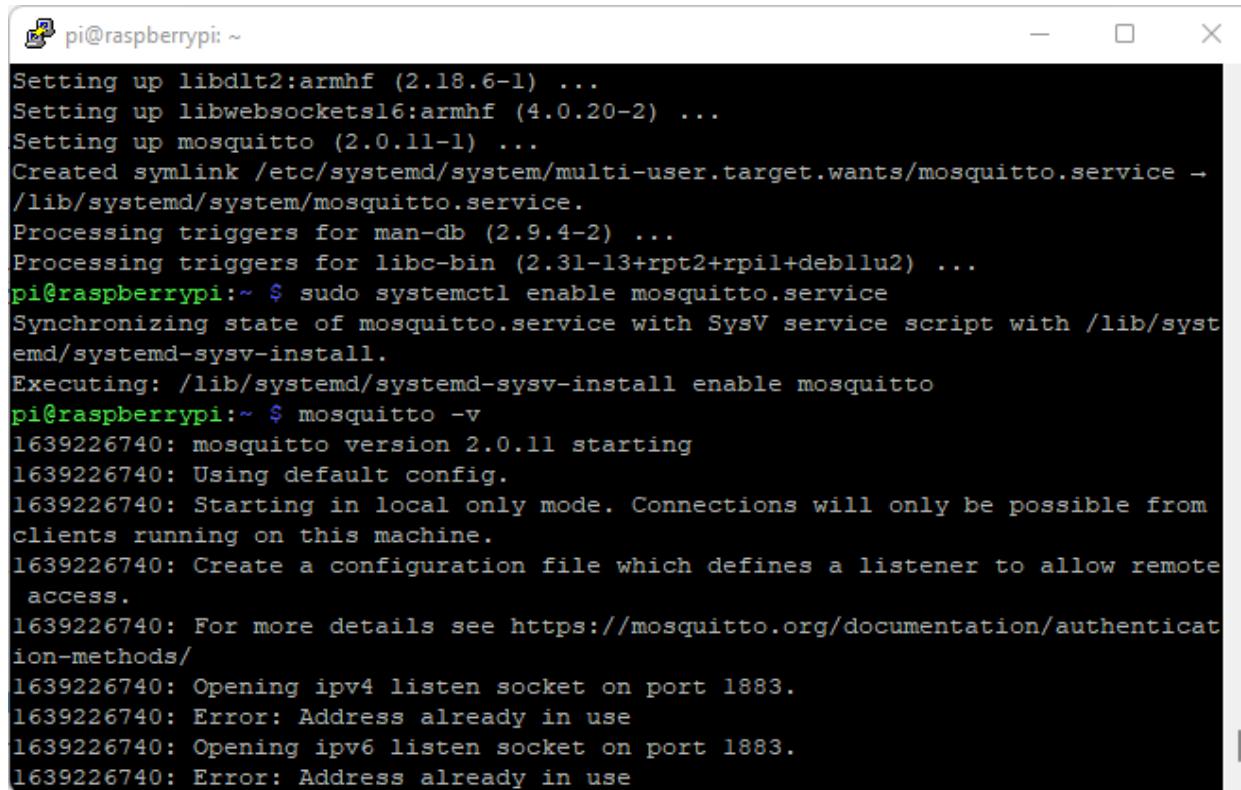
- 5) To make Mosquitto auto start when the Raspberry Pi boots, you need to run the following command (this means that the Mosquitto broker will automatically start when the Raspberry Pi starts):

```
sudo systemctl enable mosquitto.service
```

- 6) Now, test the installation by running the following command:

```
mosquitto -v
```

This returns the Mosquitto version that is currently running on your Raspberry Pi. It will be 2.0.11 or above.



The screenshot shows a terminal window titled 'pi@raspberrypi: ~'. The window contains the following text output:

```
Setting up libdlt:armhf (2.18.6-1) ...
Setting up libwebsockets16:armhf (4.0.20-2) ...
Setting up mosquitto (2.0.11-1) ...
Created symlink /etc/systemd/system/multi-user.target.wants/mosquitto.service →
/lib/systemd/system/mosquitto.service.
Processing triggers for man-db (2.9.4-2) ...
Processing triggers for libc-bin (2.31-13+rpt2+rpil+deb11u2) ...
pi@raspberrypi:~ $ sudo systemctl enable mosquitto.service
Synchronizing state of mosquitto.service with SysV service script with /lib/systemd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable mosquitto
pi@raspberrypi:~ $ mosquitto -v
1639226740: mosquitto version 2.0.11 starting
1639226740: Using default config.
1639226740: Starting in local only mode. Connections will only be possible from
clients running on this machine.
1639226740: Create a configuration file which defines a listener to allow remote
access.
1639226740: For more details see https://mosquitto.org/documentation/authentication-methods/
1639226740: Opening ipv4 listen socket on port 1883.
1639226740: Error: Address already in use
1639226740: Opening ipv6 listen socket on port 1883.
1639226740: Error: Address already in use
```

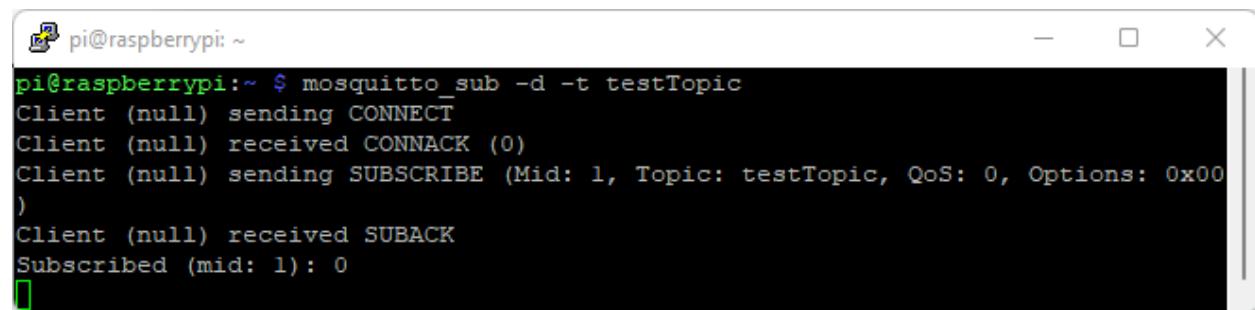
It will prompt the following message: “*Starting in local only mode. Connections will only be possible from clients running on this machine. Create a configuration file which defines a listener to allow remote access.*”

This means that by default, you can’t communicate with the Mosquitto broker from another device. We don’t want this to happen, we want to be able to access using the ESP32 and/or ESP8266 boards. We’ll take a look at this in a few moments.

Testing MQTT Broker Installation

To subscribe to an MQTT topic with Mosquitto Client open a terminal Window #1 and enter the command:

```
mosquitto_sub -d -t testTopic
```



```
pi@raspberrypi: ~ $ mosquitto_sub -d -t testTopic
Client (null) sending CONNECT
Client (null) received CONNACK (0)
Client (null) sending SUBSCRIBE (Mid: 1, Topic: testTopic, QoS: 0, Options: 0x00
)
Client (null) received SUBACK
Subscribed (mid: 1): 0
```

You’re now subscribed to a topic called **testTopic**.

To publish a sample message to **testTopic**, open a terminal Window #2 (a new SSH connection with your Pi or a new terminal window) and run the following command:

```
mosquitto_pub -d -t testTopic -m "Hello world!"
```

```

pi@raspberrypi: ~ $ mosquitto_sub -d -t testTopic
Client mosqsub/867-raspberrypi sending CONNECT
Client mosqsub/867-raspberrypi received CONNACK
Client mosqsub/867-raspberrypi sending SUBSCRIBE (Mid: 1, Topic: testTopic, QoS: 0)
Client mosqsub/867-raspberrypi received SUBACK
Subscribed (mid: 1): 0
Client mosqsub/867-raspberrypi received PUBLISH (d0, q0, r0, m0, 'testTopic', ... (12 bytes))
Hello world!
[green square icon]

Window #1

```



```

pi@raspberrypi: ~ $ mosquitto_pub -d -t testTopic -m "Hello world!"
Client mosqpub/868-raspberrypi sending CONNECT
Client mosqpub/868-raspberrypi received CONNACK
Client mosqpub/868-raspberrypi sending PUBLISH (d0, q0, r0, m1, 'testTopic', ... (12 bytes))
Client mosqpub/868-raspberrypi sending DISCONNECT
pi@raspberrypi: ~ $ [green square icon]

Window #2

```

The message “Hello World!” is received in Window #1 as illustrated in the figure above. This means everything is working as expected.

Enable Remote Access/Authentication

To enable remote access so that we can communicate with other IoT devices, we need to edit/create a configuration file. You can edit the configuration file with one of the following options:

- No authentication;
- Authentication with user and password.

Choose the section that is more suitable for your scenario.

Mosquitto Broker Enable Remote Access (No Authentication)

1. Run the following command to open the *mosquitto.conf* file.

```
sudo nano /etc/mosquitto/mosquitto.conf
```

2. Move to the end of the file using the arrow keys and paste the following two lines:

```
listener 1883
allow_anonymous true
```

3. Then, press **CTRL-X** to exit and save the file. Press **Y** and **Enter**.
4. Restart Mosquitto for the changes to take effect.

```
sudo systemctl restart mosquitto
```

Mosquitto Broker Enable Remote Access (Authentication: user and password)

1. Run the following command, but replace **YOUR_USERNAME** with the username you want to use:

```
sudo mosquitto_passwd -c /etc/mosquitto/passwd YOUR_USERNAME
```

I'll be using the MQTT user **sara**, so I run the command as follows:

```
sudo mosquitto_passwd -c /etc/mosquitto/passwd sara
```

When you run the preceding command with the desired username, you'll be asked to enter a password. No characters will be displayed while you enter the password. Enter the password and memorize the user/pass combination, you'll need it later in your projects to make a connection with the broker.

This previous command creates a password file called **passwd** on the **/etc/mosquitto** directory.

Now, we need to edit the Mosquitto configuration file so that it only allows authentication with the username and password we've defined.

2. Run the following command to edit the configuration file:

```
sudo nano /etc/mosquitto/mosquitto.conf
```

3. Add the following line at the top of the file (make sure it is at the top of the file, otherwise it won't work):

```
per_listener_settings true
```

4. Add the following three lines to allow connection for authenticated users and tell Mosquitto where the username/password file is located.

```
allow_anonymous false
listener 1883
```

```
password_file /etc/mosquitto/passwd
```

Your configuration file will look as follows (the new lines are in bold/yellow):

```
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

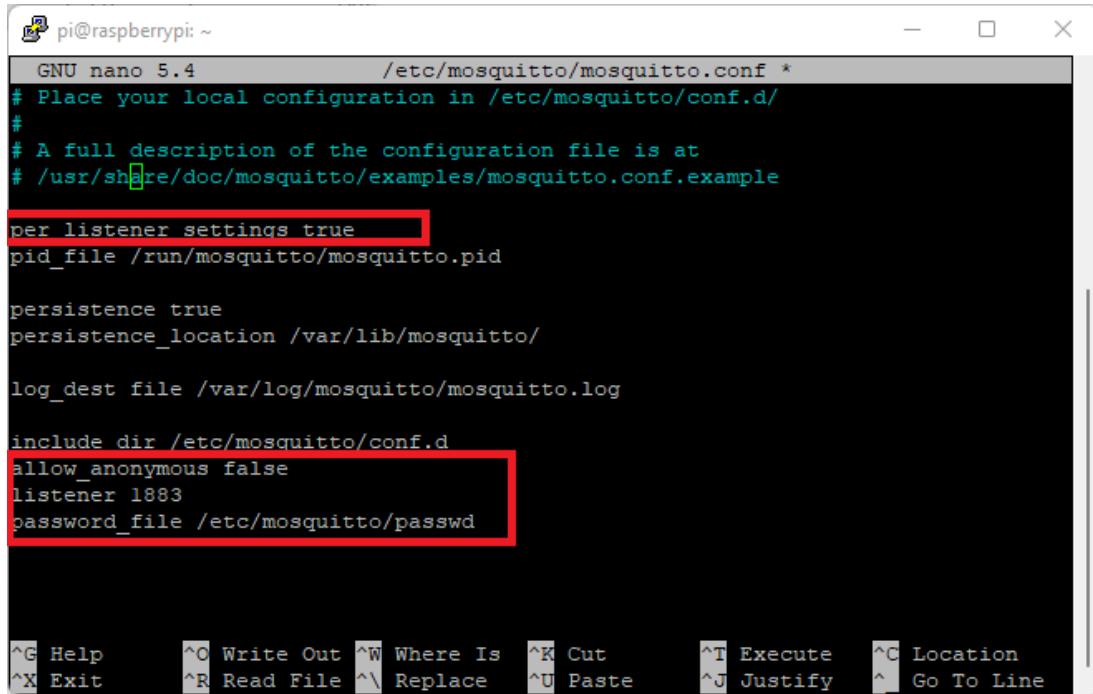
per_listener_settings true

pid_file /run/mosquitto/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d
allow_anonymous false
listener 1883
password_file /etc/mosquitto/passwd
```



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window shows the contents of the file "/etc/mosquitto/mosquitto.conf". The "password_file /etc/mosquitto/passwd" line is highlighted with a red box. The terminal also displays a set of keyboard shortcuts at the bottom.

```
GNU nano 5.4          /etc/mosquitto/mosquitto.conf *
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

per_listener_settings true
pid_file /run/mosquitto/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d
allow_anonymous false
listener 1883
password_file /etc/mosquitto/passwd

^G Help      ^O Write Out  ^W Where Is   ^K Cut      ^T Execute   ^C Location
^X Exit      ^R Read File  ^\ Replace    ^U Paste    ^J Justify   ^
^L Go To Line
```

5. Press **CTRL-X**, then **Y**, and finally press **Enter** to exit and save the changes.
6. Restart Mosquitto for the changes to take effect.

```
sudo systemctl restart mosquitto
```

Wait a few seconds. Then, to check if Mosquitto is running, you can type the following command:

```
sudo systemctl status mosquitto
```

```
pi@raspberrypi:~ $ sudo mosquitto_passwd -c /etc/mosquitto/passwd sara
Password:
Reenter password:
pi@raspberrypi:~ $ sudo nano /etc/mosquitto/mosquitto.conf
pi@raspberrypi:~ $ sudo systemctl restart mosquitto
pi@raspberrypi:~ $ sudo systemctl status mosquitto
● mosquitto.service - Mosquitto MQTT Broker
   Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled; vendor pre>
   Active: active (running) since Mon 2022-08-29 18:22:55 WEST; 27s ago
     Docs: man:mosquitto.com(5)
           man:mosquitto(8)
  Process: 13791 ExecStartPre=/bin/mkdir -m 740 -p /var/log/mosquitto (code=exit>
  Process: 13792 ExecStartPre=/bin/chown mosquitto /var/log/mosquitto (code=exit>
  Process: 13793 ExecStartPre=/bin/mkdir -m 740 -p /run/mosquitto (code=exit>
  Process: 13794 ExecStartPre=/bin/chown mosquitto /run/mosquitto (code=exit>
 Main PID: 13795 (mosquitto)
   Tasks: 1 (limit: 4164)
     CPU: 81ms
    CGroup: /system.slice/mosquitto.service
              └─13795 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

Aug 29 18:22:55 raspberrypi systemd[1]: Starting Mosquitto MQTT Broker...
Aug 29 18:22:55 raspberrypi systemd[1]: Started Mosquitto MQTT Broker.
lines 1-17/17 (END)
```

Now, you have authentication with username and password enabled.

If you try to publish messages using the same command we used before, it won't work because we need to provide a username and password. We'll see how to do that soon.

Remember that every time you want to communicate with the broker, you'll need to provide the username and password.

Add More Users/Change Password

To add more users to an existing password file, or to change the password for an existing user, leave out the `-c` argument as follows:

```
mosquitto_passwd <password file> <username>
```

For example, if I want to change the password for the `sara` user and taking into account that the password file we created was called `passwd`, the command will be as follows:

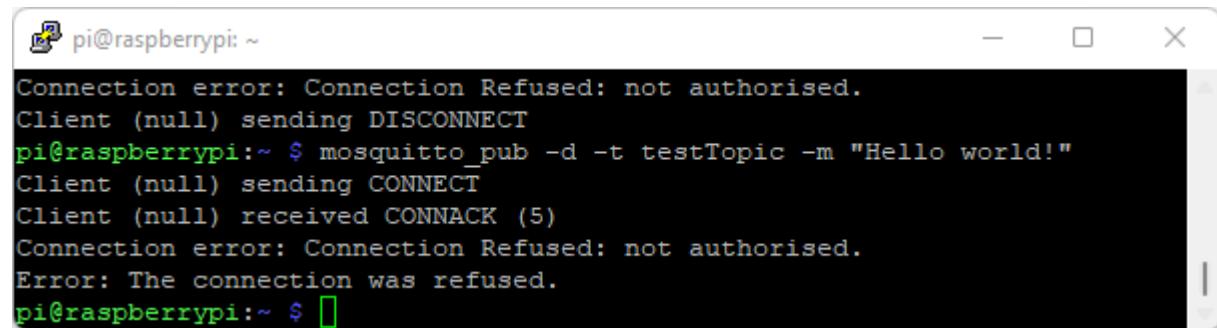
```
sudo mosquitto_passwd /etc/mosquitto/passwd sara
```

Testing MQTT Broker Installation (User and Password)

Now, if you try the commands, you've tested previously like the following:

```
mosquitto_pub -d -t testTopic -m "Hello world!"
```

You should get a message as follows “Connection Refused not authorised”.



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window contains the following text:

```
Connection error: Connection Refused: not authorised.  
Client (null) sending DISCONNECT  
pi@raspberrypi:~ $ mosquitto_pub -d -t testTopic -m "Hello world!"  
Client (null) sending CONNECT  
Client (null) received CONNACK (5)  
Connection error: Connection Refused: not authorised.  
Error: The connection was refused.  
pi@raspberrypi:~ $
```

You need to pass the user and password to the command, as follows:

```
mosquitto_pub -d -t testTopic -m "Hello world!" -u user -P pass
```

Replace **user** with your username and **pass** with your password.

This time, you should publish the message successfully.

```
Client (null) sending CONNECT  
Client (null) received CONNACK (0)  
Client (null) sending PUBLISH (d0, q0, r0, m1, 'testTopic', ... (12 bytes))  
Client (null) sending DISCONNECT
```

This means that the Mosquitto broker with user/password authentication is set up properly.

To subscribe to an MQTT topic, you also need to pass the username/password.

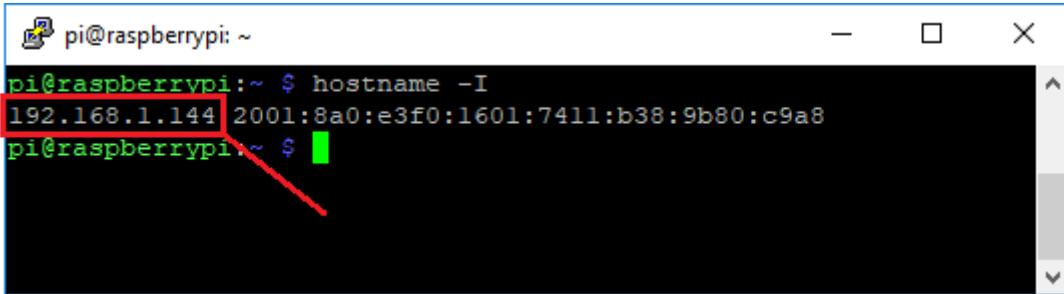
For example, enter the command:

```
mosquitto_sub -d -t testTopic -u user -P pass
```

Raspberry Pi IP Address

To use Mosquitto broker later in your projects, you need to know the Raspberry Pi IP address. To get your Raspberry Pi IP address, type the next command in your Pi Terminal window:

```
hostname -I
```



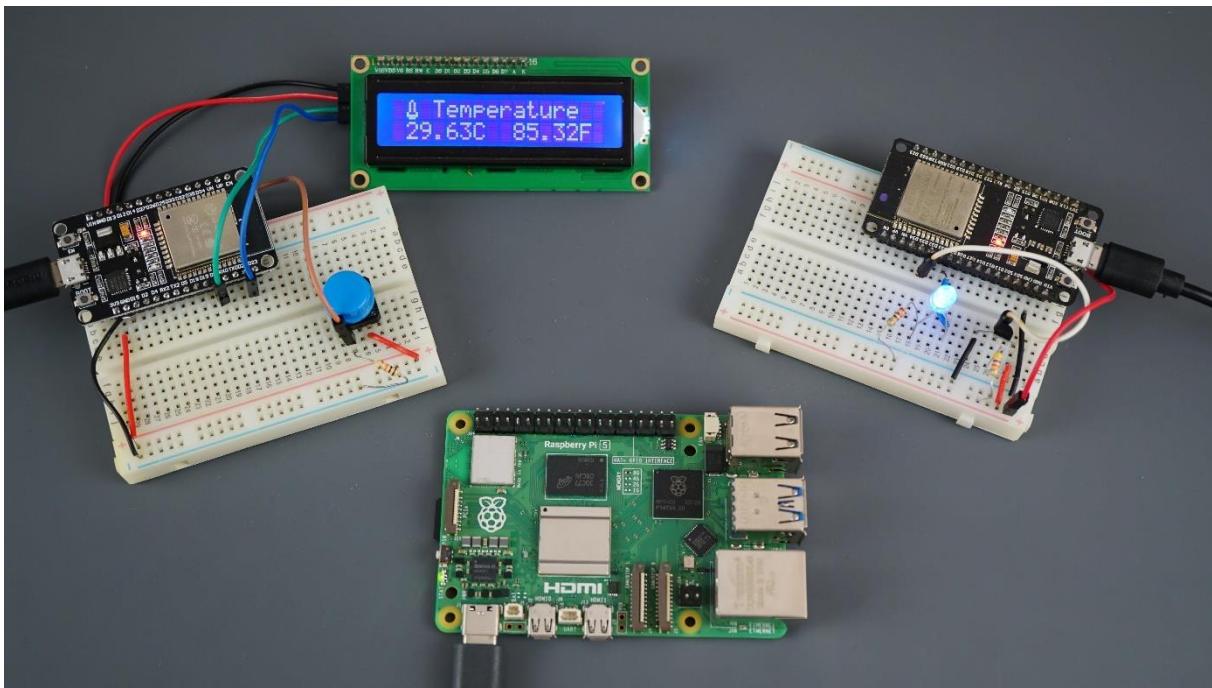
```
pi@raspberrypi: ~ $ hostname -I  
192.168.1.144 2001:8a0:e3f0:1601:7411:b38:9b80:c9a8  
pi@raspberrypi: ~ $
```

In this example, the Raspberry Pi IP address is **192.168.1.144**. Save your IP address. You'll need it in the next Units, so that the ESP32 is able to connect to the Mosquitto MQTT broker installed on your Raspberry Pi.

Wrapping Up

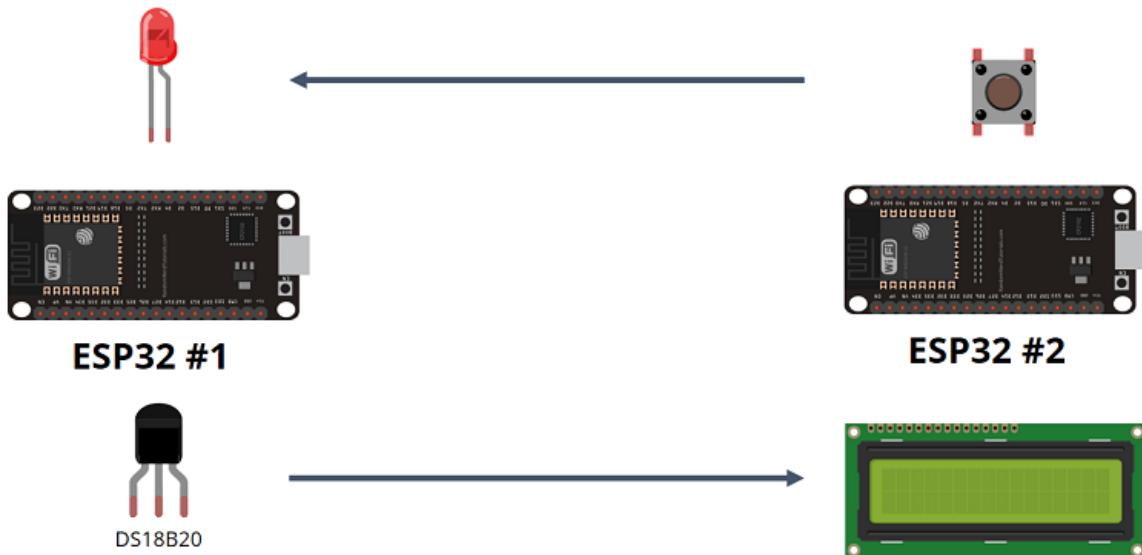
These were the steps to install the Mosquitto broker on a Raspberry Pi. In the next Unit, we'll set up two ESP32 boards as MQTT clients, and you'll see how everything works with practical examples.

9.3 - MQTT Project: MQTT Client ESP32#1



This Unit demonstrates how you can use MQTT to exchange data between two ESP32 boards. We're going to build a simple project to illustrate the most important concepts.

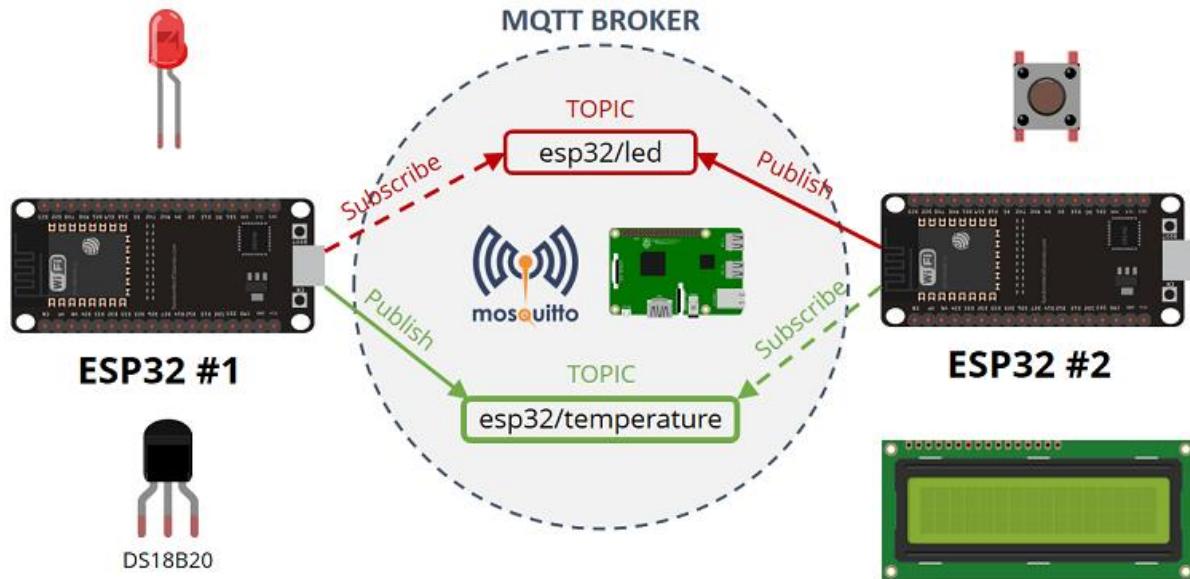
Here's a quick overview of the project. We'll have two ESP32 boards: ESP32 #1 and ESP32 #2:



- ESP32 #1 is connected to an LED and takes temperature readings with the DS18B20 sensor;

- ESP32 #2 is attached to a pushbutton that, when pressed, toggles the LED of the ESP32 #1;
- ESP32 #2 is connected to an I2C LCD to display temperature readings received from ESP32 #1.

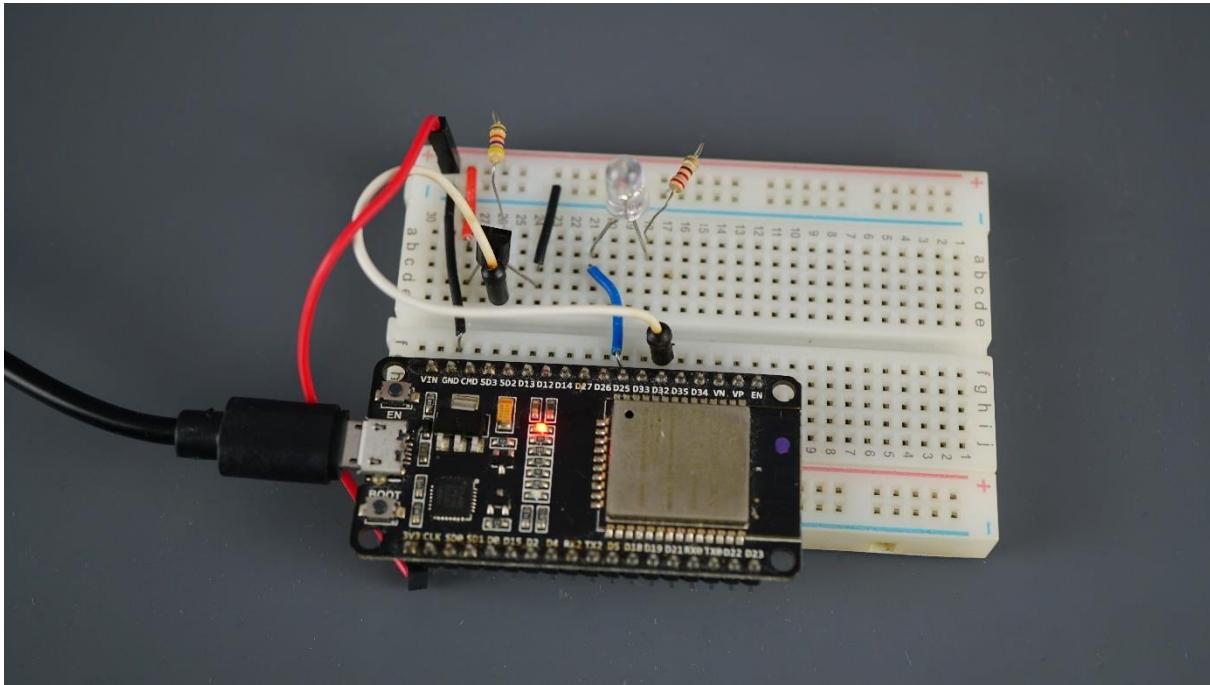
Here's the MQTT diagram of this setup.



- ESP32 #1 is subscribed to the topic **esp32/led** and publishes temperature readings on topic **esp32/temperature**.
- When you press the ESP32 #2 pushbutton, it publishes a message on the topic **esp32/led** to control the LED attached to ESP32 #1.
- ESP32 #2 is subscribed to **esp32/temperature** topic to receive temperature readings and displays them on the LCD.

We're using the Mosquitto broker installed on a Raspberry Pi to distribute the messages between the MQTT clients.

In this Unit, we're going to prepare ESP32 #1 (in the next Unit, you'll prepare ESP32 #2).



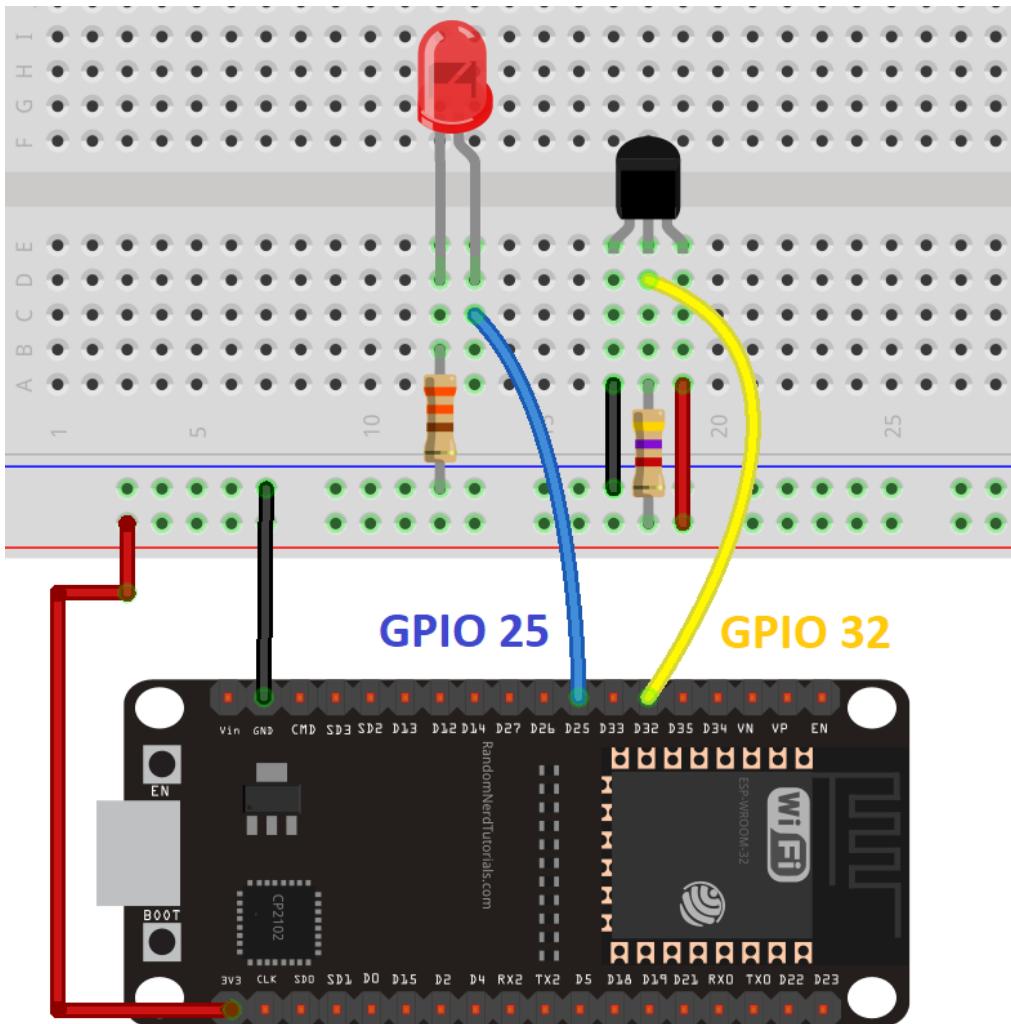
Wiring the Circuit

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [220 Ohm resistor \(or similar value\)](#)
- [DS18B20 temperature sensor](#)
- [4.7k Ohm resistor](#)
- [Jumper wires](#)
- [Breadboard](#)

Start by assembling the circuit by following the next schematic diagram:

- Connect an LED with a 220 Ohm (or similar) resistor to GPIO 25;
- Wire the DS18B20 sensor with a 4.7K pull-up resistor to GPIO 32.



Installing Libraries

Follow the next steps to install the required libraries in your Arduino IDE.

Installing the Async TCP Library

To use MQTT with the ESP32, you need the [Async TCP library](#). Follow these exact steps to install the library.

1. [Click here to download the Async TCP client library](#). You'll get a .zip folder in your **Downloads** folder.
2. In your Arduino IDE, go to **Sketch > Include Libraries > Add .ZIP Library** and select the library you just downloaded.

Installing the Async MQTT Client Library

To use MQTT with the ESP, you also need the Async MQTT client library. Follow the next steps to install it.

1. [Click here to download the Async MQTT client library](#). You should have a .zip folder in your **Downloads** folder
2. In your Arduino IDE, go to **Sketch > Include Libraries > Add .ZIP Library** and select the library you just downloaded.

Installing the One Wire Library and Dallas Temperature Library

You also need to install the [One Wire library by Paul Stoffregen](#) and the [Dallas Temperature library](#) to interface with the DS18B20 sensor.

To install these libraries, go to **Sketch > Include Libraries > Manage Libraries** and search for the libraries' names. Then, install the libraries.

Code

With the libraries installed, open your Arduino IDE and copy the code provided.

- [Click here to download the code.](#)

```
#include <WiFi.h>
extern "C" {
    #include "freertos/FreeRTOS.h"
    #include "freertos/timers.h"
}
#include <AsyncMqttClient.h>
#include <OneWire.h>
#include <DallasTemperature.h>

// Change the credentials below, so your ESP32 connects to your router
#define WIFI_SSID "REPLACE_WITH_YOUR_SSID"
#define WIFI_PASSWORD "REPLACE_WITH_YOUR_PASSWORD"

// Change the MQTT_HOST variable to your Raspberry Pi IP address,
// so it connects to your Mosquitto MQTT broker
#define MQTT_HOST IPAddress(192, 168, 1, XXX)
#define MQTT_PORT 1883

#define BROKER_USER "REPLACE_WITH_BROKER_USERNAME"
#define BROKER_PASS "REPLACE_WITH_BROKER_PASSWORD"

// Create objects to handle MQTT client
AsyncMqttClient mqttClient;
TimerHandle_t mqttReconnectTimer;
```

```

TimerHandle_t wifiReconnectTimer;

String temperatureString = "";      // Variable to hold the temperature reading
unsigned long previousMillis = 0;   // Stores last time temperature was published
const long interval = 5000;         // interval at which to publish sensor readings

const int ledPin = 25;             // GPIO where the LED is connected to
int ledState = LOW;               // the current state of the output pin

// GPIO where the DS18B20 is connected to
const int oneWireBus = 32;
// Setup a OneWire instance to communicate with any OneWire devices
OneWire oneWire(oneWireBus);
// Pass our oneWire reference to Dallas Temperature sensor
DallasTemperature sensors(&oneWire);

void connectToWifi() {
    Serial.println("Connecting to Wi-Fi...");
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}

void connectToMqtt() {
    Serial.println("Connecting to MQTT...");
    mqttClient.connect();
}

void WiFiEvent(WiFiEvent_t event) {
    Serial.printf("[WiFi-event] event: %d\n", event);
    switch(event) {
        case ARDUINO_EVENT_WIFI_STA_GOT_IP:
            Serial.println("WiFi connected");
            Serial.println("IP address: ");
            Serial.println(WiFi.localIP());
            connectToMqtt();
            break;
        case ARDUINO_EVENT_WIFI_STA_DISCONNECTED:
            Serial.println("WiFi lost connection");
            // ensure we don't reconnect to MQTT while reconnecting to Wi-Fi
            xTimerStop(mqttReconnectTimer, 0);
            xTimerStart(wifiReconnectTimer, 0);
            break;
    }
}

// Add more topics that want your ESP32 to be subscribed to
void onMqttConnect(bool sessionPresent) {
    Serial.println("Connected to MQTT.");
    Serial.print("Session present: ");
    Serial.println(sessionPresent);
    // ESP32 subscribed to esp32/led topic
    uint16_t packetIdSub = mqttClient.subscribe("esp32/led", 0);
    Serial.print("Subscribing at QoS 0, packetId: ");
    Serial.println(packetIdSub);
}

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
    Serial.println("Disconnected from MQTT.");
    if (WiFi.isConnected()) {
        xTimerStart(mqttReconnectTimer, 0);
    }
}

```

```

void onMqttSubscribe(uint16_t packetId, uint8_t qos) {
    Serial.println("Subscribe acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
    Serial.print(" qos: ");
    Serial.println(qos);
}

void onMqttUnsubscribe(uint16_t packetId) {
    Serial.println("Unsubscribe acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
}

void onMqttPublish(uint16_t packetId) {
    Serial.println("Publish acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
}

// You can modify this function to handle what happens when you receive
// a certain message in a specific topic
void onMqttMessage(char* topic, char* payload, AsyncMqttClientMessageProperties
                    properties, size_t len, size_t index, size_t total) {
    String messageTemp;
    for (int i = 0; i < len; i++) {
        //Serial.print((char)payload[i]);
        messageTemp += (char)payload[i];
    }
    // Check if the MQTT message was received on topic esp32/led
    if (strcmp(topic, "esp32/led") == 0) {
        // If the LED is off turn it on (and vice-versa)
        if (ledState == LOW) {
            ledState = HIGH;
        } else {
            ledState = LOW;
        }
        // Set the LED with the ledState of the variable
        digitalWrite(ledPin, ledState);
    }

    Serial.println("Publish received.");
    Serial.print(" message: ");
    Serial.println(messageTemp);
    Serial.print(" topic: ");
    Serial.println(topic);
    Serial.print(" qos: ");
    Serial.println(properties.qos);
    Serial.print(" dup: ");
    Serial.println(properties.dup);
    Serial.print(" retain: ");
    Serial.println(properties.retain);
    Serial.print(" len: ");
    Serial.println(len);
    Serial.print(" index: ");
    Serial.println(index);
    Serial.print(" total: ");
    Serial.println(total);
}

void setup() {

```

```

// Start the DS18B20 sensor
sensors.begin();
// Define LED as an OUTPUT and set it LOW
pinMode(ledPin, OUTPUT);
digitalWrite(ledPin, LOW);

Serial.begin(115200);

mqttReconnectTimer = xTimerCreate("mqttTimer", pdMS_TO_TICKS(2000), pdFALSE,
    (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToMqtt));
wifiReconnectTimer = xTimerCreate("wifiTimer", pdMS_TO_TICKS(2000), pdFALSE,
    (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToWifi));

WiFi.onEvent(WiFiEvent);

mqttClient.onConnect(onMqttConnect);
mqttClient.onDisconnect(onMqttDisconnect);
mqttClient.onSubscribe(onMqttSubscribe);
mqttClient.onUnsubscribe(onMqttUnsubscribe);
mqttClient.onMessage(onMqttMessage);
mqttClient.onPublish(onMqttPublish);
mqttClient.setServer(MQTT_HOST, MQTT_PORT);
mqttClient.setCredentials(BROKER_USER, BROKER_PASS);

connectToWifi();
}

void loop() {
    unsigned long currentMillis = millis();
    // Every X number of seconds (interval = 5 seconds)
    // it publishes a new MQTT message on topic esp32/temperature
    if (currentMillis - previousMillis >= interval) {
        // Save the last time a new reading was published
        previousMillis = currentMillis;
        // New temperature readings
        sensors.requestTemperatures();
        temperatureString = " " + String(sensors.getTempCByIndex(0)) + "C " +
            String(sensors.getTempFByIndex(0)) + "F";
        Serial.println(temperatureString);
        // Publish an MQTT message on topic esp32/temperature with Celsius and Fahrenheit temperature readings
        uint16_t packetIdPub2 = mqttClient.publish("esp32/temperature", 2,
            true, temperatureString.c_str());
        Serial.print("Publishing on topic esp32/temperature at QoS 2, packetId: ");
        Serial.println(packetIdPub2);
    }
}

```

To make the code work straight away, type your Wi-Fi SSID and password on the following lines.

```
#define WIFI_SSID "REPLACE_WITH_YOUR_SSID"
#define WIFI_PASSWORD "REPLACE_WITH_YOUR_PASSWORD"
```

You also need to enter the Mosquitto MQTT broker IP address. Go to the previous Unit to find your Raspberry Pi IP address.

```
#define MQTT_HOST IPAddress(192, 168, 1, XXX)
```

If you've set up your broker to use username and password, insert them on the following lines:

```
#define BROKER_USER "REPLACE_WITH_BROKER_USERNAME"  
#define BROKER_PASS "REPLACE_WITH_BROKER_PASSWORD"
```

You can upload the code as it is, and it will work. But we recommend continue reading this Unit to learn how it works.

How the Code Works

The following section imports all the required libraries.

```
#include <WiFi.h>  
extern "C" {  
    #include "freertos/FreeRTOS.h"  
    #include "freertos/timers.h"  
}  
#include <AsyncMqttClient.h>  
#include <OneWire.h>  
#include <DallasTemperature.h>
```

As said before, you need to include your SSID, password, and Raspberry Pi IP address.

```
// Change the credentials below, so your ESP32 connects to your router  
#define WIFI_SSID "REPLACE_WITH_YOUR_SSID"  
#define WIFI_PASSWORD "REPLACE_WITH_YOUR_PASSWORD"  
  
// Change the MQTT_HOST variable to your Raspberry Pi IP address,  
// so it connects to your Mosquitto MQTT broker  
#define MQTT_HOST IPAddress(192, 168, 1, XXX)  
#define MQTT_PORT 1883
```

If you've set up your broker to use username and password, insert them on the following lines:

```
#define BROKER_USER "REPLACE_WITH_BROKER_USERNAME"  
#define BROKER_PASS "REPLACE_WITH_BROKER_PASSWORD"
```

Create an object to handle the MQTT client and timers to reconnect to your MQTT broker and router when it disconnects.

```
// Create objects to handle MQTT client  
AsyncMqttClient mqttClient;  
TimerHandle_t mqttReconnectTimer;  
TimerHandle_t wifiReconnectTimer;
```

After that, declare some variables to hold the temperature and create auxiliary timer variables to publish readings every 5 seconds.

```
String temperatureString = "";      // Variable to hold the temperature reading
unsigned long previousMillis = 0;    // Stores last time temperature was published
const long interval = 5000;         // interval at which to publish sensor readings
```

Then, define the LED pin and store its initial state.

```
const int ledPin = 25;             // GPIO where the LED is connected to
int ledState = LOW;               // the current state of the output pin
```

Finally, define the DS18B20 sensor pin and create an instance of `oneWire` and `DallasTemperature` to refer to the sensor.

```
// GPIO where the DS18B20 is connected to
const int oneWireBus = 32;
// Setup a OneWire instance to communicate with any OneWire devices
OneWire oneWire(oneWireBus);
// Pass our OneWire reference to Dallas Temperature sensor
DallasTemperature sensors(&oneWire);
```

MQTT functions: connect to Wi-Fi, connect to MQTT, and Wi-Fi events

We haven't added any comments to the functions defined in the next code section. Those functions come with the `AsyncMqttClient` library. The function's names are pretty self-explanatory. For example, the `connectToWifi()` connects your ESP32 to your router:

```
void connectToWifi() {
  Serial.println("Connecting to Wi-Fi...");
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}
```

The `connectToMqtt()` connects your ESP32 to your MQTT broker:

```
void connectToMqtt() {
  Serial.println("Connecting to MQTT...");
  mqttClient.connect();
}
```

The `WiFiEvent()` function is responsible for handling the Wi-Fi events. For example, after a successful connection with the router and MQTT broker, it prints the ESP32 IP address. On the other hand, if the connection is lost, it starts a timer and tries to reconnect.

```
void WiFiEvent(WiFiEvent_t event) {
    Serial.printf("[WiFi-event] event: %d\n", event);
    switch(event) {
        case ARDUINO_EVENT_WIFI_STA_GOT_IP:
            Serial.println("WiFi connected");
            Serial.println("IP address: ");
            Serial.println(WiFi.localIP());
            connectToMqtt();
            break;
        case ARDUINO_EVENT_WIFI_STA_DISCONNECTED:
            Serial.println("WiFi lost connection");
            // ensure we don't reconnect to MQTT while reconnecting to Wi-Fi
            xTimerStop(mqttReconnectTimer, 0);
            xTimerStart(wifiReconnectTimer, 0);
            break;
    }
}
```

Subscribe to an MQTT topic

The `onMqttConnect()` function is responsible for subscribing your ESP32 to topics. You can modify this function and add more topics that you want your ESP32 to be subscribed to. In this case, the ESP32 is only subscribed to **esp32/led** topic.

```
// Add more topics that want your ESP32 to be subscribed to
void onMqttConnect(bool sessionPresent) {
    Serial.println("Connected to MQTT.");
    Serial.print("Session present: ");
    Serial.println(sessionPresent);
    // ESP32 subscribed to esp32/led topic
    uint16_t packetIdSub = mqttClient.subscribe("esp32/led", 0);
    Serial.print("Subscribing at QoS 0, packetId: ");
    Serial.println(packetIdSub);
}
```

The important part of this snippet is the following line that subscribes to an MQTT topic using the `subscribe()` method:

```
uint16_t packetIdSub = mqttClient.subscribe("esp32/led", 0);
```

This method accepts as arguments the MQTT topic you want the ESP32 to be subscribed to, and the Quality of Service (QoS). You can [read this article](#) for more information about MQTT QoS.

MQTT functions: disconnect, subscribe, unsubscribe, and publish

If the ESP32 loses connection with the MQTT broker, it prints that message in the serial monitor.

```

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
    Serial.println("Disconnected from MQTT.");
    if (WiFi.isConnected()) {
        xTimerStart(mqttReconnectTimer, 0);
    }
}

```

When the ESP32 subscribes to an MQTT topic, it prints the packet id and quality of service (QoS).

```

void onMqttSubscribe(uint16_t packetId, uint8_t qos) {
    Serial.println("Subscribe acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
    Serial.print(" qos: ");
    Serial.println(qos);
}

```

If you unsubscribe from a topic, it also prints a message with some information about that.

```

void onMqttUnsubscribe(uint16_t packetId) {
    Serial.println("Unsubscribe acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
}

```

And when you publish a message to an MQTT topic, it prints the packet id in the Serial Monitor.

```

void onMqttPublish(uint16_t packetId) {
    Serial.println("Publish acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
}

```

Receiving MQTT messages

When a message is received on a topic that the ESP32 is subscribed to, in this case the **esp32/led** topic, it executes the `onMqttMessage()` function. In this function you should add what happens when a message is received on a specific topic.

```

// You can modify this function to handle what happens when you receive a certain
// message in a specific topic
void onMqttMessage(char* topic, char* payload, AsyncMqttClientMessageProperties
                    properties, size_t len, size_t index, size_t total) {
    String messageTemp;
    for (int i = 0; i < len; i++) {
        //Serial.print((char)payload[i]);
        messageTemp += (char)payload[i];
    }
}

```

```

// Check if the MQTT message was received on topic esp32/led
if (strcmp(topic, "esp32/led") == 0) {
    // If the LED is off turn it on (and vice-versa)
    if (ledState == LOW) {
        ledState = HIGH;
    } else {
        ledState = LOW;
    }
    // Set the LED with the ledState of the variable
    digitalWrite(ledPin, ledState);
}

Serial.println("Publish received.");
Serial.print(" message: ");
Serial.println(messageTemp);
Serial.print(" topic: ");
Serial.println(topic);
Serial.print(" qos: ");
Serial.println(properties.qos);
Serial.print(" dup: ");
Serial.println(properties.dup);
Serial.print(" retain: ");
Serial.println(properties.retain);
Serial.print(" len: ");
Serial.println(len);
Serial.print(" index: ");
Serial.println(index);
Serial.print(" total: ");
Serial.println(total);
}

```

In this previous snippet, the following **if** statement, checks if a new message was published on the **esp32/led** topic.

```
if (strcmp(topic, "esp32/led") == 0) {
```

Then, you can add your logic inside that **if**

statement to make the ESP32 do something. In this case, we're toggling the LED every time we receive a message on that topic.

```

if (ledState == LOW) {
    ledState = HIGH;
} else {
    ledState = LOW;
}
// Set the LED with the ledState of the variable
digitalWrite(ledPin, ledState);

```

Basically, all these functions that we've just mentioned are callback functions. So, they are executed asynchronously.

setup()

Now, let's proceed to the `setup()`. Start the DS18B20 sensor, define the LED as an `OUTPUT`, set it to `LOW` and start the serial communication.

```
// Start the DS18B20 sensor
sensors.begin();
// Define LED as an OUTPUT and set it LOW
pinMode(ledPin, OUTPUT);
digitalWrite(ledPin, LOW);

Serial.begin(115200);
```

The next two lines create timers that will allow both the MQTT broker and Wi-Fi connection to reconnect, in case the connection is lost.

```
mqttReconnectTimer = xTimerCreate("mqttTimer", pdMS_TO_TICKS(2000), pdFALSE,
                                    (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToMqtt));
wifiReconnectTimer = xTimerCreate("wifiTimer", pdMS_TO_TICKS(2000), pdFALSE,
                                    (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToWifi));
```

The following line assigns a callback function, so when the ESP32 connects to your Wi-Fi, it will execute the `WiFiEvent` function to print the details described earlier.

```
WiFi.onEvent(WiFiEvent);
```

Finally, assign all the callback functions. This means that these functions will be executed automatically when needed. For example, when the ESP32 connects to the broker, it automatically calls the `onMqttConnect()` function, and so on.

```
mqttClient.onConnect(onMqttConnect);
mqttClient.onDisconnect(onMqttDisconnect);
mqttClient.onSubscribe(onMqttSubscribe);
mqttClient.onUnsubscribe(onMqttUnsubscribe);
mqttClient.onMessage(onMqttMessage);
mqttClient.onPublish(onMqttPublish);
mqttClient.setServer(MQTT_HOST, MQTT_PORT);
mqttClient.setCredentials(BROKER_USER, BROKER_PASS);
```

If you're not using username and password with your broker, you just need to remove the following line:

```
mqttClient.setCredentials(BROKER_USER, BROKER_PASS);
```

loop()

In the `loop()`, you create a timer that will allow you to publish new temperature readings in the **esp32/temperature** topic every 5 seconds.

```
unsigned long currentMillis = millis();
// Every X number of seconds (interval = 5 seconds)
// it publishes a new MQTT message on topic esp32/temperature
if (currentMillis - previousMillis >= interval) {
    // Save the last time a new reading was published
    previousMillis = currentMillis;
    // New temperature readings
    sensors.requestTemperatures();
    temperatureString = " " + String(sensors.getTempCByIndex(0)) + "C " +
                        String(sensors.getTempFByIndex(0)) + "F";
    Serial.println(temperatureString);
    // Publish an MQTT message on topic esp32/temperature with
    // Celsius and Fahrenheit temperature readings
    uint16_t packetIdPub2 = mqttClient.publish("esp32/temperature", 2,
                                                true, temperatureString.c_str());
    Serial.print("Publishing on topic esp32/temperature at QoS 2, packetId: ");
    Serial.println(packetIdPub2);
}
```

Publishing/Subscribing to more topics

This is a basic example but illustrates how MQTT works with publish and subscribe.

If you would like to publish more readings on different topics, you can multiply these next three lines in the `loop()`. Basically, use the `publish()` method to publish data on a topic.

```
uint16_t packetIdPub2 = mqttClient.publish("esp32/temperature", 2,
                                            true, temperatureString.c_str());
Serial.print("Publishing on topic esp32/temperature at QoS 2, packetId: ");
Serial.println(packetIdPub2);
```

On the other hand, if you want to subscribe to more topics, go to the `onMqttConnect()` function, duplicate the following line and replace the topic with another topic that you want to be subscribed to.

```
uint16_t packetIdSub = mqttClient.subscribe("esp32/led", 0);
```

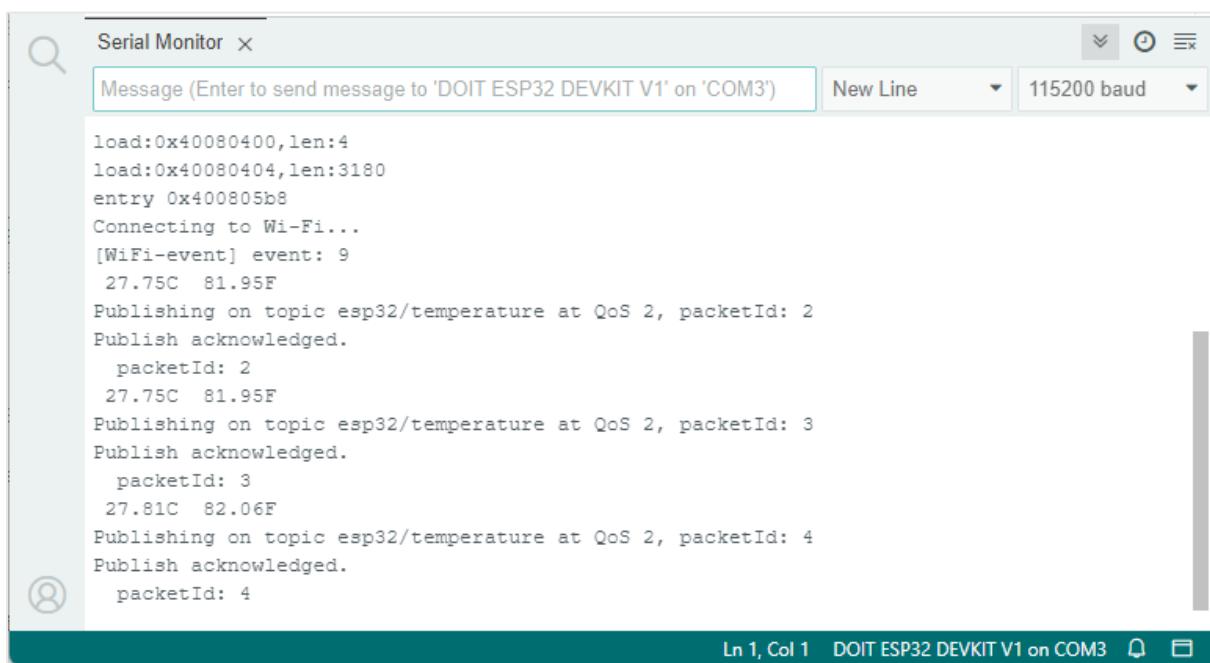
Finally, in the `onMqttMessage()` function, you can add logic to determine what happens when you receive a message in a specific topic.

Uploading the code

With your Raspberry Pi powered on and running the Mosquitto MQTT broker, upload the code to your ESP32.

Note: complete the previous Unit if you haven't prepared the Mosquitto MQTT broker.

Open the serial monitor at the 115200 baud rate and check if your ESP32 is successfully connected to your router and MQTT broker and that it starts publishing MQTT messages.



The screenshot shows the Arduino Serial Monitor window titled "Serial Monitor". The message input field contains "Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'COM3')". The settings dropdown shows "115200 baud". The main text area displays the following log output:

```
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Connecting to Wi-Fi...
[WiFi-event] event: 9
27.75C 81.95F
Publishing on topic esp32/temperature at QoS 2, packetId: 2
Publish acknowledged.
  packetId: 2
  27.75C 81.95F
Publishing on topic esp32/temperature at QoS 2, packetId: 3
Publish acknowledged.
  packetId: 3
  27.81C 82.06F
Publishing on topic esp32/temperature at QoS 2, packetId: 4
Publish acknowledged.
  packetId: 4
```

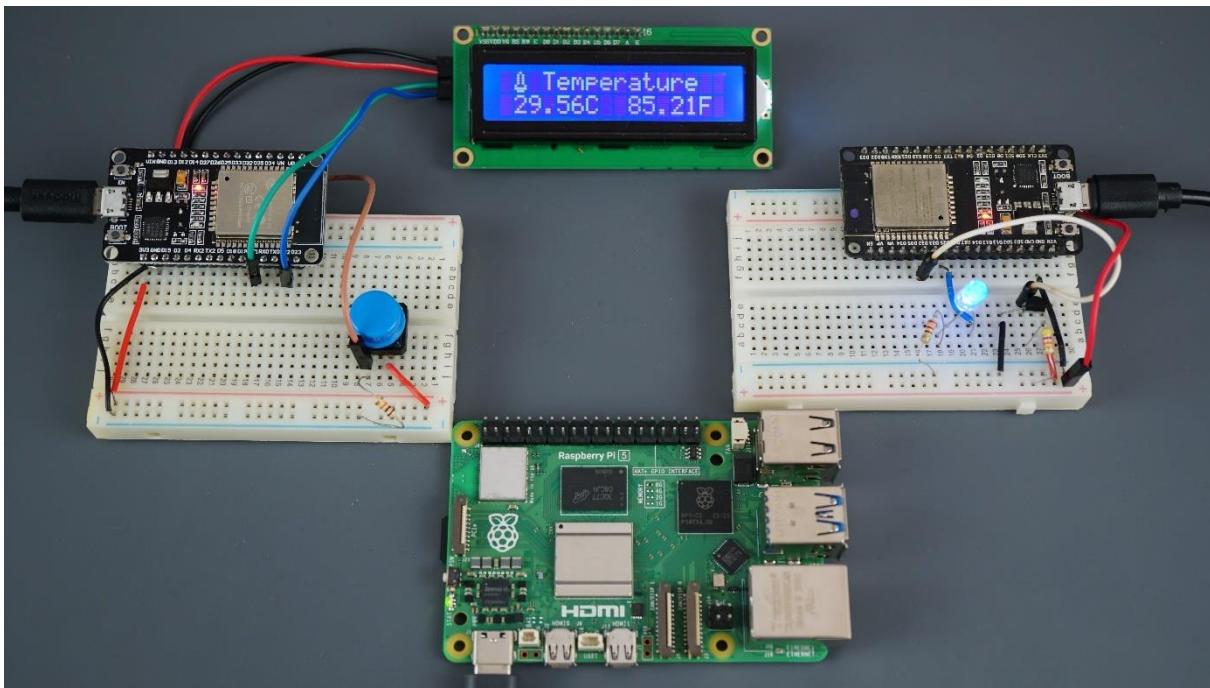
At the bottom, the status bar shows "Ln 1, Col 1 DOIT ESP32 DEVKIT V1 on COM3" and icons for volume, brightness, and close.

As you can see, it's working as expected.

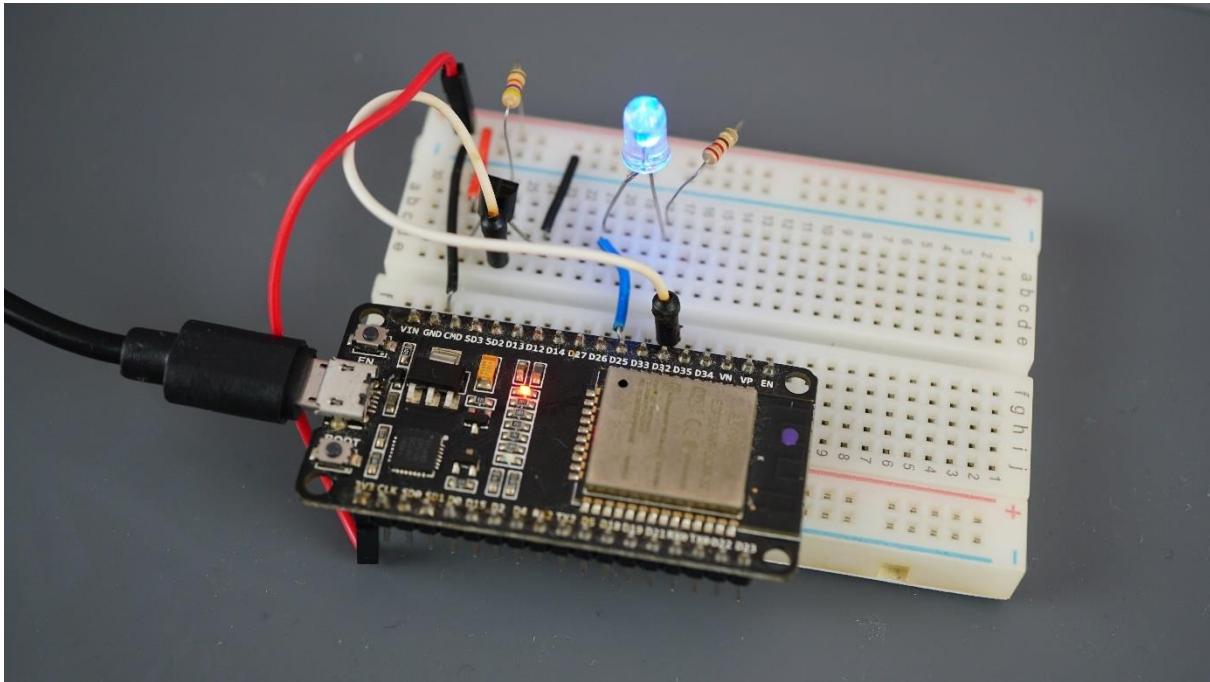
Continue To The Next Unit...

Go to the next Unit to prepare ESP32 #2 and continue this project.

9.4 - MQTT Project : MQTT Client ESP32 #2



This Unit is part 2 of the ESP32 MQTT project example. If you've followed the previous Unit, your ESP32 #1 is ready (shown in the picture below).



In this Unit, you're going to prepare ESP32 #2. To recap:

- ESP32 #2 receives the temperature readings from ESP32 #1 and displays them on an LCD;
- ESP32 #2 has a pushbutton that controls the LED of the ESP32 #1.

Wiring the Circuit

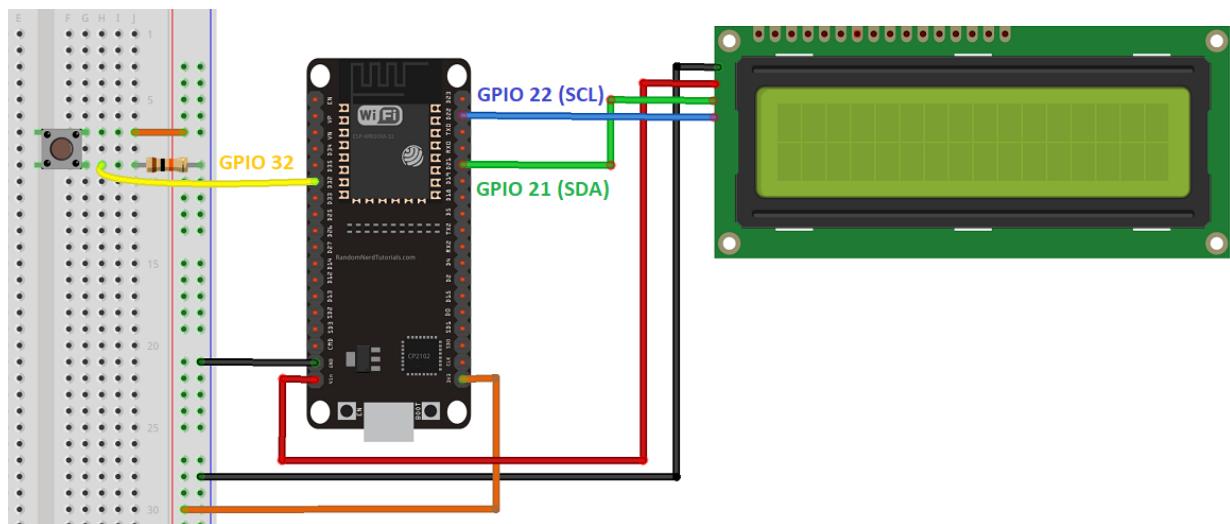
Here's a list of the parts you need to build the circuit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [Pushbutton](#)
- [10k Ohm resistor](#)
- [16x2 I2C LCD](#)
- [Jumper wires](#)
- [Breadboard](#)

Assemble the circuit:

- Wire a pushbutton to the ESP32: one lead to 3.3V and the other lead connected to GPIO 32 using a 10k Ohm pull-down resistor.
- Wire the I2C LCD: connect SDA to GPIO 21 and SCL to GPIO 22. The LCD operates at 5V, so connect it to Vin and GND.

Use the following schematic diagram as a reference.



Installing Libraries

After having your circuit ready, it's time to install the necessary libraries in your Arduino IDE.

If you've followed the previous Unit, you already have the [Async TCP library](#) and [Async MQTT Client library](#) installed. So, you only need to install the [Liquid Crystal I2C library](#). Follow the next steps to install that library.

Installing the Liquid Crystal I2C Library

1. [Click here to download the LiquidCrystal I2C library](#). You should have a .zip folder in your **Downloads** folder
2. In your Arduino IDE, go to Sketch > Include Library > Add ZIP. Library... and select the library you just downloaded.
3. Finally, restart your Arduino IDE

Note: to learn more about using the I2C LCD with the ESP32, you can read the following tutorial: [How to Use I2C LCD with ESP32 on Arduino IDE](#).

Code

After installing all required libraries, open your Arduino IDE and copy the code provided.

- [Click here to download the code.](#)

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

extern "C" {
    #include "freertos/FreeRTOS.h"
    #include "freertos/timers.h"
}

#include <AsyncMqttClient.h>
#include <LiquidCrystal_I2C.h>

// Change the credentials below, so your ESP32 connects to your router
#define WIFI_SSID "REPLACE_WITH_YOUR_SSID"
#define WIFI_PASSWORD "REPLACE_WITH_YOUR_PASSWORD"

// Change the MQTT_HOST variable to your Raspberry Pi IP address,
// so it connects to your Mosquitto MQTT broker
#define MQTT_HOST IPAddress(XXX, XXX, X, XXX) //MQTT BROKER IP ADDRESS
/*for example:
#define MQTT_HOST IPAddress(192, 168, 1, 106)*/
#define MQTT_PORT 1883

#define BROKER_USER "REPLACE_WITH_BROKER_USERNAME"
#define BROKER_PASS "REPLACE_WITH_BROKER_PASSWORD"
```

```

// Create objects to handle MQTT client
AsyncMqttClient mqttClient;
TimerHandle_t mqttReconnectTimer;
TimerHandle_t wifiReconnectTimer;

// Set the LCD number of columns and rows
const int lcdColumns = 16;
const int lcdRows = 2;

// Set LCD address, number of columns and rows
// if you don't know your display address, run an I2C scanner sketch
LiquidCrystal_I2C lcd(0x27, lcdColumns, lcdRows);

// Thermometer icon
byte thermometerIcon[8] = {
    B00100,
    B01010,
    B01010,
    B01010,
    B01010,
    B10001,
    B11111,
    B01110
};

// Define GPIO where the pushbutton is connected to
const int buttonPin = 32;
int buttonState;           // current reading from the input pin (pushbutton)
int lastButtonState = LOW; // previous reading from the input pin (pushbutton)
unsigned long lastDebounceTime = 0; // the last time the output pin was toggled
unsigned long debounceDelay = 50; // the debounce time; increase if the output flickers

void connectToWifi() {
    Serial.println("Connecting to Wi-Fi...");
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}

void connectToMqtt() {
    Serial.println("Connecting to MQTT...");
    mqttClient.connect();
}

void WiFiEvent(WiFiEvent_t event) {
    Serial.printf("[WiFi-event] event: %d\n", event);
    switch(event) {
        case ARDUINO_EVENT_WIFI_STA_GOT_IP:
            Serial.println("WiFi connected");
            Serial.println("IP address: ");
            Serial.println(WiFi.localIP());
            connectToMqtt();
            break;
        case ARDUINO_EVENT_WIFI_STA_DISCONNECTED:
            Serial.println("WiFi lost connection");
            // ensure we don't reconnect to MQTT while reconnecting to Wi-Fi
            xTimerStop(mqttReconnectTimer, 0);
            xTimerStart(wifiReconnectTimer, 0);
            break;
    }
}

// Add more topics that want your ESP32 to be subscribed to

```

```

void onMqttConnect(bool sessionPresent) {
    Serial.println("Connected to MQTT.");
    Serial.print("Session present: ");
    Serial.println(sessionPresent);
    uint16_t packetIdSub = mqttClient.subscribe("esp32/temperature", 0);
    Serial.print("Subscribing at QoS 0, packetId: ");
    Serial.println(packetIdSub);
}

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
    Serial.println("Disconnected from MQTT.");
    if (WiFi.isConnected()) {
        xTimerStart(mqttReconnectTimer, 0);
    }
}

void onMqttSubscribe(uint16_t packetId, uint8_t qos) {
    Serial.println("Subscribe acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
    Serial.print(" qos: ");
    Serial.println(qos);
}

void onMqttUnsubscribe(uint16_t packetId) {
    Serial.println("Unsubscribe acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
}

void onMqttPublish(uint16_t packetId) {
    Serial.println("Publish acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
}

// You can modify this function to handle what happens when you receive a
// certain message in a specific topic
void onMqttMessage(char* topic, char* payload, AsyncMqttClientMessageProperties
    properties, size_t len, size_t index, size_t total) {
    String messageTemp;
    for (int i = 0; i < len; i++) {
        //Serial.print((char)payload[i]);
        messageTemp += (char)payload[i];
    }
    if (strcmp(topic, "esp32/temperature") == 0) {
        Serial.println("==> Updating LCD... ==<");
        Wire.setClock(10000);
        // Clears row, better than the slow lcd.clear()
        lcd.print(std::string(" ", lcdColumns).c_str());
        lcd.setCursor(0, 1);
        lcd.print(messageTemp);
    }

    Serial.println("Publish received.");
    Serial.print(" message: ");
    Serial.println(messageTemp);
    Serial.print(" topic: ");
    Serial.println(topic);
    Serial.print(" qos: ");
    Serial.println(properties.qos);
}

```

```

Serial.print(" dup: ");
Serial.println(properties.dup);
Serial.print(" retain: ");
Serial.println(properties.retain);
Serial.print(" len: ");
Serial.println(len);
Serial.print(" index: ");
Serial.println(index);
Serial.print(" total: ");
Serial.println(total);
}

void setup() {
    // Initialize LCD
    lcd.init();
    // Turn on LCD backlight
    lcd.backlight();
    // Create thermometer icon
    lcd.createChar(0, thermometerIcon);
    lcd.setCursor(1, 0);
    lcd.write(0);
    // Print header
    lcd.print(" Temperature");

    // Define buttonPin as an INPUT
    pinMode(buttonPin, INPUT);

    Serial.begin(115200);

    mqttReconnectTimer = xTimerCreate("mqttTimer", pdMS_TO_TICKS(2000), pdFALSE,
        (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToMqtt));
    wifiReconnectTimer = xTimerCreate("wifiTimer", pdMS_TO_TICKS(2000), pdFALSE,
        (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToWifi));

    WiFi.onEvent(WiFiEvent);

    mqttClient.onConnect(onMqttConnect);
    mqttClient.onDisconnect(onMqttDisconnect);
    mqttClient.onSubscribe(onMqttSubscribe);
    mqttClient.onUnsubscribe(onMqttUnsubscribe);
    mqttClient.onMessage(onMqttMessage);
    mqttClient.onPublish(onMqttPublish);
    mqttClient.setServer(MQTT_HOST, MQTT_PORT);
    mqttClient.setCredentials(BROKER_USER, BROKER_PASS);
    mqttClient.setClientId("ESP3222");

    connectToWifi();
}

void loop() {
    // Read the state of the pushbutton and save it in a local variable
    int reading = digitalRead(buttonPin);

    // If the pushbutton state changed (due to noise or pressing it), reset the timer
    if (reading != lastButtonState) {
        // Reset the debouncing timer
        lastDebounceTime = millis();
    }

    // If the button state has changed, after the debounce time
    if ((millis() - lastDebounceTime) > debounceDelay) {

```

```

// And if the current reading is different than the current buttonState
if (reading != buttonState) {
    buttonState = reading;
    // Publish an MQTT message on topic esp32/led to toggle the LED (turn the LED on or off)
    if (buttonState == HIGH) {
        mqttClient.publish("esp32/led", 0, true, "toggle");
        Serial.println("Publishing on topic esp32/led topic at QoS 0");
    }
}
// Save the reading. Next time through the loop, it'll be the lastButtonState
lastButtonState = reading;
}

```

Type your SSID, password, and MQTT broker IP address, broker username and password and the code will work straight away.

```

// Change the credentials below, so your ESP32 connects to your router
#define WIFI_SSID "REPLACE_WITH_YOUR_SSID"
#define WIFI_PASSWORD "REPLACE_WITH_YOUR_PASSWORD"

// Change the MQTT_HOST variable to your Raspberry Pi IP address,
// so it connects to your Mosquitto MQTT broker
#define MQTT_HOST IPAddress(XXX, XXX, X, XXX) //MQTT BROKER IP ADDRESS
/*for example:
#define MQTT_HOST IPAddress(192, 168, 1, 106)*/
#define MQTT_PORT 1883

#define BROKER_USER "REPLACE_WITH_BROKER_USERNAME"
#define BROKER_PASS "REPLACE_WITH_BROKER_PASSWORD"

```

Continue reading to learn how the code works.

How Does the Code Work?

We'll skip most code sections because they were already explained in the previous Unit.

Define the LCD number of columns and rows. In this case, we're using a 16x2 character LCD.

```

// Set the LCD number of columns and rows
const int lcdColumns = 16;
const int lcdRows = 2;

```

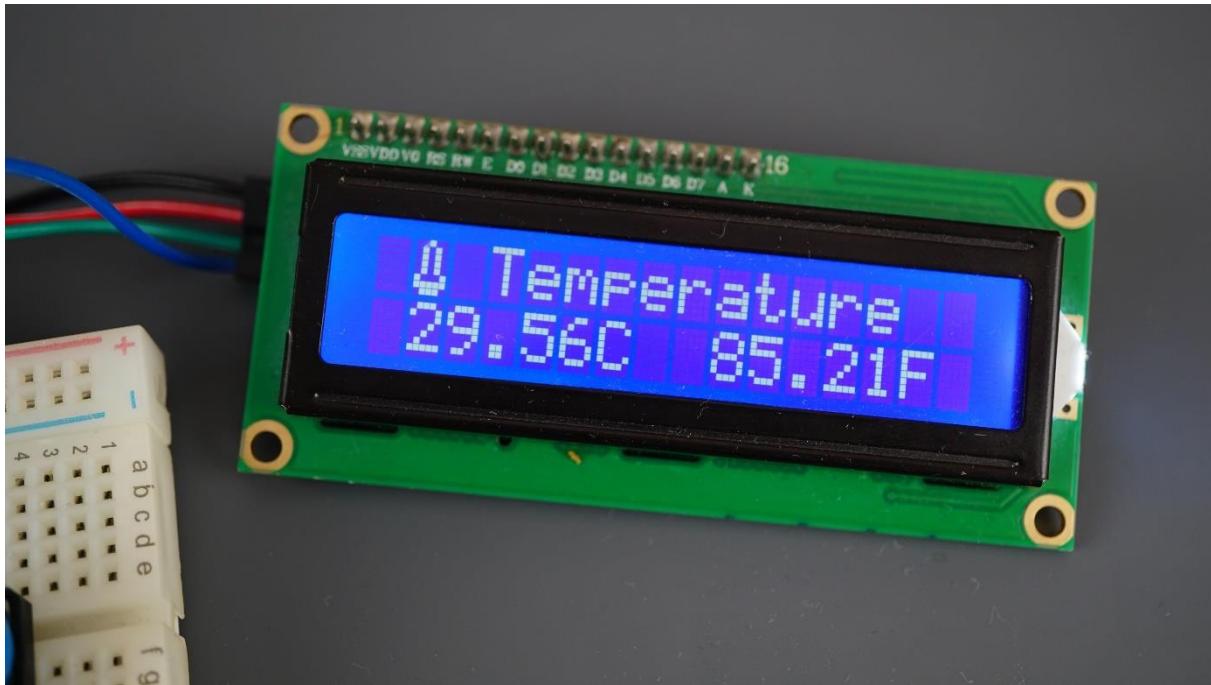
The following byte array is used to display a thermometer icon on the LCD.

```

// Thermometer icon
byte thermometerIcon[8] = {
    B00100,
    B01010,
    B01010,

```

```
B01010,  
B01010,  
B10001,  
B11111,  
B01110  
};
```



Define the button Pin, a variable to store the current button state, last button state and declare auxiliary variables to create a debounce timer to avoid false pushbutton presses.

```
// Define GPIO where the pushbutton is connected to  
const int buttonPin = 32;  
int buttonState; // current reading from the input pin (pushbutton)  
int lastButtonState = LOW; // previous reading from the input pin (pushbutton)  
unsigned long lastDebounceTime = 0; // the last time the output pin was toggled  
unsigned long debounceDelay = 50; // the debounce time; increase if the output flickers
```

MQTT functions

The `connectToWiFi()`, `connectToMQTT()` and `WiFiEvent()` functions were already explained in the previous Unit.

The `onMqttConnect()` function is where you add the topics you want your ESP32 to be subscribed to.

```
// Add more topics that want your ESP32 to be subscribed to  
void onMqttConnect(bool sessionPresent) {  
    Serial.println("Connected to MQTT.");  
    Serial.print("Session present: ");  
    Serial.println(sessionPresent);  
    uint16_t packetIdSub = mqttClient.subscribe("esp32/temperature", 0);
```

```
Serial.print("Subscribing at QoS 0, packetId: ");
Serial.println(packetIdSub);
}
```

In this case, the ESP32 is only subscribed to the **esp32/temperature** topic, but you can change the `onMQTTConnect()` function to subscribe to more topics.

In the `onMQTTMessage()` is where you implement what happens when you receive a message on a subscribed topic. You can create more `if` statements to check other topics, or to check the message content.

```
void onMQTTMessage(char* topic, char* payload, AsyncMQTTClientMessageProperties properties, size_t len, size_t index, size_t total) {
    String messageTemp;
    for (int i = 0; i < len; i++) {
        //Serial.print((char)payload[i]);
        messageTemp += (char)payload[i];
    }
    if (strcmp(topic, "esp32/temperature") == 0) {
        Serial.println("== Updating LCD... ==");
        Wire.setClock(10000);
        // Clears row, better than the slow lcd.clear()
        lcd.print(std::string(" ", lcdColumns).c_str());
        lcd.setCursor(0, 1);
        lcd.print(messageTemp);
    }
    ...
}
```

In this case, when we receive a message in the **esp32/temperature** topic, we display the message in the LCD.

setup()

In the `setup()`, we start the LCD, turn on the back light and create the thermometer icon, so we can display it on the LCD.

```
// Initialize LCD
lcd.init();
// Turn on LCD backlight
lcd.backlight();
// Create thermometer icon
lcd.createChar(0, thermometerIcon);
```

Set the `buttonPin` as an INPUT.

```
// Define buttonPin as an INPUT
pinMode(buttonPin, INPUT);
```

Then, create the reconnect timers:

```
mqttReconnectTimer = xTimerCreate("mqttTimer", pdMS_TO_TICKS(2000), pdFALSE,
    (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToMqtt));
wifiReconnectTimer = xTimerCreate("wifiTimer", pdMS_TO_TICKS(2000), pdFALSE,
    (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToWifi));
```

And set all the callback functions for the Wi-Fi and MQTT events.

```
WiFi.onEvent(WiFiEvent);

mqttClient.onConnect(onMqttConnect);
mqttClient.onDisconnect(onMqttDisconnect);
mqttClient.onSubscribe(onMqttSubscribe);
mqttClient.onUnsubscribe(onMqttUnsubscribe);
mqttClient.onMessage(onMqttMessage);
mqttClient.onPublish(onMqttPublish);
mqttClient.setServer(MQTT_HOST, MQTT_PORT);
mqttClient.setCredentials(BROKER_USER, BROKER_PASS);
```

If you're not using a username and password with the broker, remove the following line:

```
mqttClient.setCredentials(BROKER_USER, BROKER_PASS);
```

loop()

In the `loop()`, we publish an MQTT message on the **esp32/led** topic when the pushbutton is pressed.

We're using the button Debounce example code that ensures that you don't get false pushbutton presses.

When the button is pressed, the following if statement is `true` and the message "toggle" is published in the **esp32/led** topic.

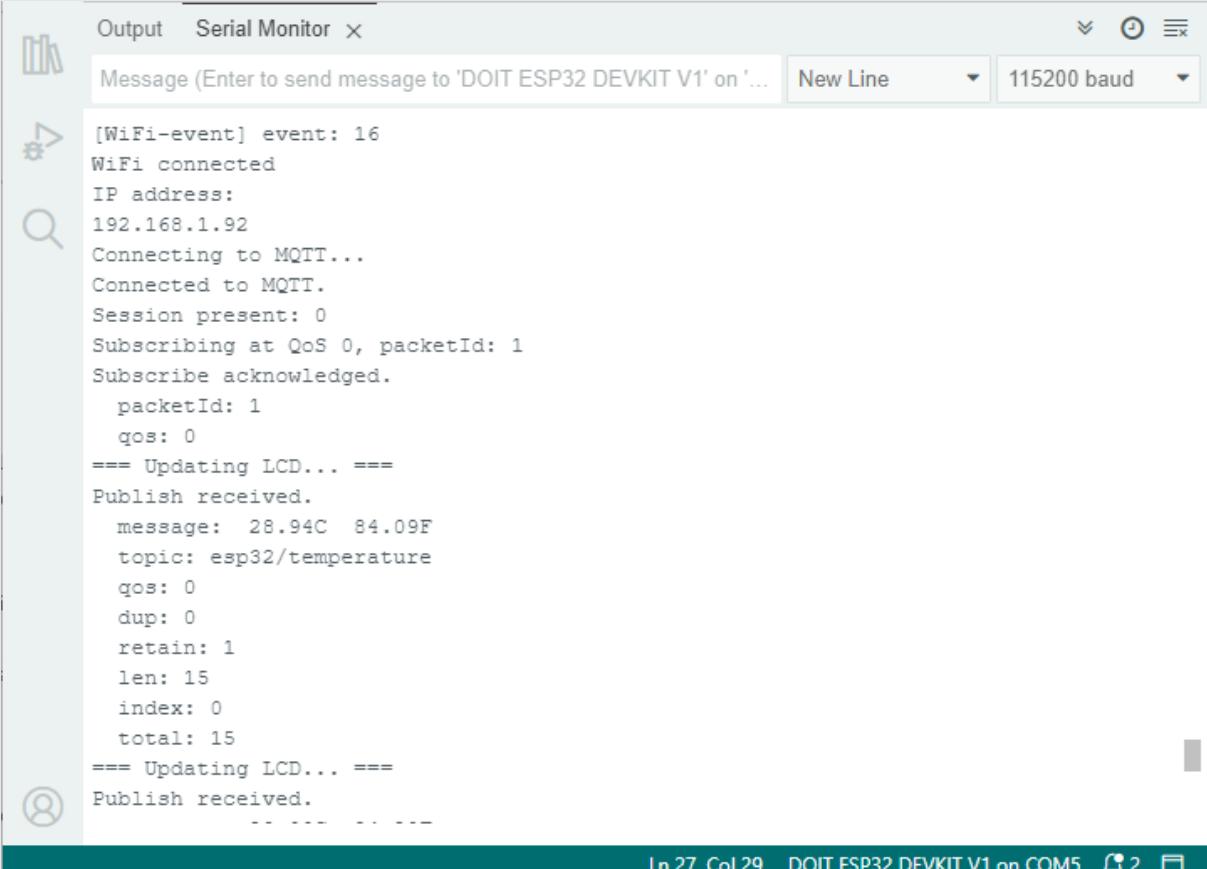
```
if (buttonState == HIGH) {
    mqttClient.publish("esp32/led", 0, true, "toggle");
    Serial.println("Publishing on topic esp32/led topic at QoS 0");
}
```

That's it for the code explanation. You can upload the code to your ESP32.

Demonstration

With your Raspberry Pi powered on and running the Mosquitto MQTT broker, open the serial monitor at the 115200 baud rate.

Check if your ESP32 is being successfully connected to your router and MQTT broker. As you can see in the following figure, it's working properly. It is receiving the MQTT messages from the other ESP32 with temperature and humidity.

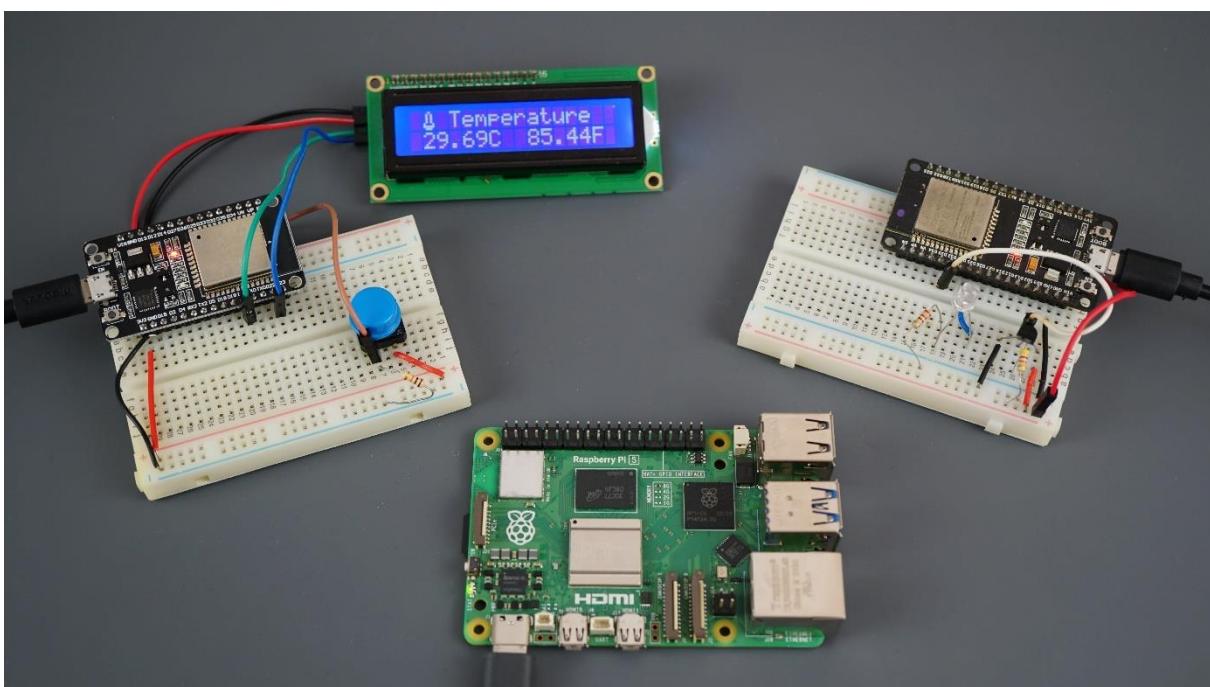


The screenshot shows the Arduino Serial Monitor window. The title bar says "Output" and "Serial Monitor". The message area displays the following log:

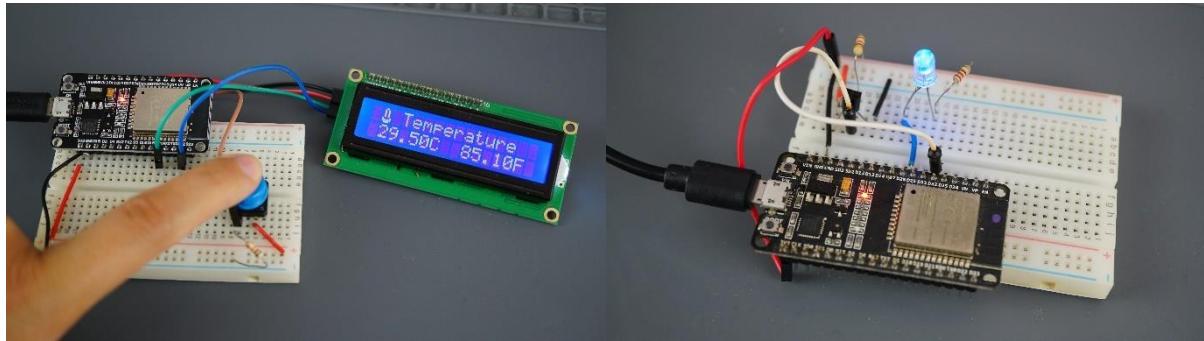
```
[WiFi-event] event: 16
WiFi connected
IP address:
192.168.1.92
Connecting to MQTT...
Connected to MQTT.
Session present: 0
Subscribing at QoS 0, packetId: 1
Subscribe acknowledged.
  packetId: 1
  qos: 0
== Updating LCD... ==
Publish received.
  message: 28.94C 84.09F
  topic: esp32/temperature
  qos: 0
  dup: 0
  retain: 1
  len: 15
  index: 0
  total: 15
== Updating LCD... ==
Publish received.
  - - - - -
```

The status bar at the bottom indicates "Ln 27, Col 29 DOIT ESP32 DEVKIT V1 on COM5" and shows a signal strength icon with "2" and a refresh icon.

It should be displaying the temperature readings on the LCD.



If you press the pushbutton, it instantly toggles the LED attached to the ESP32 #1.



You can quickly expand this system to add more ESP32 boards to your network. You can create more MQTT topics to send commands and receive other sensor readings. To make this process more practical, you can also use a home automation platform to add a dashboard to your system, like Node-RED, Home Assistant, Domoticz, or OpenHAB, for example.

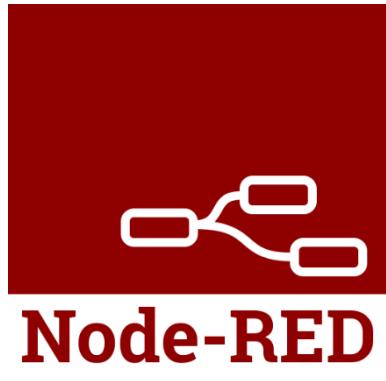


Wrapping Up

In this Unit (and the previous one), we've shown you how to exchange data between two ESP32 boards using MQTT. In the following Units, you'll learn how to control the ESP32 through Node-RED and send data from the ESP32 to the Node-RED dashboard using MQTT.

9.5 - Installing Node-RED and Node-RED Dashboard on a RPi

This Unit is a quick introduction to Node-RED. We'll cover what's Node-RED and how to install it. We'll also install the Node-RED Dashboard nodes.



Note: building complex flows with Node-RED or explore all its functionalities is not the purpose of this Unit. With this Unit (and the next one) we intend to provide the necessary information on connecting the ESP32 to Node-RED using the MQTT communication protocol. For an eBook dedicated to Node-RED with the ESP32, check out: [SMART HOME with Raspberry Pi, ESP32, and ESP8266 eBook](#).

Prerequisites

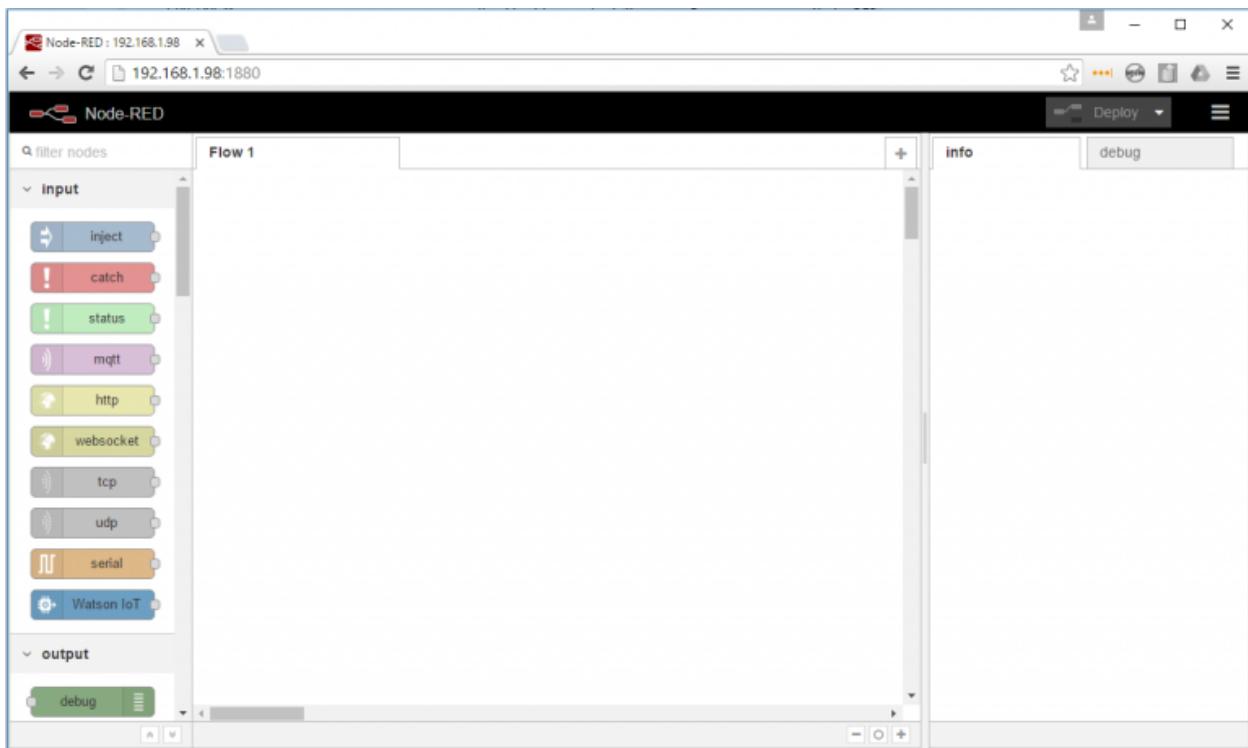
Before continuing:

- You should be familiar with the Raspberry Pi board – [read Getting Started with Raspberry Pi](#);
- You should have the Raspberry Pi OS installed in your Raspberry Pi – [Install Raspberry Pi OS, Set Up Wi-Fi, Enable and Connect with SSH](#)
- You also need the following hardware: [Raspberry Pi board](#) – read [Best Raspberry Pi Starter Kits, MicroSD Card – 32GB Class10](#), and [Power Supply \(5V 2.5A\)](#);
- You also need [Mosquitto MQTT Broker installed](#).

What's Node-RED?

[Node-RED](#) is a powerful open-source tool for building Internet of Things (IoT) applications with the goal of simplifying the programming component.

Node-RED runs on the web browser and uses visual programming that allows you to connect code blocks, known as nodes, together to perform a task. The nodes when wired together are called flows.



Why is Node-RED a great solution?

- Node-RED is open source and developed by IBM.
- The Raspberry Pi runs Node-RED perfectly.
- It is a visual programming tool, which makes it more accessible to a wider range of users.
- With Node-RED you can spend more time making cool stuff, rather than spending countless hours writing code.
- There are many Node-RED nodes and examples for almost anything you may want to build in your home automation system.
- Node-RED has a big community so you can always ask for help on Node-RED dedicated forums.

What can you do with Node-RED?

Node-RED makes it easy to:

- Control your RPi GPIOs;
- Establish an MQTT connection with other devices (ESP32, ESP8266, etc.);
- Create a responsive graphical user interface for your projects;
- Communicate with third-party services (IFTTT.com, Adafruit.io, Thing Speak, Home Assistant, InfluxDB, etc.);
- Retrieve data from the web (weather forecast, stock prices, emails, etc.);
- Create time-triggered events;
- Store and retrieve data from a database;
- And much more.

Here's a repository [with some examples of flows and nodes](#) for Node-RED.

Having an SSH connection established with your Raspberry Pi, follow the next steps to install Node-RED.

First, update and upgrade your system with the following command:

```
sudo apt update && sudo apt upgrade
```

At some point, you'll be asked if you want to proceed. Click **Y** and then **Enter**. This process may take a few minutes.

When it's completed, enter the following command to install Node-RED:

```
bash <(curl -sL  
https://raw.githubusercontent.com/node-red/linux-installers/m  
aster/deb/update-nodejs-and-nodered)
```

To paste a command on the Terminal window, simply click the mouse right button.

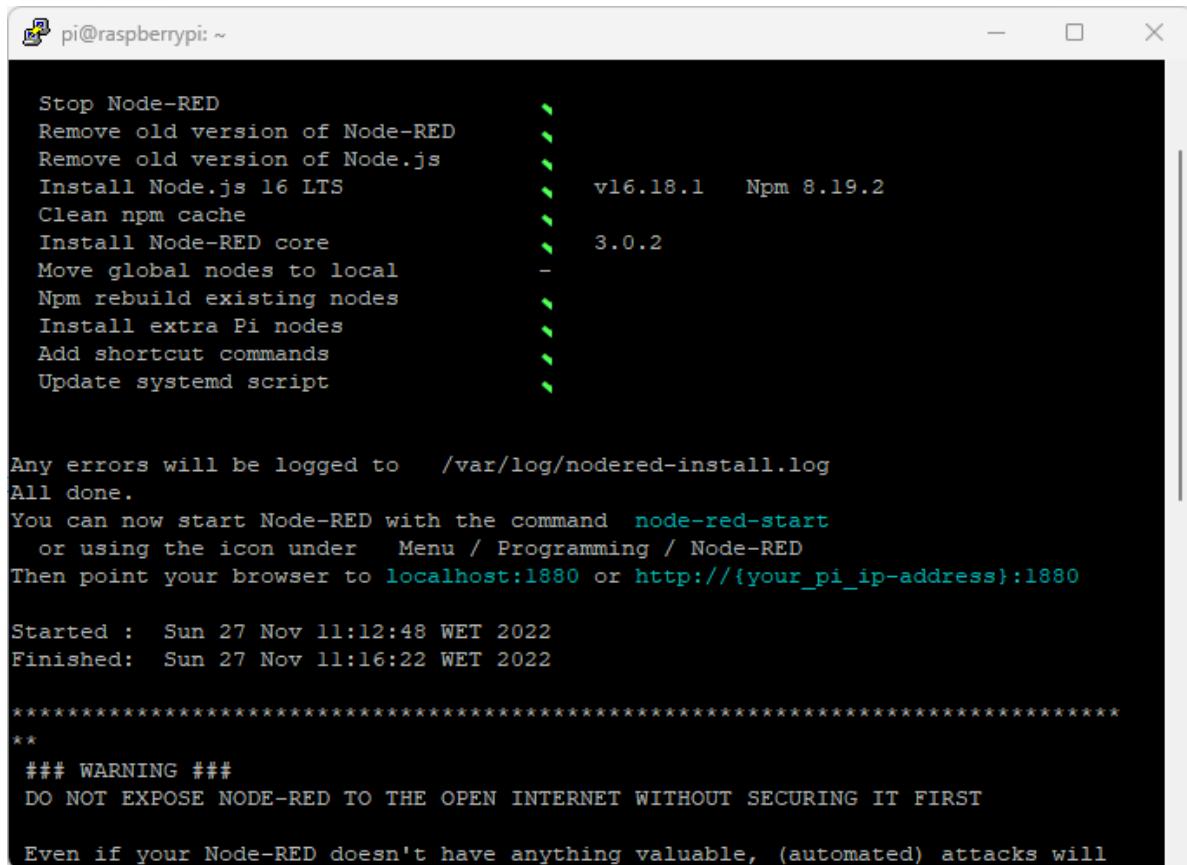
It won't work with CTRL+V. Then, press Enter to run the command.

If you get an error message like "*Curl: no URL specified*" or a similar error message, it means the command was not copied properly. If that's the case, we recommend going to the Node-RED installation instructions page and copy the command from there. Here's the link for the official installation instructions page:

- <https://nodered.org/docs/getting-started/raspberrypi>

After running the command, you'll be asked: "Would you like to install Pi-specific nodes?" Press **Y** and **Enter**.

It will take a few minutes to install Node-RED. In the end, you should get a similar message on the Terminal window (see next page).



The screenshot shows a terminal window titled "pi@raspberrypi: ~". The window contains the following text:

```
Stop Node-RED
Remove old version of Node-RED
Remove old version of Node.js
Install Node.js 16 LTS
Clean npm cache
Install Node-RED core
Move global nodes to local
Npm rebuild existing nodes
Install extra Pi nodes
Add shortcut commands
Update systemd script

Any errors will be logged to /var/log/nodered-install.log
All done.
You can now start Node-RED with the command node-red-start
or using the icon under Menu / Programming / Node-RED
Then point your browser to localhost:1880 or http://{your_pi_ip-address}:1880

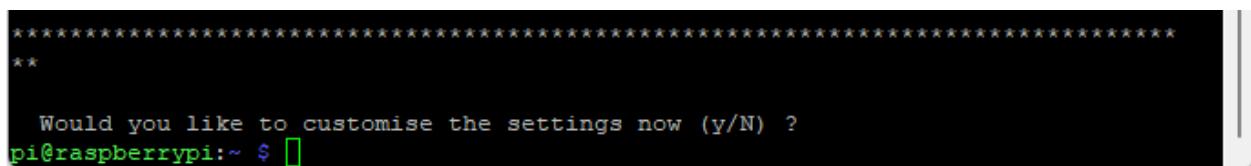
Started : Sun 27 Nov 11:12:48 WET 2022
Finished: Sun 27 Nov 11:16:22 WET 2022

*****
** 
### WARNING ###
DO NOT EXPOSE NODE-RED TO THE OPEN INTERNET WITHOUT SECURING IT FIRST

Even if your Node-RED doesn't have anything valuable, (automated) attacks will
```

Configure Node-RED Settings

You'll be asked if you want to customize the settings now. Press **Y** and then press **Enter** (in some versions this step is omitted and it will skip to the screenshot on the next page).

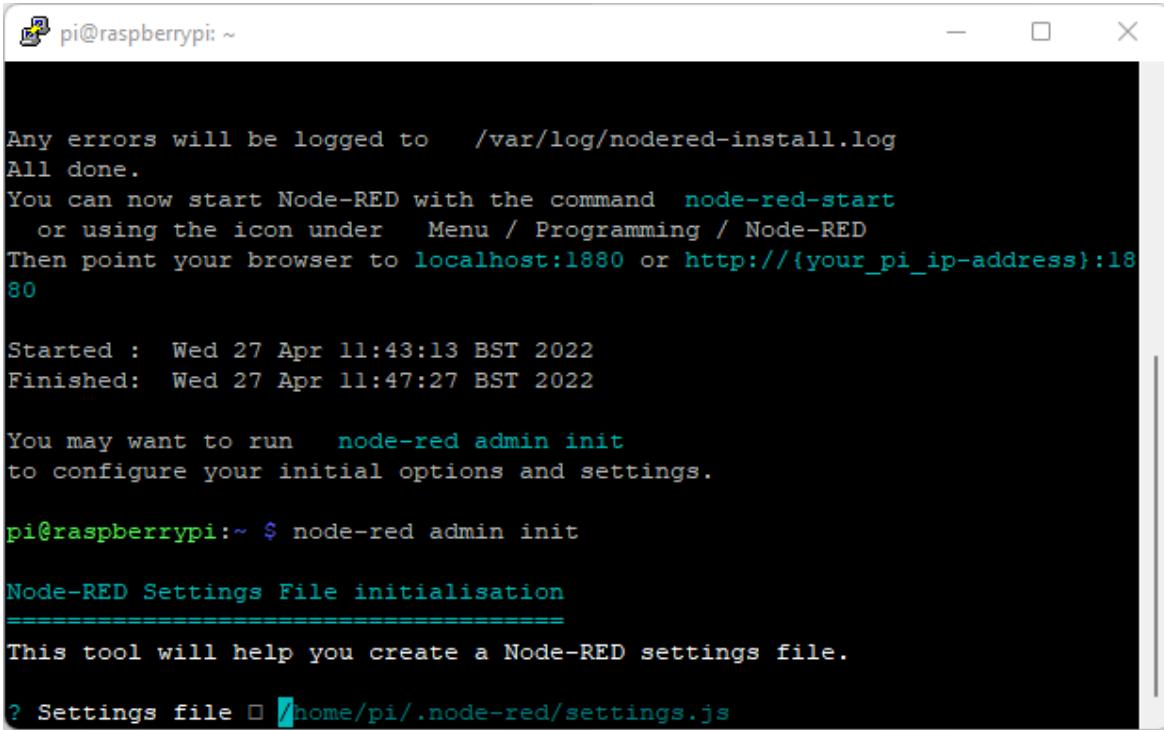


The screenshot shows a terminal window with the following text:

```
*****
**
Would you like to customise the settings now (y/N) ?
pi@raspberrypi:~ $
```

If you get an error, run the following command to start customizing the settings:

```
node-red admin init
```



```
pi@raspberrypi: ~
Any errors will be logged to    /var/log/nodered-install.log
All done.
You can now start Node-RED with the command  node-red-start
or using the icon under  Menu / Programming / Node-RED
Then point your browser to  localhost:1880 or http://{your_pi_ip-address}:18
80

Started :  Wed 27 Apr 11:43:13 BST 2022
Finished:  Wed 27 Apr 11:47:27 BST 2022

You may want to run  node-red admin init
to configure your initial options and settings.

pi@raspberrypi:~ $ node-red admin init

Node-RED Settings File initialisation
=====
This tool will help you create a Node-RED settings file.

? Settings file □ /home/pi/.node-red/settings.js
```

- Press **Enter** to create a Node-RED Settings file on `/home/pi/.node-red/settings.js` (keep the default `settings.js` file location)
- Do you want to set up user security? **Yes**.
- Enter a username and press **Enter** ([you need to remember it later](#)).
- Enter a password and press **Enter** ([you need to remember it later](#)).
- Then, you need to define user permissions. We'll set full access, make sure the **full access** option is highlighted in blue and press **Enter**.
- You can add other users with different permissions if you want. We'll just create one user for now. You can always add other users later.
- Do you want to enable the Projects feature? **No**.
- Enter a name for your flows file. Press **Enter** to select the default name `flows.json`.
- Provide a passphrase to encrypt your credentials file. Learn more about what is a [passphrase](#).
- Select a theme for the editor. Simply press **Enter** to select default.
- Press **Enter** again to select the default text editor.
- Allow Function nodes to load external modules? **Yes**.

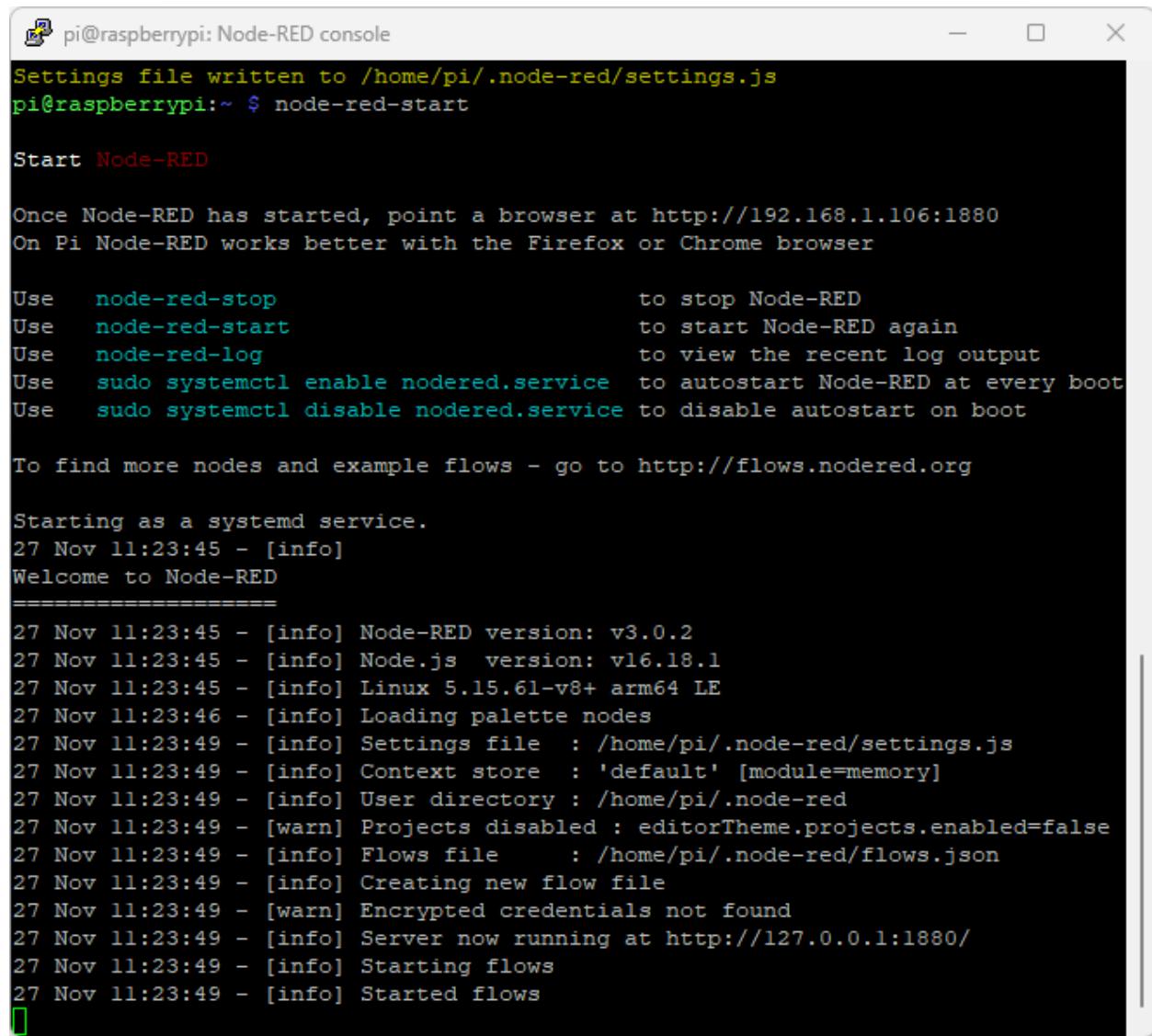
Node-RED configuration was successful. All settings are saved on `settings.js`.

Start Node-RED

Run the following command to start Node-RED:

```
node-red-start
```

You should get a similar message in the Terminal window:



```
pi@raspberrypi: Node-RED console
Settings file written to /home/pi/.node-red/settings.js
pi@raspberrypi:~ $ node-red-start

Start Node-RED

Once Node-RED has started, point a browser at http://192.168.1.106:1880
On Pi Node-RED works better with the Firefox or Chrome browser

Use node-red-stop          to stop Node-RED
Use node-red-start         to start Node-RED again
Use node-red-log           to view the recent log output
Use sudo systemctl enable nodered.service to autostart Node-RED at every boot
Use sudo systemctl disable nodered.service to disable autostart on boot

To find more nodes and example flows - go to http://flows.nodered.org

Starting as a systemd service.
27 Nov 11:23:45 - [info]
Welcome to Node-RED
=====
27 Nov 11:23:45 - [info] Node-RED version: v3.0.2
27 Nov 11:23:45 - [info] Node.js version: v16.18.1
27 Nov 11:23:45 - [info] Linux 5.15.61-v8+ arm64 LE
27 Nov 11:23:46 - [info] Loading palette nodes
27 Nov 11:23:49 - [info] Settings file : /home/pi/.node-red/settings.js
27 Nov 11:23:49 - [info] Context store : 'default' [module=memory]
27 Nov 11:23:49 - [info] User directory : /home/pi/.node-red
27 Nov 11:23:49 - [warn] Projects disabled : editorTheme.projects.enabled=false
27 Nov 11:23:49 - [info] Flows file : /home/pi/.node-red/flows.json
27 Nov 11:23:49 - [info] Creating new flow file
27 Nov 11:23:49 - [warn] Encrypted credentials not found
27 Nov 11:23:49 - [info] Server now running at http://127.0.0.1:1880/
27 Nov 11:23:49 - [info] Starting flows
27 Nov 11:23:49 - [info] Started flows
```

Autostart Node-RED on boot

To automatically run Node-RED when the Pi boots up, you need to enter the following command (press **CTRL + C** to stop the previous task before entering the new command).

```
sudo systemctl enable nodered.service
```

This means that as long as your Raspberry Pi is powered up, Node-RED will be running.

Now, restart your Pi so the autostart takes effect. The next time the Raspberry Pi restarts, Node-RED will be already running.

```
sudo reboot
```

Note: If, later on, you want to disable autostart on boot, you can run:
`sudo systemctl disable nodered.service`

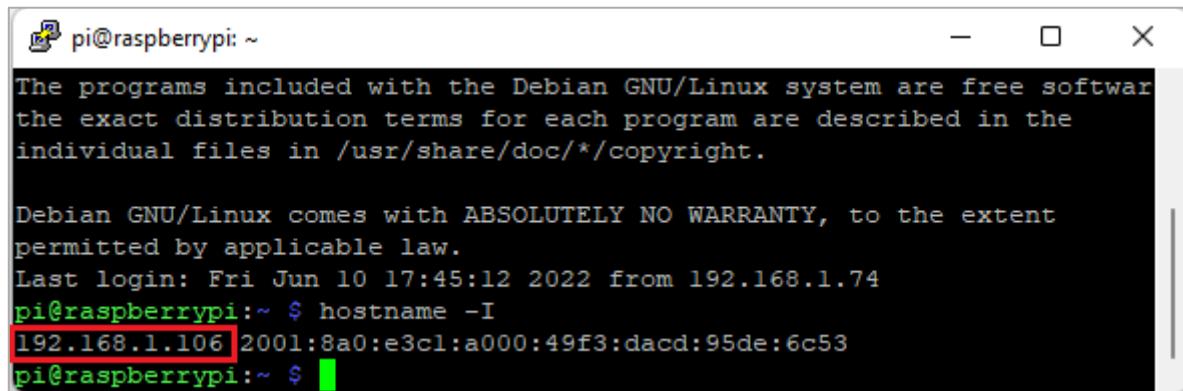
For more information about the installation process, check the [official documentation](#).

Access Node-RED

Node-RED runs on port 1880. To access Node-RED open a browser on your computer and type the Raspberry Pi IP address followed by :1880. For example, in my case:

```
192.168.1.106:1880
```

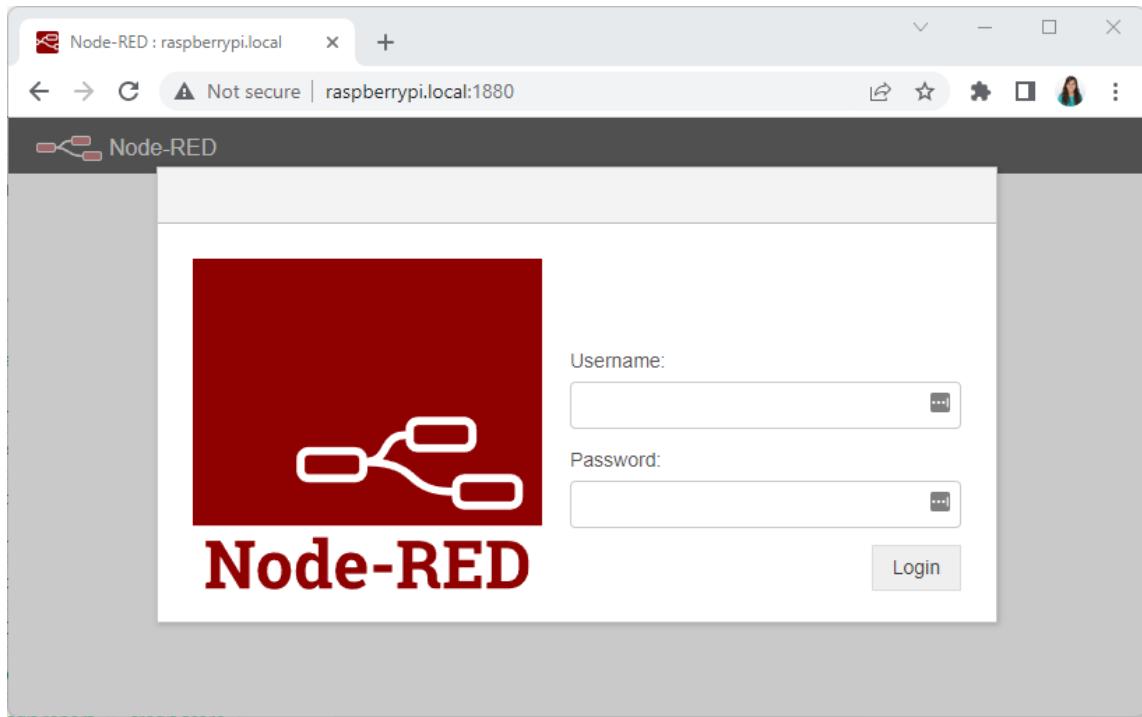
Remember: to get your Raspberry Pi IP address, you can run the following command: `hostname -I`



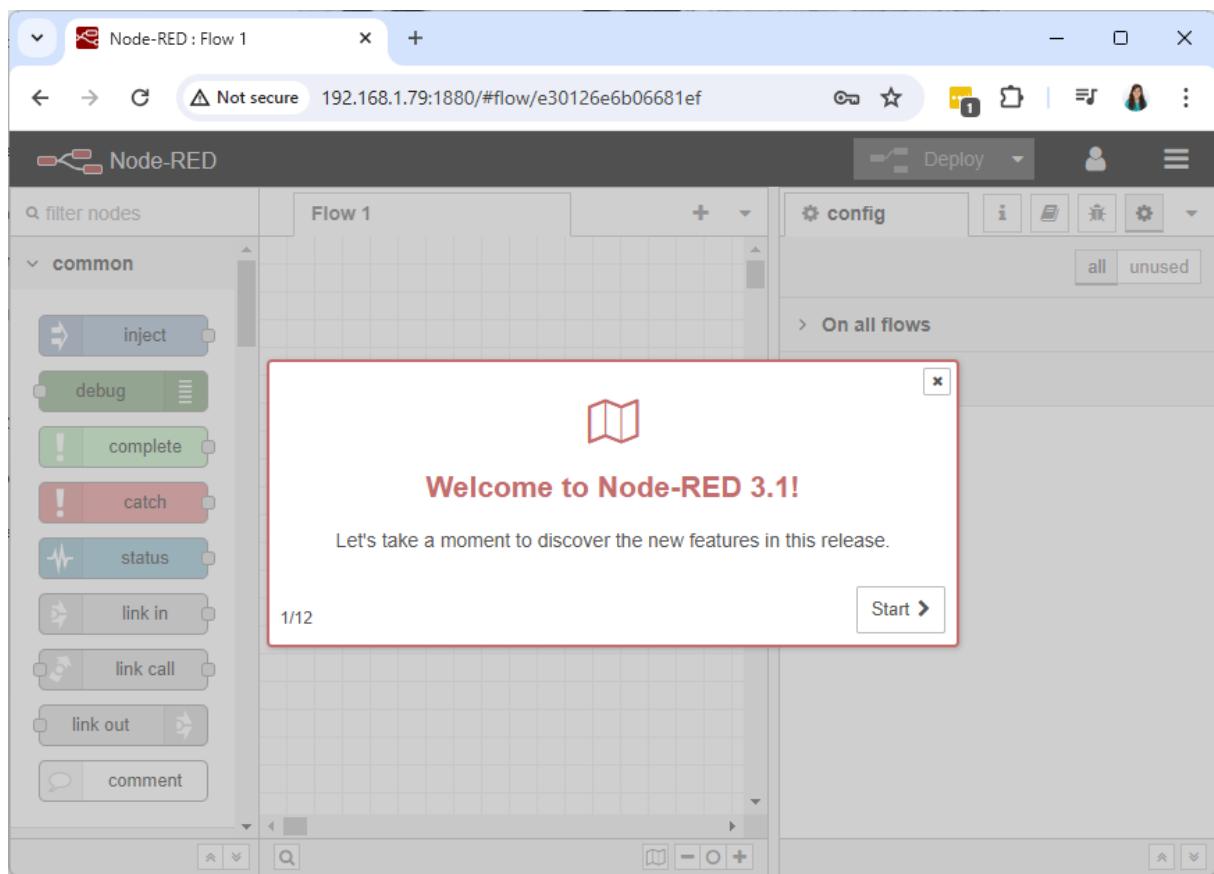
```
pi@raspberrypi: ~
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Jun 10 17:45:12 2022 from 192.168.1.74
pi@raspberrypi:~ $ hostname -I
192.168.1.106 2001:8a0:e3c1:a000:49f3:dacd:95de:6c53
pi@raspberrypi:~ $
```

After entering the Raspberry Pi IP address followed by :1880 on the web browser, the Node-RED login page should load. Insert your Node-RED username and password that you've defined when configuring the Node-RED settings.

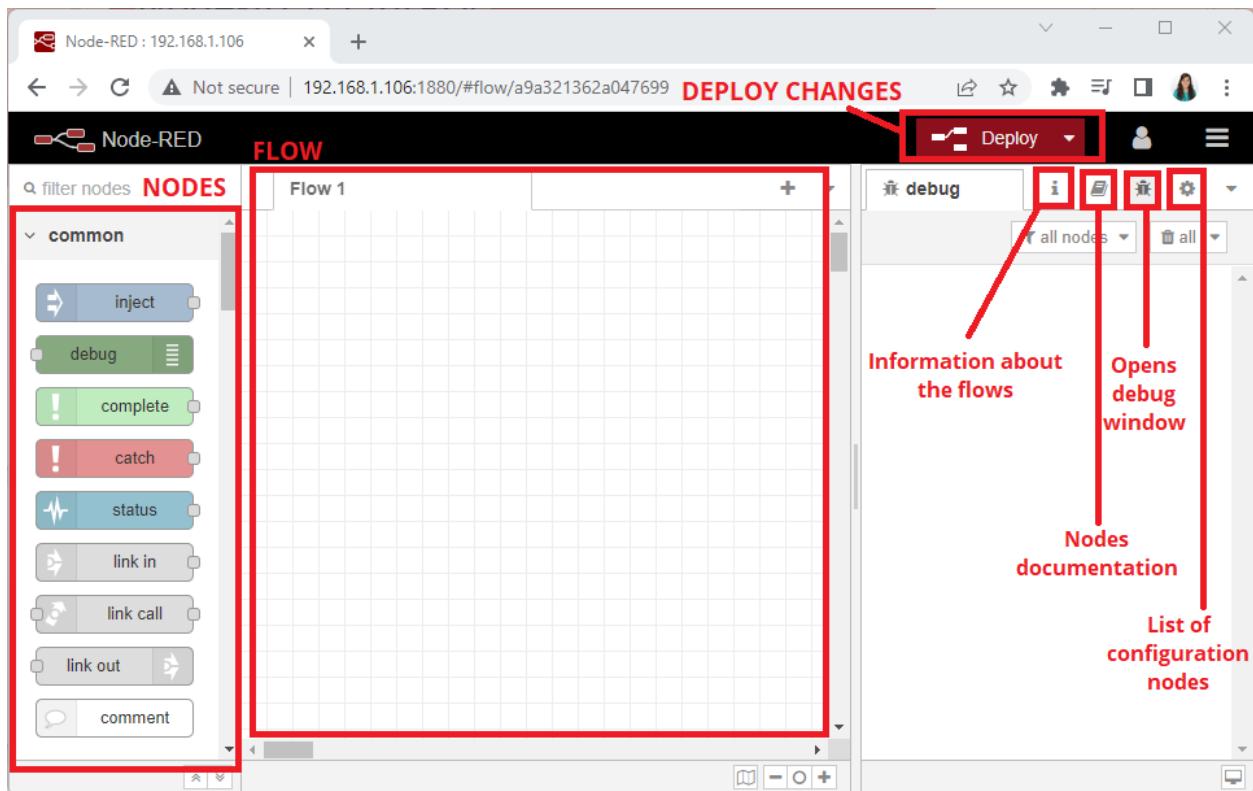


Now, you have access to Node-RED. You can start building your flows.



Node-RED Overview

The following picture shows the Node-RED main sections labeled.



Nodes

On the left sidebar, you can see a list with a bunch of blocks. These blocks are called **nodes** and they are separated by their functionality. If you select a node, you can see how it works in the **nodes documentation** tab.

Nodes have input and/or output ports to receive and send information to other nodes. For example, a node receives an input from a previous node, processes that information, and outputs a different message to another node that will do something with that information. The information passed between nodes is called a **message**.

Flow

The **nodes** are the building blocks of a **flow**. You wire nodes together to create a **flow** that will perform a certain task. A **Flow** is also a tab in the workspace where you place and organize the nodes.

In the center, you have the **Flow** and this is where you place the nodes.

Right Sidebar

The right sidebar presents several tools.

- **Information:** shows information about the flows;
- **Help:** shows the nodes' documentation;
- **Debug:** the bug icon opens a debugging window that shows messages passed to debug nodes—it's useful for debugging purposes;
- **Config nodes:** the gear icon shows information about configuration nodes. Configuration nodes do not appear on the main workspace, and they are special nodes that hold reusable configurations that can be shared by several nodes in a flow like MQTT broker settings.
- **Deploy:** the deploy button saves all changes made to the flow and starts running the flow.

Creating a Simple Flow

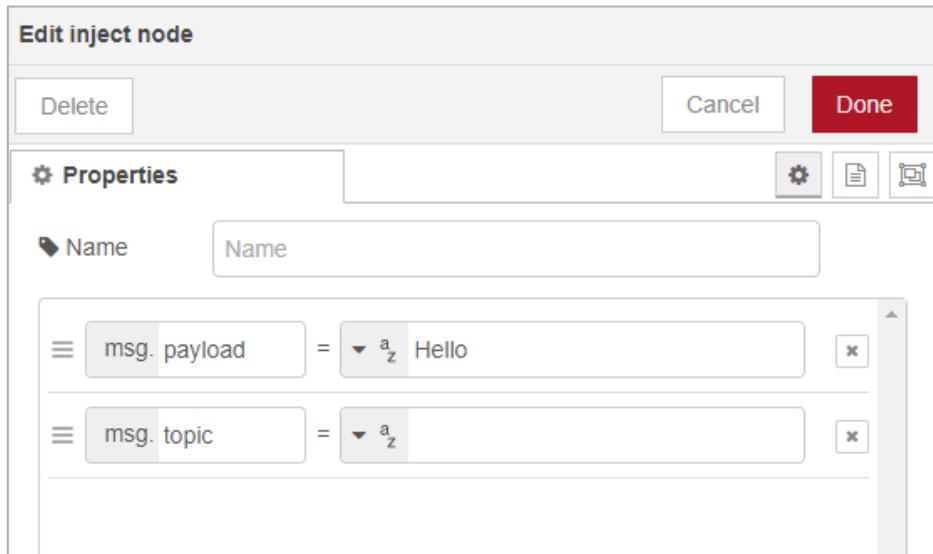
To get you used to the Node-RED interface, let's create a simple flow. The flow we'll create, simply prints a message to the debug console, when triggered.

Drag an **inject** node and a **debug** node to your flow and wire them together. When dragged to the flow, the inject node will change its name to **timestamp** and the debug node to **msg.payload** or **debug 1** (depending on the Node-RED version) by default.



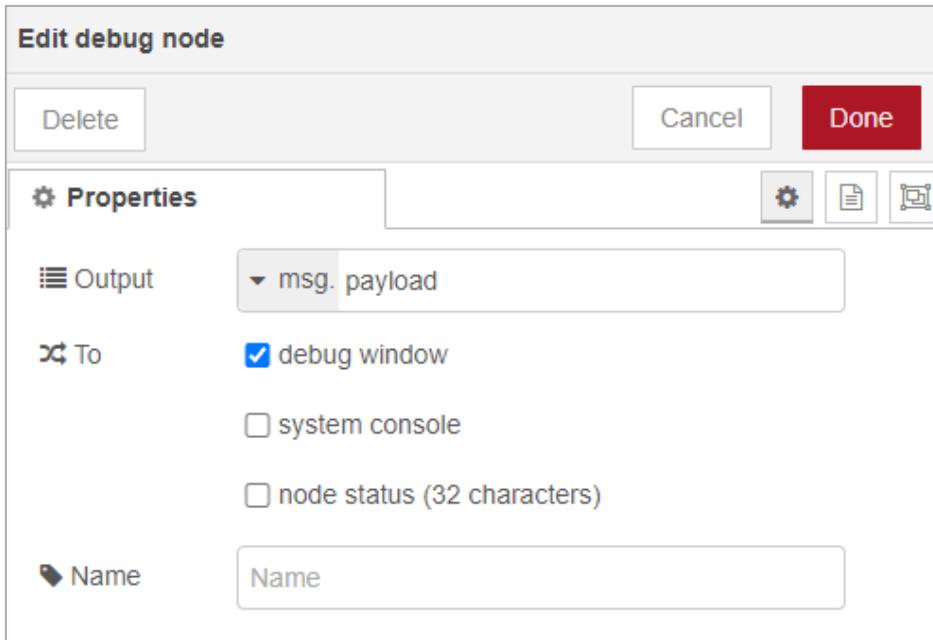
Now, let's edit the **inject** node. Double-click the node. In the figure below, you can see the different settings you can change.

On the **msg.payload** field, select **string** and type **Hello**. Then, click **Done**.



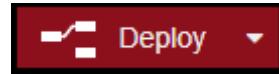
Messages (`msg`) in Node-RED are JavaScript objects that can have multiple properties. The `payload` is the default property most nodes work with. You can think of it as the main content of the message you want to send to the next node. In our case, we're simply sending a text message.

We won't edit the **debug** node for now, but you can double-click on it to check its properties.



You can select the output of the debug node, which is `msg.payload`, and where we want to send that output. In our case, we want to send it to the debug window.

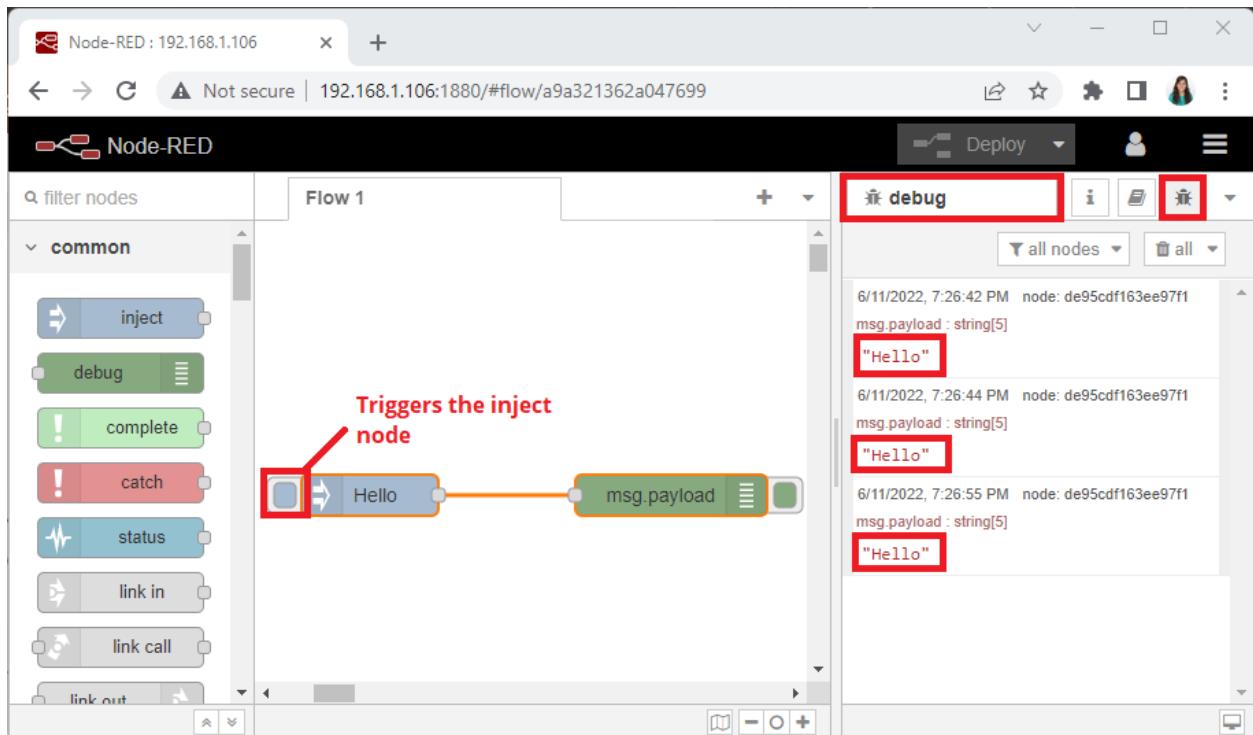
To save your application, you need to click the **Deploy** button in the top right corner.



Your application is saved.

Testing the flow

Let's test our simple flow. Open the debug window. Then, click the inject node to trigger the flow.



As you can see, our message is printed in the debug window when you trigger the inject node. This is a very basic example and it doesn't do anything useful. However, the purpose of this unit is to get you familiar with the Node-RED interface.

Installing Node-RED Dashboard

Node-RED Dashboard is a module that provides a set of nodes in Node-RED to quickly create a live data dashboard. For example, it provides nodes to quickly create a user interface with buttons, sliders, charts, gauges, etc.

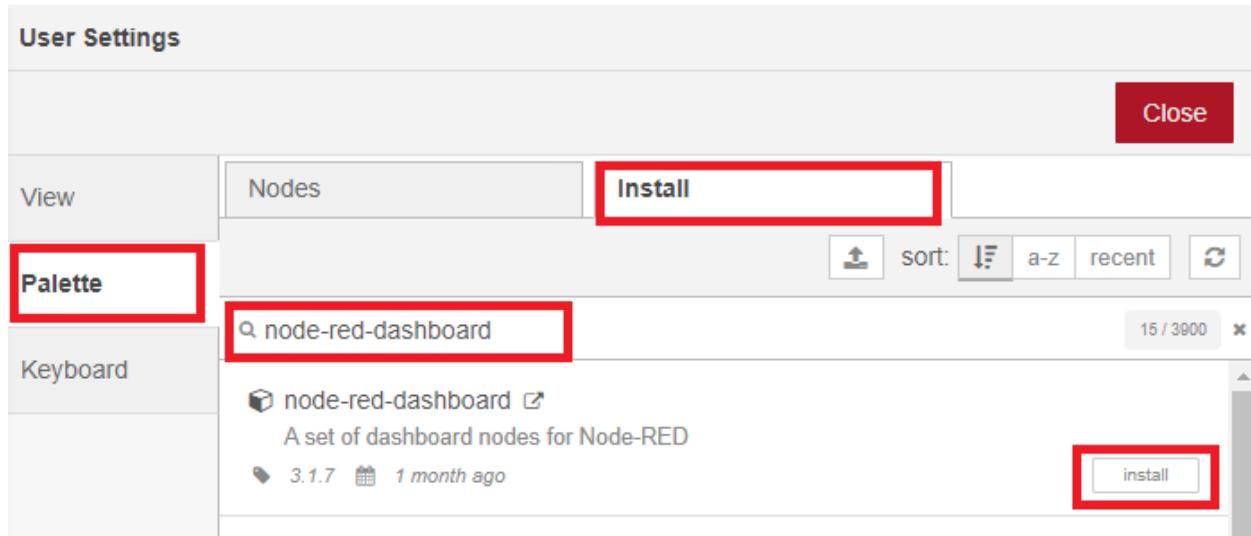
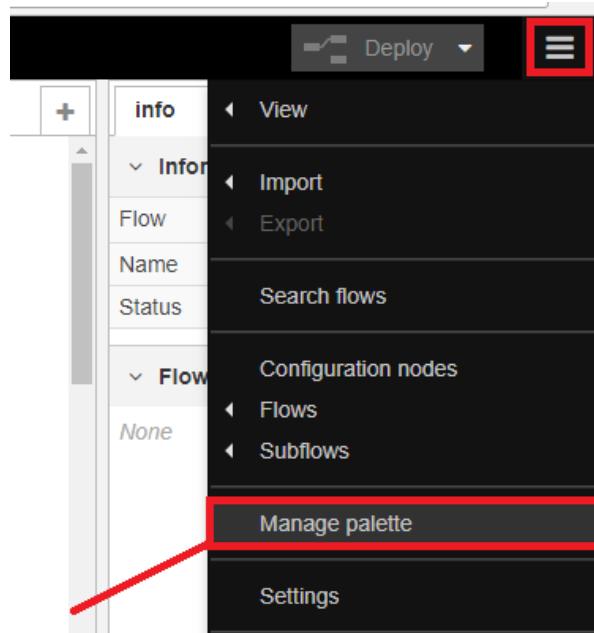
To learn more about Node-RED Dashboard you can check the following links:

- Node-RED site: <https://flows.nodered.org/node/node-red-dashboard>

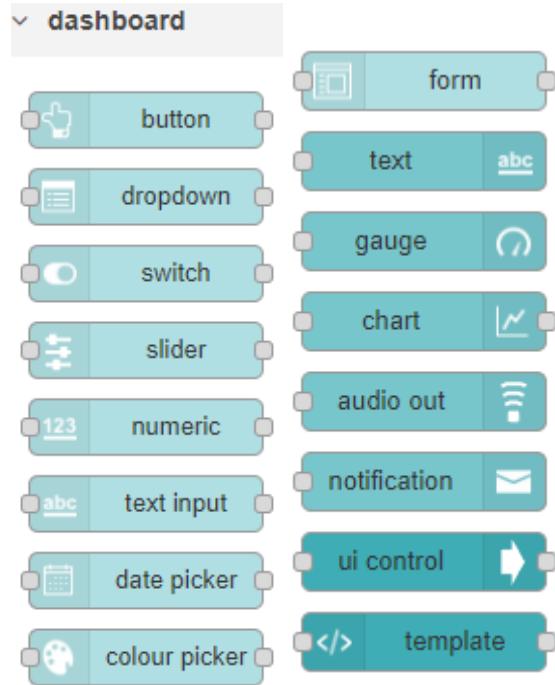
- GitHub: <https://github.com/node-red/node-red-dashboard>

You can install Node-RED dashboard nodes using the **Menu > Manage Palette**.

Then, search for `node-red-dashboard` and install it.



After installing, the dashboard nodes will show up on the palette.

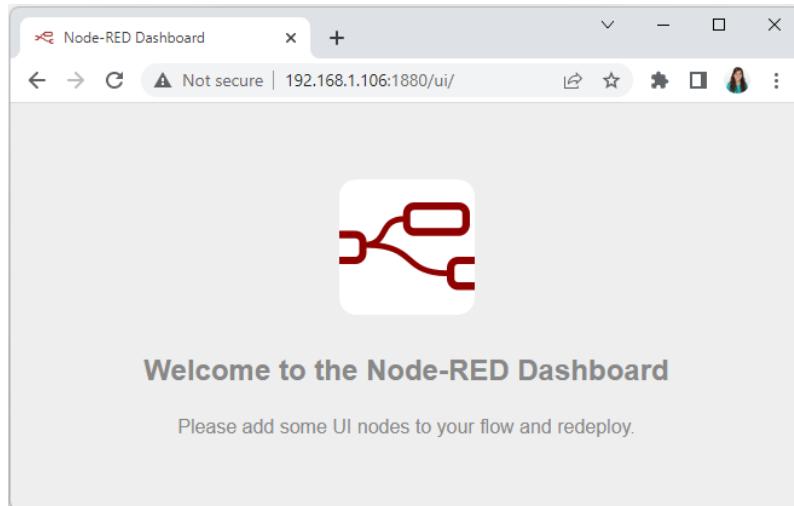


Dashboard nodes provide widgets that show up in your application user interface (UI). The user interface is accessible on the following URL:

```
http://Your_RPi_IP_address:1880/ui
```

For example, in my case:

```
http://192.168.1.106:1880/ui
```



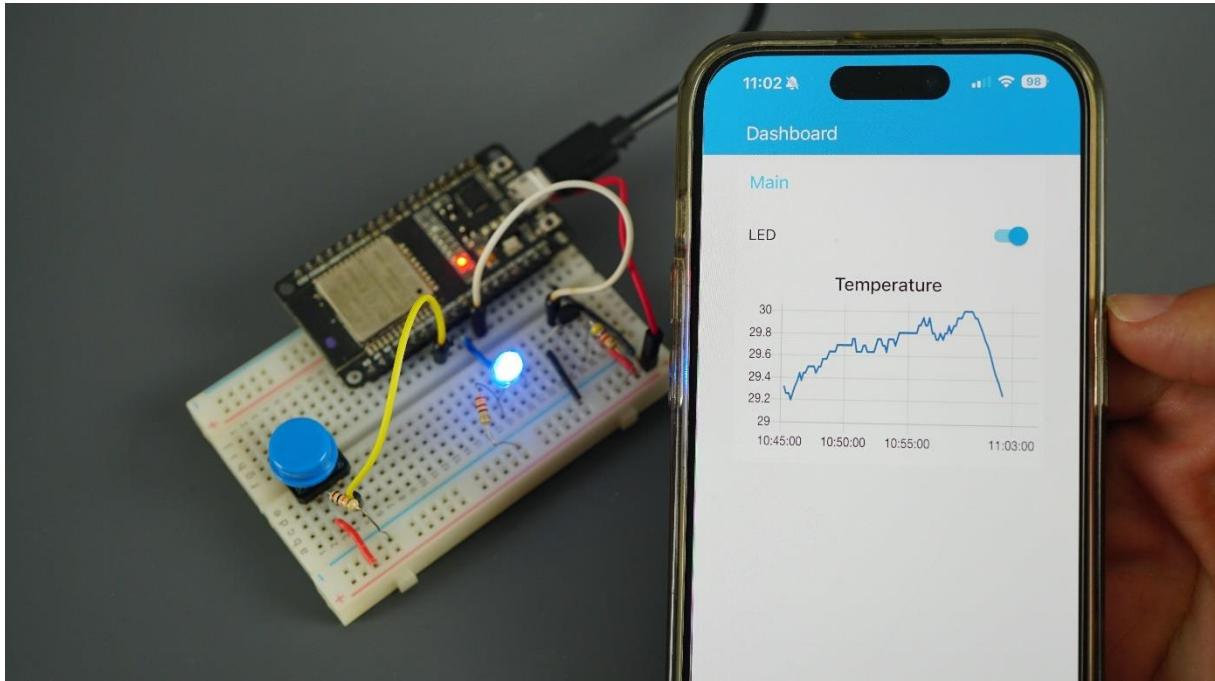
At the moment, you'll see the previous screen when you access the UI. That's because you haven't added any of those dashboard nodes to the flow.

Continue to the Next Unit ...

Now, you have everything ready to integrate Node-RED with your ESP32 and create a user interface. Continue to the next unit to create a Node-RED user interface to interact with the ESP32 board.

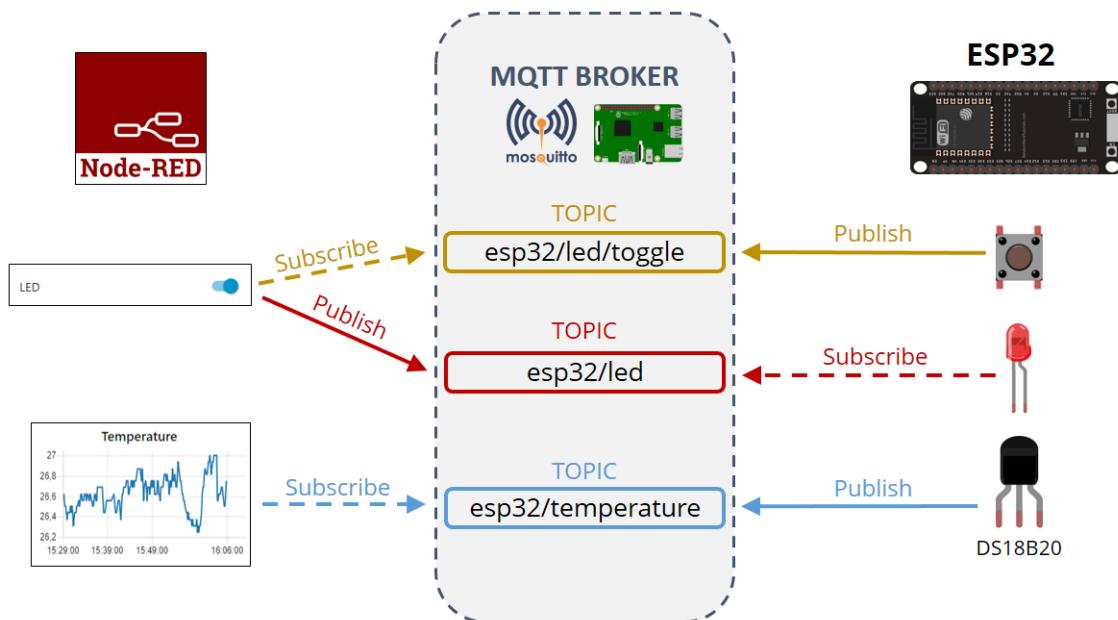
9.6 - Connect the ESP32 to Node-RED using MQTT

This unit will demonstrate how to use Node-RED Dashboard to control the ESP32 GPIOs and display temperature readings in a chart. We'll build a simple project to illustrate the most important concepts (publish and subscribe with Node-RED).



Project Overview

Here's a high-level overview of the project we'll build:



- The ESP32 is attached to a pushbutton that, when pressed, publishes on the **esp32/led/toggle** topic. Node-RED is subscribed to that topic, and when it receives a message, the dashboard LED switch changes its state;
- When the dashboard LED switch changes its state, it publishes a message on the **esp32/led** topic. The ESP32 is subscribed to that topic, and when it receives a message, it toggles the LED. (Instead of an LED, you can control any other output);
- The ESP32 takes temperature readings with the DS18B20 sensor. The readings are published in the **esp32/temperature** topic;
- Node-RED is subscribed to the **esp32/temperature** topic. So, it receives the DS18B20 temperature readings and publishes the readings in a chart.

Preparing Your ESP32

If you've followed all previous Units in this Module, you should have all the necessary Arduino IDE libraries installed. If you don't, make sure you install the following libraries before continuing with this Unit:

- [Async TCP](#)
- [Async MQTT Client](#)
- [Dallas Temperature](#)
- [One Wire](#)

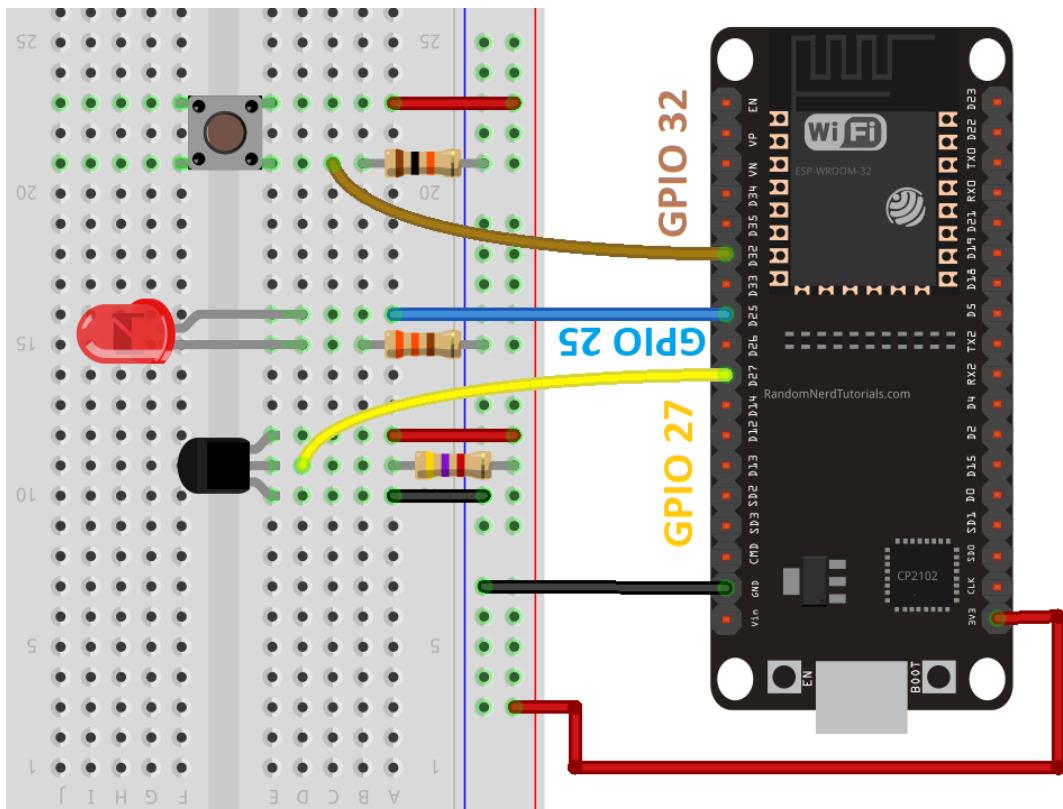
Wiring the Circuit

To build the circuit for this project, you need the following parts:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [220 Ohm resistor \(or similar value\)](#)
- [DS18B20 temperature sensor](#)
- [4.7k Ohm resistor](#)
- [Pushbutton](#)
- [10k Ohm resistor](#)

- [Jumper wires](#)
- [Breadboard](#)

Wire the circuit by following the next schematic diagram:



Uploading the code

After installing the required libraries, copy the following code to your Arduino IDE.

- [Click here to download the code.](#)

```
#include <WiFi.h>
extern "C" {
    #include "freertos/FreeRTOS.h"
    #include "freertos/timers.h"
}
#include <AsyncMqttClient.h>
#include <OneWire.h>
#include <DallasTemperature.h>

// Change the credentials below, so your ESP32 connects to your router
#define WIFI_SSID "REPLACE_WITH_YOUR_SSID"
#define WIFI_PASSWORD " REPLACE_WITH_YOUR_PASSWORD"

// Change the MQTT_HOST variable to your Raspberry Pi IP address,
// so it connects to your Mosquitto MQTT broker
#define MQTT_HOST IPAddress(xxx, xxx, xxx, xxx)
#define MQTT_PORT 1883

#define BROKER_USER "REPLACE_WITH_YOUR_BROKER_USERNAME"
```

```

#define BROKER_PASS " REPLACE_WITH_YOUR_BROKER_PASSWORD"

// Create objects to handle MQTT client
AsyncMqttClient mqttClient;
TimerHandle_t mqttReconnectTimer;
TimerHandle_t wifiReconnectTimer;

unsigned long previousMillis = 0;    // Stores last time temperature was published
const long interval = 10000;        // interval at which to publish sensor readings

const int ledPin = 25;              // GPIO where the LED is connected to
int ledState = LOW;                // the current state of the output pin

// GPIO where the DS18B20 is connected to
const int oneWireBus = 27;
// Setup a oneWire instance to communicate with any OneWire devices
OneWire oneWire(oneWireBus);
// Pass our oneWire reference to Dallas Temperature sensor
DallasTemperature sensors(&oneWire);

const int buttonPin = 32;           // Define GPIO where the pushbutton is connected
int buttonState;                  // current reading from the input pin (pushbutton)
int lastButtonState = LOW;         // previous reading from the input pin (pushbutton)
unsigned long lastDebounceTime = 0; // the last time the output pin was toggled
unsigned long debounceDelay = 50;   // the debounce time; increase if the output flicker

void connectToWifi() {
    Serial.println("Connecting to Wi-Fi...");
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}

void connectToMqtt() {
    Serial.println("Connecting to MQTT...");
    mqttClient.connect();
}

void WiFiEvent(WiFiEvent_t event) {
    Serial.printf("[WiFi-event] event: %d\n", event);
    switch(event) {
        case ARDUINO_EVENT_WIFI_STA_GOT_IP:
            Serial.println("WiFi connected");
            Serial.println("IP address: ");
            Serial.println(WiFi.localIP());
            connectToMqtt();
            break;
        case ARDUINO_EVENT_WIFI_STA_DISCONNECTED:
            Serial.println("WiFi lost connection");
            // ensure we don't reconnect to MQTT while reconnecting to Wi-Fi
            xTimerStop(mqttReconnectTimer, 0);
            xTimerStart(wifiReconnectTimer, 0);
            break;
    }
}

// Add more topics that want your ESP32 to be subscribed to
void onMqttConnect(bool sessionPresent) {
    Serial.println("Connected to MQTT.");
    Serial.print("Session present: ");
    Serial.println(sessionPresent);
    // ESP32 subscribed to esp32/led topic
    uint16_t packetIdSub = mqttClient.subscribe("esp32/led", 0);
}

```

```

    Serial.print("Subscribing at QoS 0, packetId: ");
    Serial.println(packetIdSub);
}

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
    Serial.println("Disconnected from MQTT.");
    if (WiFi.isConnected()) {
        xTimerStart(mqttReconnectTimer, 0);
    }
}

void onMqttSubscribe(uint16_t packetId, uint8_t qos) {
    Serial.println("Subscribe acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
    Serial.print(" qos: ");
    Serial.println(qos);
}

void onMqttUnsubscribe(uint16_t packetId) {
    Serial.println("Unsubscribe acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
}

void onMqttPublish(uint16_t packetId) {
    Serial.println("Publish acknowledged.");
    Serial.print(" packetId: ");
    Serial.println(packetId);
}

// You can modify this function to handle what happens when you receive a
// certain message in a specific topic
void onMqttMessage(char* topic, char* payload, AsyncMqttClientMessageProperties
properties, size_t len, size_t index, size_t total) {
    String messageTemp;
    for (int i = 0; i < len; i++) {
        //Serial.print((char)payload[i]);
        messageTemp += (char)payload[i];
    }
    // Check if the MQTT message was received on topic esp32/led
    if (strcmp(topic, "esp32/led") == 0) {
        // If the LED is off turn it on (and vice-versa)
        if (messageTemp == "on") {
            digitalWrite(ledPin, HIGH);
        }
        else if (messageTemp == "off") {
            digitalWrite(ledPin, LOW);
        }
    }
    Serial.println("Publish received.");
    Serial.print(" message: ");
    Serial.println(messageTemp);
    Serial.print(" topic: ");
    Serial.println(topic);
    Serial.print(" qos: ");
    Serial.println(properties.qos);
    Serial.print(" dup: ");
    Serial.println(properties.dup);
    Serial.print(" retain: ");
}

```

```

Serial.println(properties.retain);
Serial.print(" len: ");
Serial.println(len);
Serial.print(" index: ");
Serial.println(index);
Serial.print(" total: ");
Serial.println(total);
}

void setup() {
    // Start the DS18B20 sensor
    sensors.begin();

    // Define LED as an OUTPUT and set it LOW
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);

    // Define buttonPin as an INPUT
    pinMode(buttonPin, INPUT);

    Serial.begin(115200);

    mqttReconnectTimer = xTimerCreate("mqttTimer", pdMS_TO_TICKS(2000), pdFALSE,
                                      (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToMqtt));
    wifiReconnectTimer = xTimerCreate("wifiTimer", pdMS_TO_TICKS(2000), pdFALSE,
                                      (void*)0, reinterpret_cast<TimerCallbackFunction_t>(connectToWifi));

    WiFi.onEvent(WiFiEvent);

    mqttClient.onConnect(onMqttConnect);
    mqttClient.onDisconnect(onMqttDisconnect);
    mqttClient.onSubscribe(onMqttSubscribe);
    mqttClient.onUnsubscribe(onMqttUnsubscribe);
    mqttClient.onMessage(onMqttMessage);
    mqttClient.onPublish(onMqttPublish);
    mqttClient.setServer(MQTT_HOST, MQTT_PORT);
    mqttClient.setCredentials(BROKER_USER, BROKER_PASS);

    connectToWifi();
}

void loop() {
    unsigned long currentMillis = millis();
    // Every X number of seconds (interval = 5 seconds)
    // it publishes a new MQTT message on topic esp32/temperature
    if (currentMillis - previousMillis >= interval) {
        // Save the last time a new reading was published
        previousMillis = currentMillis;
        // New temperature readings
        sensors.requestTemperatures();

        // Publish an MQTT message on topic esp32/temperature with Celsius degrees
        uint16_t packetIdPub2 = mqttClient.publish("esp32/temperature", 2, true,
                                                    String(sensors.getTempCByIndex(0)).c_str());

        // Publish an MQTT message on topic esp32/temperature with Fahrenheit degrees
        //uint16_t packetIdPub2 = mqttClient.publish("esp32/temperature", 2, true,
        //                                            String(sensors.getTempFByIndex(0)).c_str());

        Serial.print("Publishing on topic esp32/temperature at QoS 2, packetId: ");
        Serial.println(packetIdPub2);
}

```

```

}

// Read the state of the pushbutton and save it in a local variable
int reading = digitalRead(buttonPin);

// If the pushbutton state changed (due to noise or pressing it), reset the timer
if (reading != lastButtonState) {
    // Reset the debouncing timer
    lastDebounceTime = millis();
}

// If the button state has changed, after the debounce time
if ((millis() - lastDebounceTime) > debounceDelay) {
    // And if the current reading is different than the current buttonState
    if (reading != buttonState) {
        buttonState = reading;
        // Publish an MQTT message on topic esp32/led/toggle to
        // toggle the LED (turn the LED on or off)
        if ((buttonState == HIGH)) {
            if (!digitalRead(ledPin)) {
                mqttClient.publish("esp32/led/toggle", 0, true, "on");
                Serial.println("Publishing on topic esp32/led/toggle topic at QoS 0");
            }
            else if (digitalRead(ledPin)) {
                mqttClient.publish("esp32/led/toggle", 0, true, "off");
                Serial.println("Publishing on topic esp32/led/toggle topic at QoS 0");
            }
        }
    }
}
// Save the reading. Next time through the loop, it'll be the lastButtonState
lastButtonState = reading;
}

```

To make this code work, you just need to type your SSID, password, MQTT broker IP address, username and password. The code will work straight away.

```

// Change the credentials below, so your ESP32 connects to your router
#define WIFI_SSID "REPLACE_WITH_YOUR_SSID"
#define WIFI_PASSWORD " REPLACE_WITH_YOUR_PASSWORD"

// Change the MQTT_HOST variable to your Raspberry Pi IP address,
// so it connects to your Mosquitto MQTT broker
#define MQTT_HOST IPAddress(xxx, xxx, xxx, xxx)
#define MQTT_PORT 1883
#define BROKER_USER "REPLACE_WITH_YOUR_BROKER_USERNAME"
#define BROKER_PASS " REPLACE_WITH_YOUR_BROKER_PASSWORD"

```

We won't explore how this code works. The parts related to MQTT publishing and subscribing were already covered in previous Units.

Temperature Celsius/Fahrenheit

By default, the code is publishing the temperature in Celsius degrees. If you want to publish temperature readings in Fahrenheit, go to the `loop()` and comment these two lines:

```
// Publish an MQTT message on topic esp32/temperature with Fahrenheit degrees
//uint16_t packetIdPub2 = mqttClient.publish("esp32/temperature", 2, true,
//                                         String(sensors.getTempFByIndex(0)).c_str());
```

Then, uncomment the next two lines

```
// Publish an MQTT message on topic esp32/temperature with Celsius degrees
uint16_t packetIdPub2 = mqttClient.publish("esp32/temperature", 2, true,
                                         String(sensors.getTempCByIndex(0)).c_str());
```

Creating the Node-RED flow

Before creating the flow, you need to have the following installed on your Raspberry Pi (covered in previous units):

- Node-RED
- Node-RED Dashboard
- Mosquitto MQTT Broker

Importing the flow

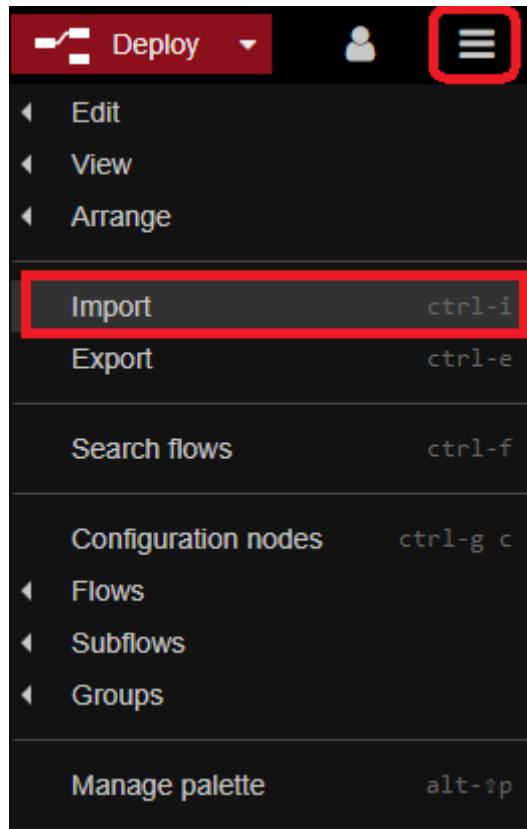
After that, import the Node-RED flow provided. Go to the [GitHub repository to see the raw file, and copy the flow provided.](#)



A screenshot of a GitHub raw file viewer. The top bar shows "2 lines (1 sloc) | 2.32 KB". Below the code area are buttons for "Raw", "Blame", "History", and icons for download, edit, and delete. The code itself is a single line of JSON:

```
1  [{"id": "9e58624.7faaba", "type": "mqtt out", "z": "c02b79b2.501998", "name": "", "topic": "esp32/led", "qos": "", "retain": "", "broker": "10e78a89.5b4fd"}]
```

Next, in the Node-RED window, at the top right corner, select the menu, and go to **Import**.

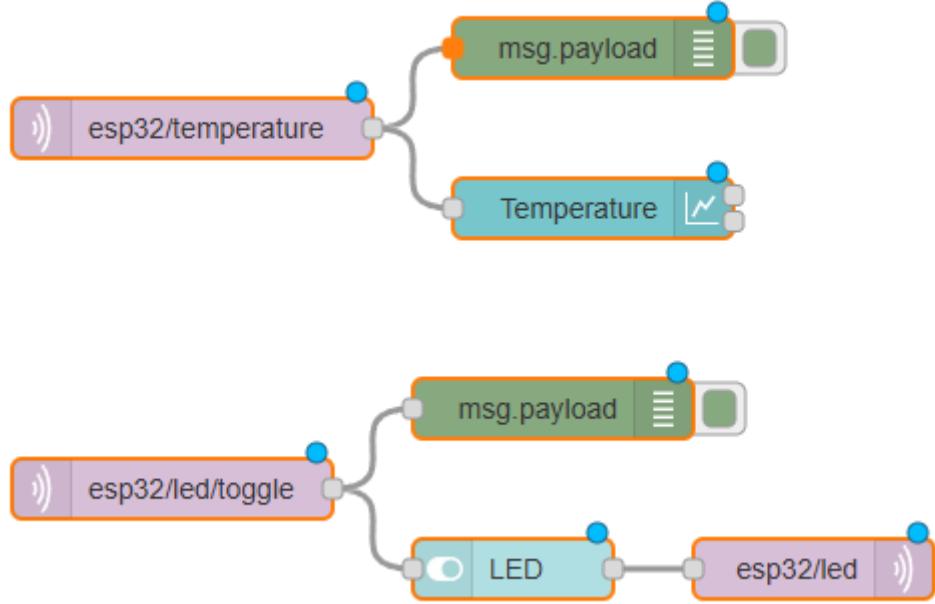


Then, paste the flow provided and click the **Import** button.

The screenshot shows the 'Import nodes' dialog box. On the left, there's a sidebar with 'Clipboard' and 'Local' tabs. The 'Clipboard' tab is selected, showing a large JSON string representing a flow. To the right of the sidebar is a text input field labeled 'Paste flow json or select a file to import'. Below the clipboard area, there are two buttons: 'Import to current flow' and 'new flow'. At the bottom right are 'Cancel' and 'Import' buttons.

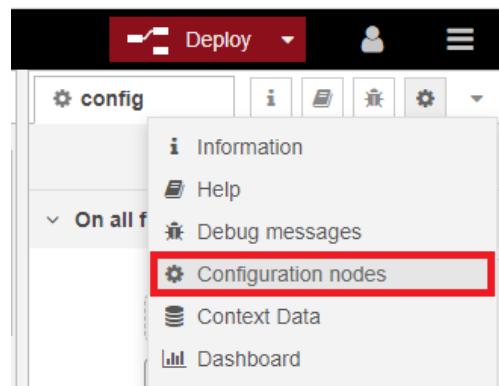
```
[{"id": "9e58624.7faaba", "type": "mqtt_out", "z": "c02b79b2.501998", "name": "", "topic": "esp32/led", "qos": "", "retain": "", "broker": "10e78a89.5b4fd5", "x": 740, "y": 520, "wires": []}, {"id": "abf7079a.653be8", "type": "mqtt_in", "z": "c02b79b2.501998", "name": "", "topic": "esp32/temperature", "qos": "2", "broker": "10e78a89.5b4fd5", "x": 430, "y": 300, "wires": [{"cc79021b.9a751": "46e7770d.86d9e8"}]}, {"id": "83cf37cf.c76988", "type": "ui_switch", "z": "c02b79b2.501998", "name": "", "label": "LED", "group": "61285987.c20328", "order": 0, "width": 0, "height": 0, "passthru": true, "decouple": false, "topic": "", "style": "", "onvalue": "on", "onvalueType": "str", "onicon": "", "oncolor": "", "offvalue": "off", "offvalueType": "str", "officon": "", "offcolor": "", "x": 590, "y": 520, "wires": [{"9e58624.7faaba": "cc79021b.9a751"}]}, {"id": "cc79021b.9a751", "type": "debug", "z": "c02b79b2.501998", "name": "", "active": true, "tosidebar": true, "console": false, "tostatus": false, "complete": false, "x": 630, "y": 260, "wires": []}, {"id": "dd25cb97.5921d8", "type": "mqtt_in", "z": "c02b79b2.501998", "name": "", "topic": "esp32/led/toggle", "qos": "2", "broker": "10e78a89.5b4fd5", "x": 420, "y": 480, "wires": [{"83cf37cf.c76988": "b9659d3c.bf3d3"}]}, {"id": "b9659d3c.bf3d3", "type": "debug", "z": "c02b79b2.501998", "name": "", "active": true, "tosidebar": true, "console": false, "tostatus": false, "complete": false, "x": 630, "y": 740, "wires": []}]
```

The following nodes should load in your flow:



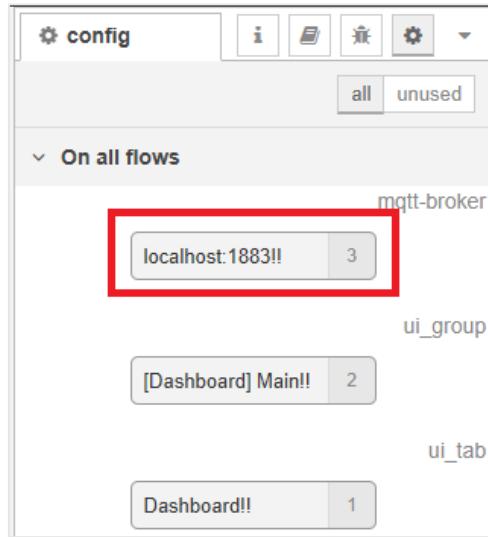
Now, you need to add your MQTT broker username and password.

At the top right corner click on the little arrow and then select **Configuration nodes**.

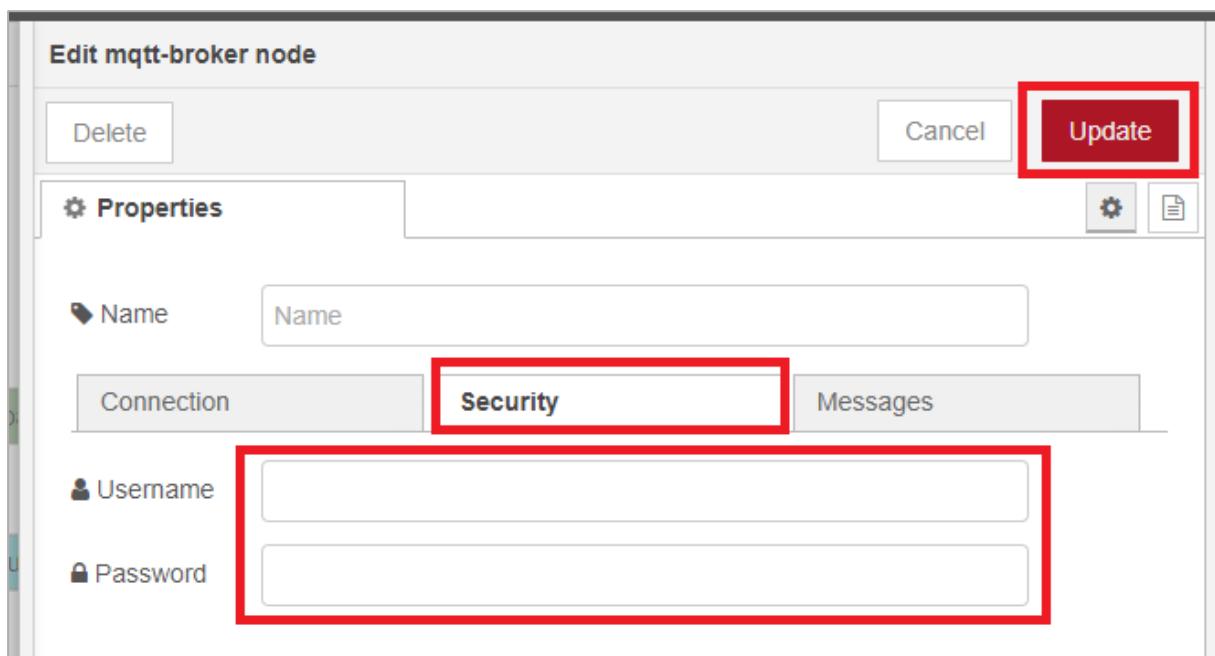


There should be a node called **localhost:1883** that refers to the MQTT broker.

Double-click on that node.



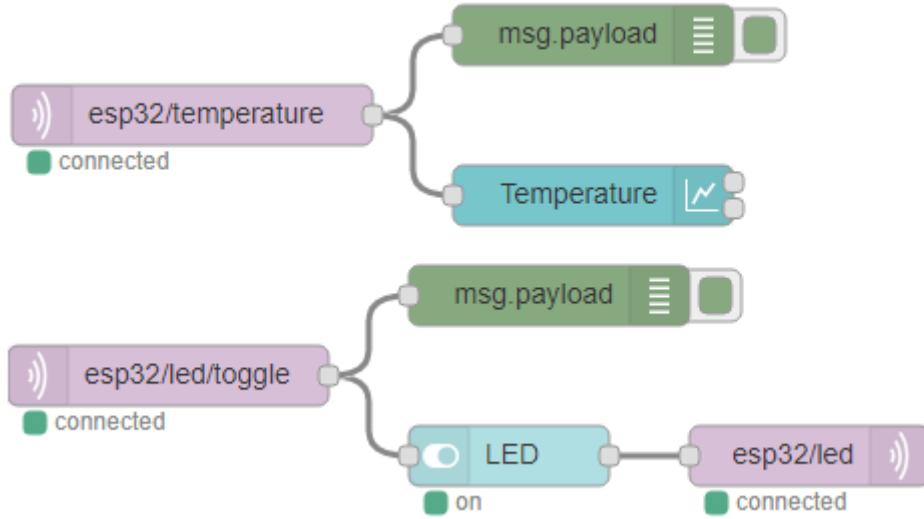
Then, select the **Security** tab and enter the broker username and password. Then, click on **Update**.



After making all changes to your flow, click the **Deploy** button to save all the changes.



If you inserted the right MQTT broker username and password, you should see a "connected" message under your MQTT nodes.

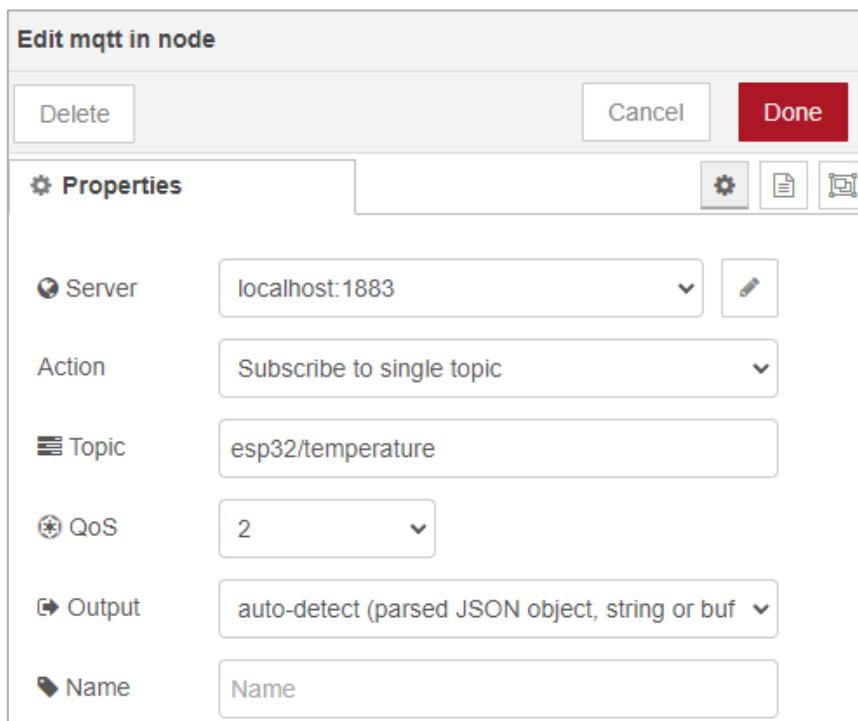


Understanding the Flow

Let's take a look at each node to understand how the flow works.

MQTT in node

The first node is a subscribe MQTT node. If you double-click the node, the following window opens.



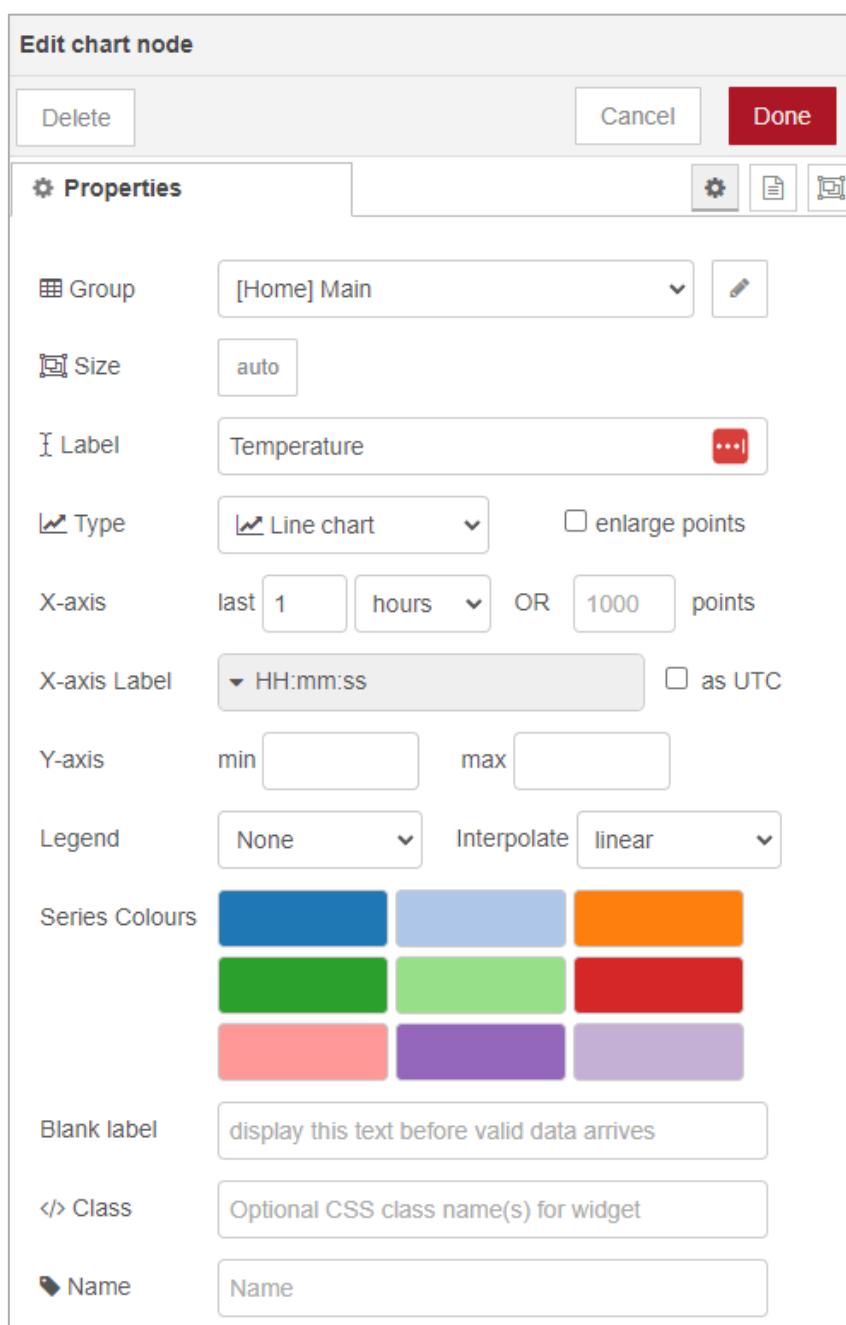
In this node, you can write the topic you want to be subscribed to. In this case, we're subscribing to the **esp32/temperature** topic to receive temperature readings.

The “Server” field corresponds to the MQTT broker IP address. We’re running the Mosquitto broker and Node-RED on the same Raspberry Pi. So, we use the localhost.

The “QoS” field corresponds to the MQTT Quality of Service level. Finally, you can edit the “Name” field with a name for your node.

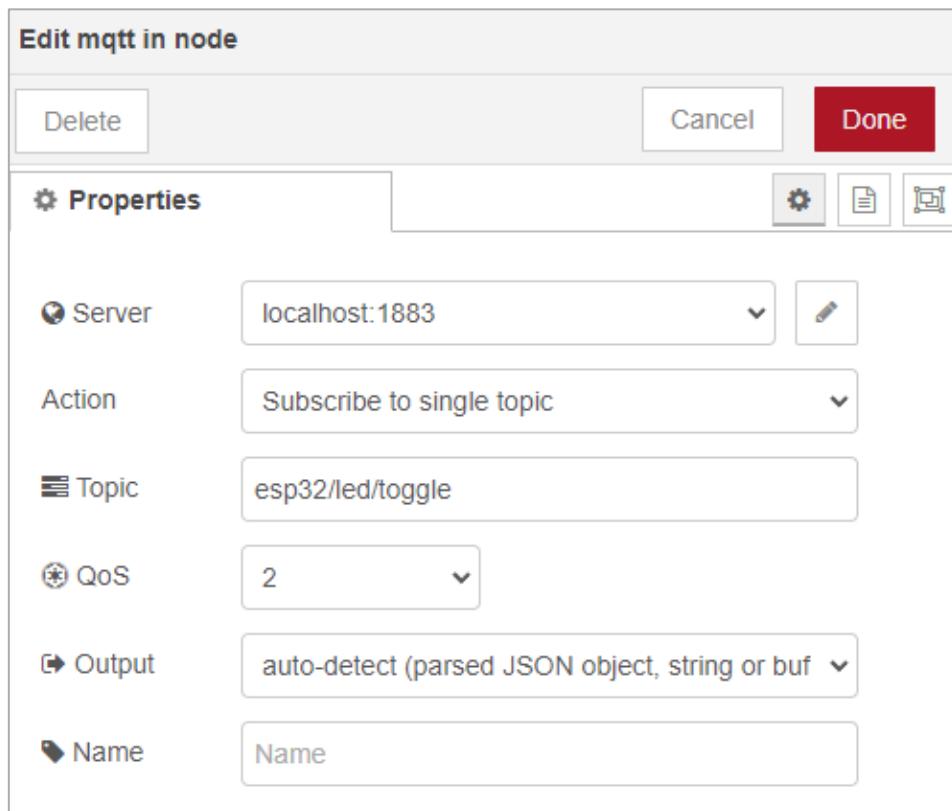
Chart node

When we receive temperature readings on the **esp32/temperature** topic, we want to display them on a chart. That’s what we do in the following node. If you double-click the chart node, you can edit its properties.



MQTT in node

Next, we have another subscribe MQTT node. This node is subscribed to the **esp32/led/toggle** topic to receive the information to turn the LED on or off. We can either receive an “on” or “off” message.



Switch node

To control the LED, you have two options:

- You can toggle the Node-RED Dashboard switch. In this scenario, when the switch is toggled, it publishes an MQTT message on the **esp32/led** topic to turn the LED on or off;
- Or you can press the physical pushbutton connected to the ESP32. After a button press, the ESP32 publishes a message on the **esp32/led/toggle** topic. The Node-RED Dashboard switch is subscribed to that topic, and it will be controlled accordingly to the received message.

Edit switch node

Delete Cancel Done

Properties

Group	[Home] Main	<input type="button" value=""/>
Size	auto	<input type="button" value=""/>
Label	LED	<input type="button" value=""/>
Tooltip	optional tooltip	<input type="button" value=""/>
Icon	Default	<input type="button" value=""/>
→ Pass through msg if payload matches valid state: <input checked="" type="checkbox"/>		
<input checked="" type="checkbox"/> When clicked, send:		
On Payload	a_z	on
Off Payload	a_z	off
Topic	a_z	<input type="button" value=""/>
</> Class	Optional CSS class name(s) for widget	
Name	<input type="button" value=""/>	

MQTT out node

After this, we have an MQTT publish Node to publish the message from the switch on the **esp32/led** topic.

Edit mqtt out node

Delete Cancel Done

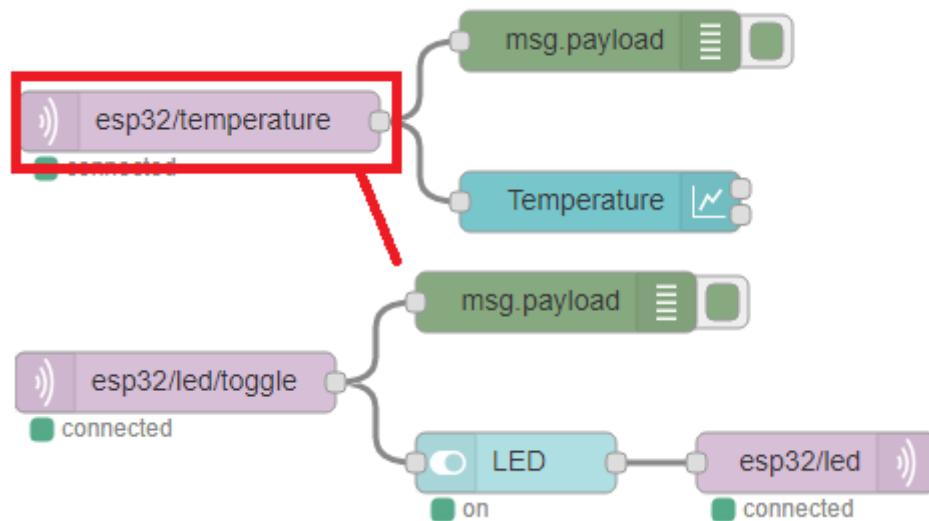
node properties

Server	localhost:1883	<input type="button" value=""/>
Topic	esp32/led	<input type="button" value=""/>
QoS	<input type="button" value=""/>	Retain <input type="button" value=""/>
Name	Name	

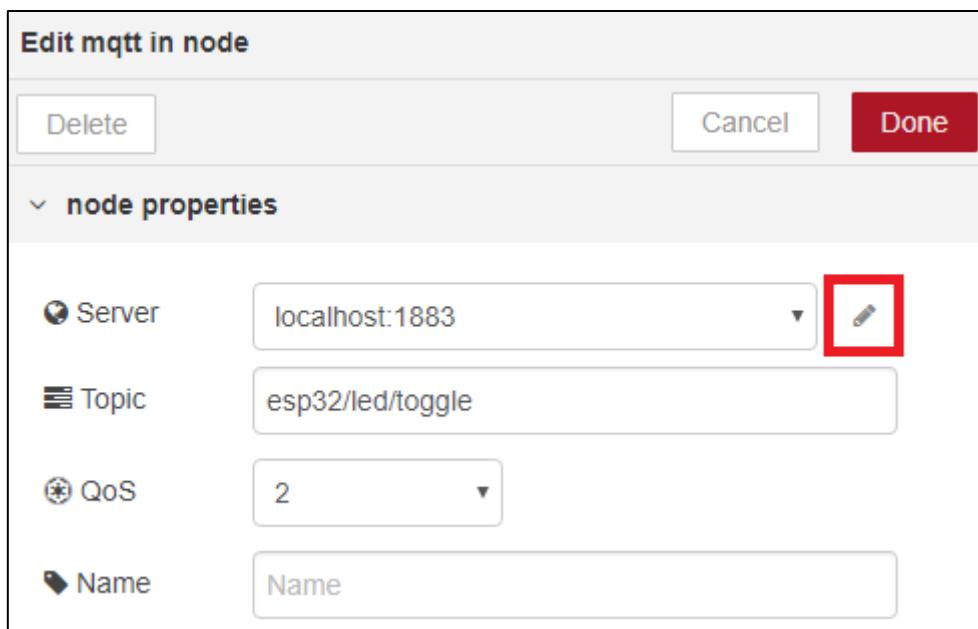
The sequence and logic of these nodes are easier to understand if you also look at the MQTT diagram at the beginning of this Unit.

MQTT broker settings

We're assuming that Node-RED and MQTT broker are installed on the same Raspberry Pi. If you're using another MQTT broker, you need to double-click one of the MQTT nodes:



Click the Edit button in the "Server" field:



Type your MQTT server IP address and port number.

Edit mqtt in node > **Edit mqtt-broker node**

[Delete](#) [Cancel](#) **Update**

Name [...](#)

Connection **Security** **Messages**

Server **localhost** Port **1883**

Enable secure (SSL/TLS) connection

Client ID

Keep alive time (s) **60** Use clean session

Use legacy MQTT 3.1 support

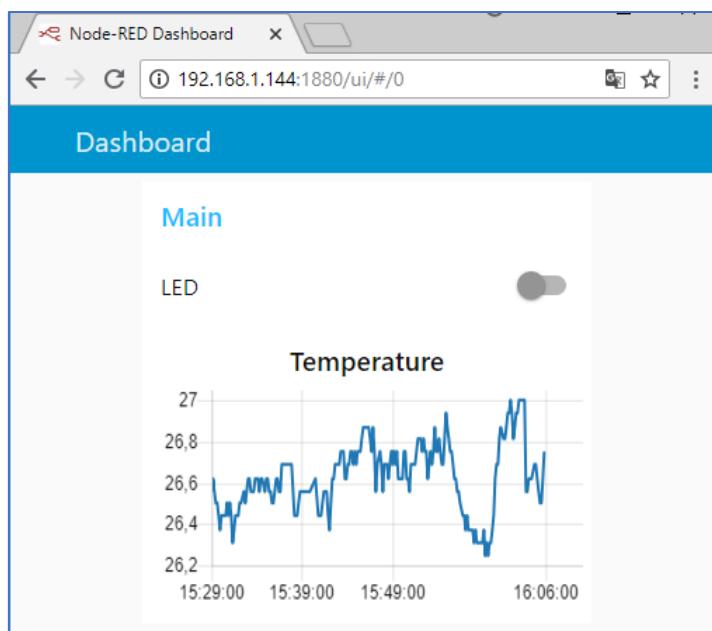
Note: since the Raspberry Pi is running the Mosquitto broker locally, we can leave the Server “localhost” and Port “1883”. You might need to change those settings and update them in your specific use case so that Node-RED can establish an MQTT connection with your broker.

Node-RED Dashboard

Now, your Node-RED application is ready. To access Node-RED Dashboard and see how your application looks, access any browser in your local network and type:

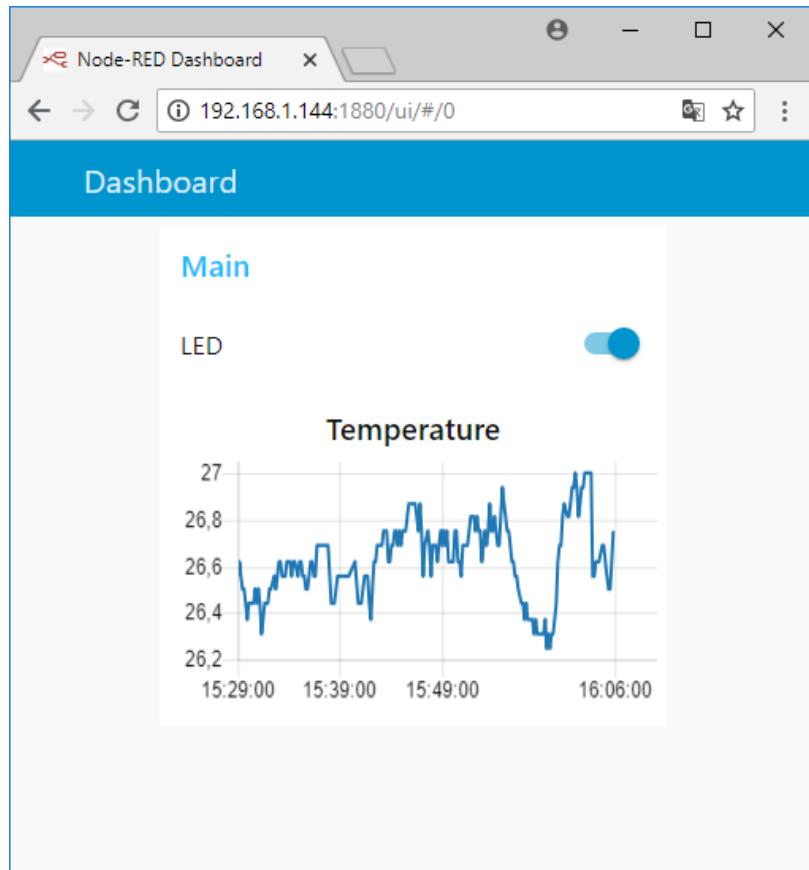
`http://Your_RPi_IP_address:1880/ui`

Your application should look as shown in the following figure.

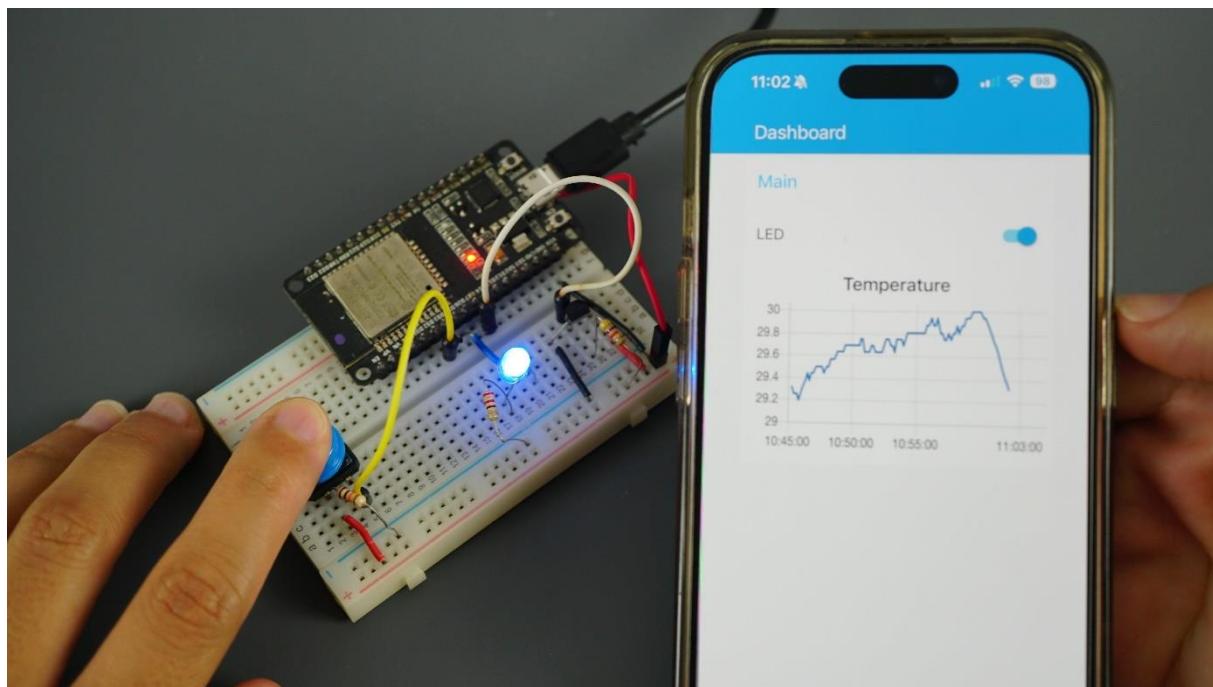


Demonstration

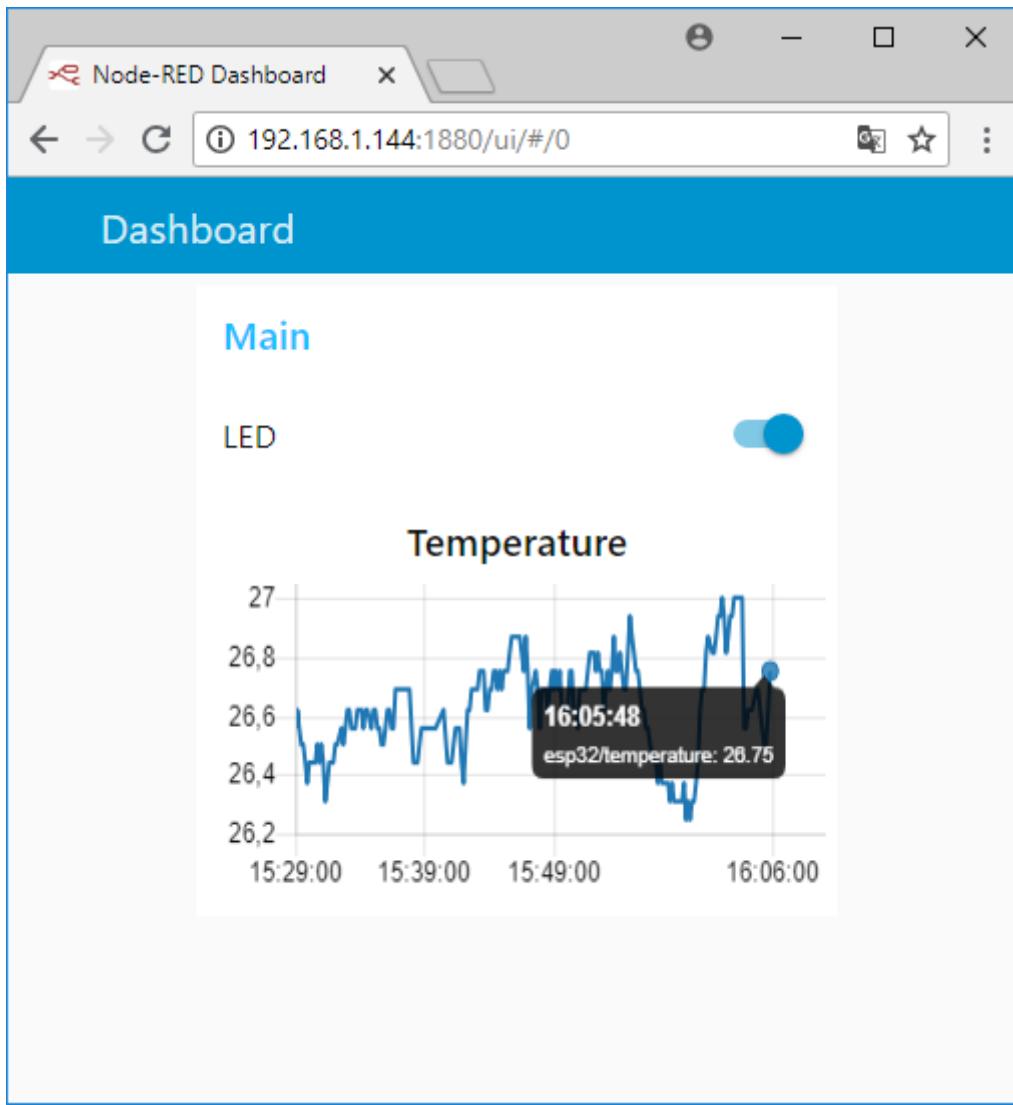
You can turn the LED on or off by pressing the switch.



If you press the pushbutton to change the LED state, the Node-RED dashboard button automatically updates its state.



Finally, new DS18B20 temperature readings are displayed in the chart every 10 seconds.



Wrapping Up

This Unit illustrates how you can interface the ESP32 with Node-RED using MQTT. We've shown you how to publish on a topic to control the ESP32 and subscribe to a particular topic to receive messages and temperature readings.

Now, you can easily control your ESP32 through your home automation Node-RED Dashboard.

We have an eBook dedicated to Home Automation and IoT using Node-RED and other platforms that you may like: [SMART HOME with Raspberry Pi, ESP32, and ESP8266](#).

MODULE 10

**ESP-NOW Communication
Protocol**

10.1 – ESP-NOW: Getting Started



In this Module, you'll learn how to use ESP-NOW to exchange data between ESP32 boards. 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.

Introducing ESP-NOW

Stating 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.*”

ESP NOW

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 straightforward to implement, and it is excellent 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 for building 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: there isn't such thing as “sender/master” and “receiver/slave” in the ESP-NOW documentation. 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”.

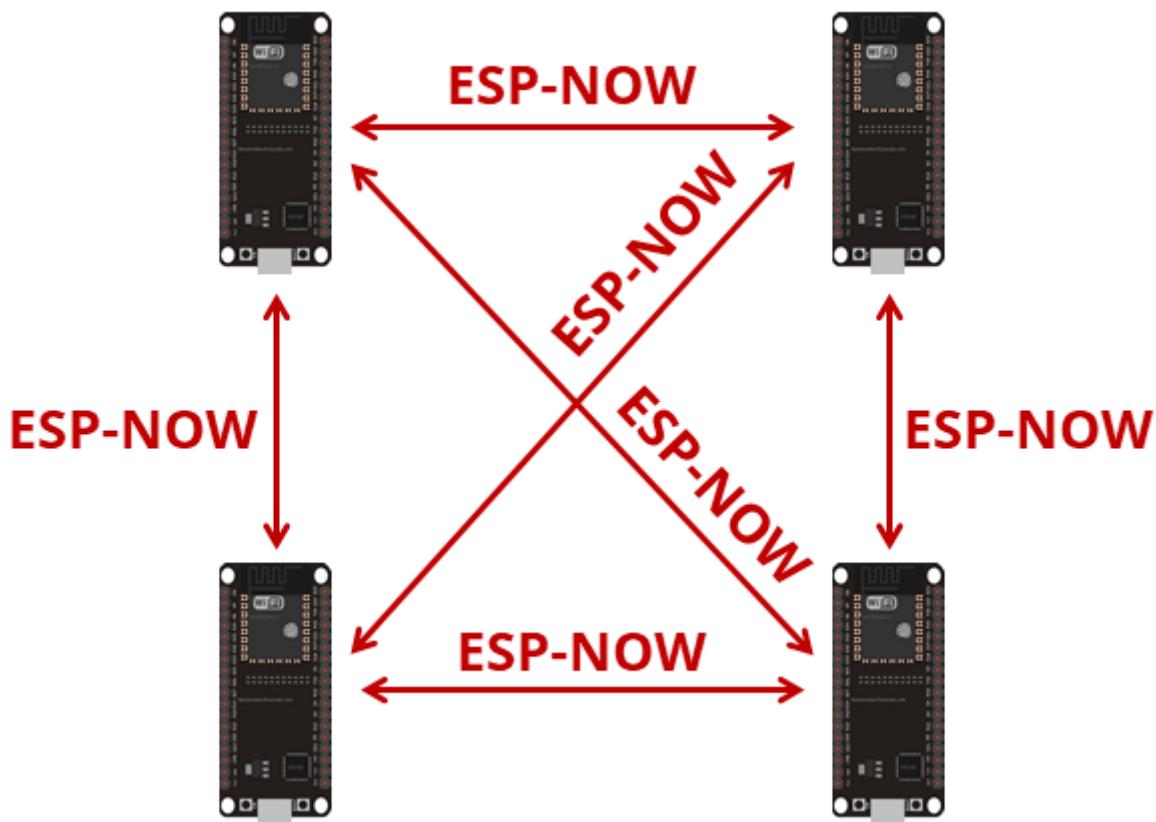
ESP-NOW Two-Way Communication

With ESP-NOW, each board can be a sender and a receiver at the same time. So, you can establish a two-way communication between boards.

For example, you can have two boards communicating with each other.



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 for building a network in which you can have several ESP32 boards exchanging data with each other.

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 information to.

Each ESP32 has a unique MAC Address, and that's how we identify each board to send data to it using ESP-NOW.

To get your board's MAC Address, upload the following code.

- [Click here to download the code.](#)

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

void readMacAddress(){
    uint8_t baseMac[6];
    esp_err_t ret = esp_wifi_get_mac(WIFI_IF_STA, baseMac);
    if (ret == ESP_OK) {
        Serial.printf("%02x:%02x:%02x:%02x:%02x:%02x\n",
                      baseMac[0], baseMac[1], baseMac[2],
                      baseMac[3], baseMac[4], baseMac[5]);
    } else {
        Serial.println("Failed to read MAC address");
    }
}

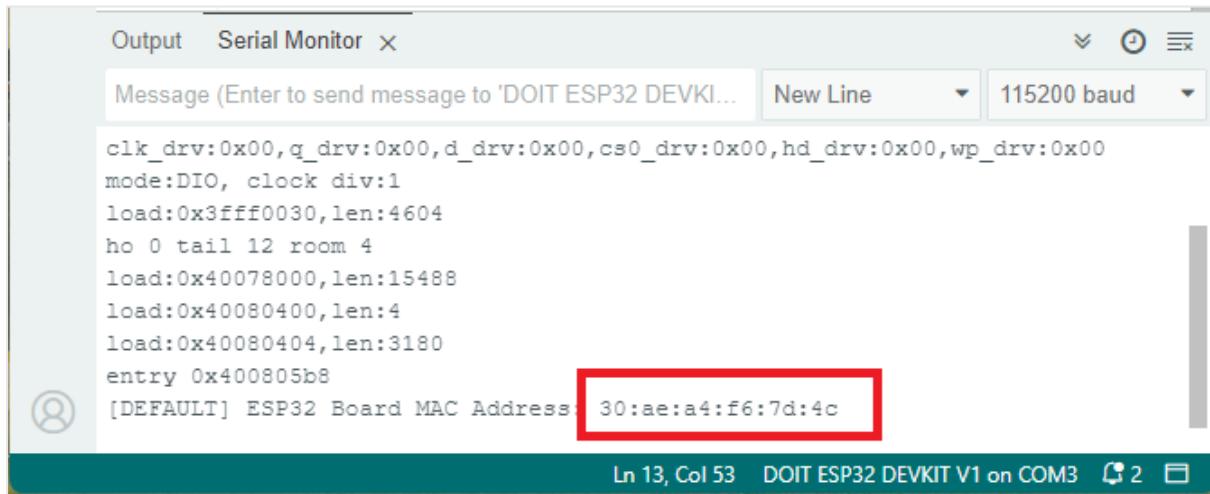
void setup(){
    Serial.begin(115200);

    WiFi.mode(WIFI_STA);
    WiFi.STA.begin();

    Serial.print("[DEFAULT] ESP32 Board MAC Address: ");
    readMacAddress();
}

void loop(){
}
```

After uploading the code, open the serial monitor at a baud rate of 115200 and press the ESP32 RST/EN button. The MAC address will be printed as follows:



The screenshot shows the Arduino Serial Monitor window. The title bar says "Output" and "Serial Monitor". The message area displays the following text:

```
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
[DEFAULT] ESP32 Board MAC Address: 30:ae:a4:f6:7d:4c
```

The MAC address "30:ae:a4:f6:7d:4c" is highlighted with a red rectangle. The status bar at the bottom shows "Ln 13, Col 53 DOIT ESP32 DEVKIT V1 on COM3" and some icons.

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).

Note: a structure is a user-defined datatype that allows you to combine data of different types. It is similar to an Array, but an array holds data of similar type only. The structure, on the other hand, can store data of any type.

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 sent;
3. Add a peer device (the receiver). For this, you need to know the receiver's 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.

ESP-NOW Useful Functions

Here's a summary of the most essential ESP-NOW functions:

Function	Description
<code>esp_now_init()</code>	Initializes ESP-NOW. You must initialize Wi-Fi before initializing ESP-NOW.

<code>esp_now_add_peer()</code>	Call this function to pair a device and pass as an argument the peer MAC address.
<code>esp_now_send()</code>	Send data with ESP-NOW.
<code>esp_now_register_send_cb()</code>	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.
<code>esp_now_register_rcv_cb()</code>	Register a callback function that is triggered upon receiving data. When data is received via ESP-NOW, a function is called.

Note: 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.

- [Click here to download the code.](#)

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

// REPLACE WITH YOUR RECEIVER MAC Address
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

esp_now_peer_info_t peerInfo;

// 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
```

```

struct_message myData;

// callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
    Serial.print("\r\nLast Packet Send Status:\t");
    Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Send Success" : "Send Fail");
}

void setup() {
    // Init Serial Monitor
    Serial.begin(115200);

    // Set device as a Wi-Fi Station
    WiFi.mode(WIFI_STA);

    // Init ESP-NOW
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    // Once ESPNow is successfully Init, we will register for Send CB to
    // get the status of Trasnmitted packet
    esp_now_register_send_cb(OnDataSent);

    // 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;
    }
}

void loop() {
    // Set values to send
    strcpy(myData.a, "THIS IS A CHAR");
    myData.b = random(1,20);
    myData.c = 1.2;
    myData.d = false;

    // Send message via ESP-NOW
    esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &myData,
                                    sizeof(myData));

    if (result == ESP_OK) {
        Serial.println("Sent with success");
    }
    else {
        Serial.println("Error sending the data");
    }
    delay(2000);
}

```

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[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
```

In our case, the receiver MAC address is: `30:AE:A4:F6:7D:4C`. So, the previous line will look as follows:

```
uint8_t broadcastAddress[] = {0x30, 0xAE, 0xA4, 0xF6, 0x7D, 0x4C};
```

Create a variable of type `esp_now_peer_info_t` to store the information about the peer (the ESP32 board that will receive the messages).

```
esp_now_peer_info_t peerInfo;
```

Then, create a structure that contains the type of data we want to send. We called this structure `struct_message`, and it contains four different variable types. You can change this to send whatever variable types you want.

```
// 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;
```

Then, create a new variable of type `struct_message` that is called `myData` that will store the values of the variables.

```
// Create a struct_message called myData
struct_message myData;
```

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 sent or not.

```
// callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
```

```
Serial.print("\r\nLast Packet Send Status:\t");
Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Send Success" : "Send Fail");
}
```

In the `setup()`, initialize the serial monitor for debugging purposes:

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

Set the device as a Wi-Fi station. This is needed to use ESP-NOW:

```
// Set device as a Wi-Fi Station
WiFi.mode(WIFI_STA);
```

Initialize ESP-NOW:

```
// Init 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 any other type of data.

Finally, send the message as follows:

```
// Send message via ESP-NOW
esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &myData,
                                sizeof(myData));

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.

- [Click here to download the code.](#)

```
#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;

// callback function that will be executed when data is received
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
    memcpy(&myData, incomingData, sizeof(myData));
    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();
}

void setup() {
    // Initialize Serial Monitor
    Serial.begin(115200);

    // Set device as a Wi-Fi Station
    WiFi.mode(WIFI_STA);

    // Init ESP-NOW
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    // Once ESPNow is successfully Init, we will register for recv CB to
    // get recv packer info
    esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv));
}

void loop() {
}

```

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, int len) {
```

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 sender ESP32. 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()`, initialize the Serial Monitor.

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

Set the device as a Wi-Fi Station.

```
WiFi.mode(WIFI_STA);
```

Initialize ESP-NOW:

```
// Init 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.

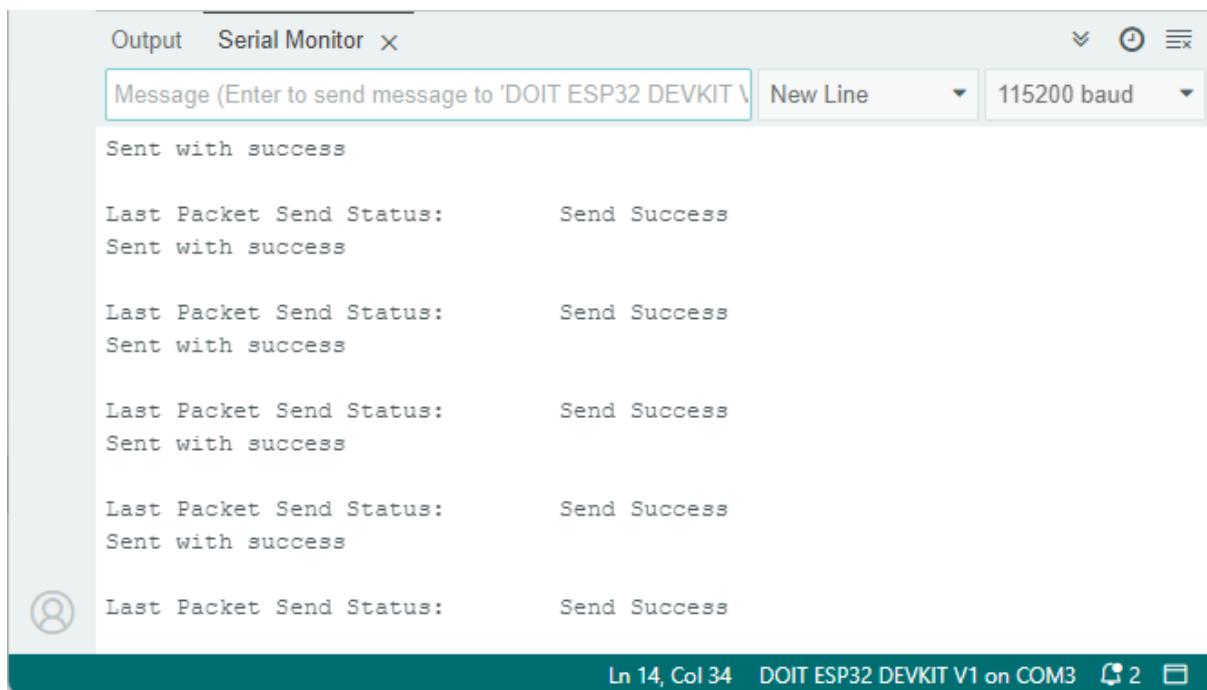
```
esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv));
```

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).

Serial Monitor X

Message (Enter to send message to 'DOIT ESP32 DEVKIT V1') New Line 115200 baud

```
Bytes received: 44
Char: THIS IS A CHAR
Int: 14
Float: 1.20
Bool: 0

Bytes received: 44
Char: THIS IS A CHAR
Int: 16
Float: 1.20
Bool: 0

Bytes received: 44
Char: THIS IS A CHAR
Int: 2
```

Ln 53, Col 1 DOIT ESP32 DEVKIT V1 on COM5

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.



Wrapping Up

We tried to keep this example 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 > ESP_NOW** and choose one of the example sketches.

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 be a sender and receiver at the same time (two-way communication), and one board can send data to multiple boards (one-to-many) and also receive data from multiple boards (many-to-one). These topics will be covered in the next units.

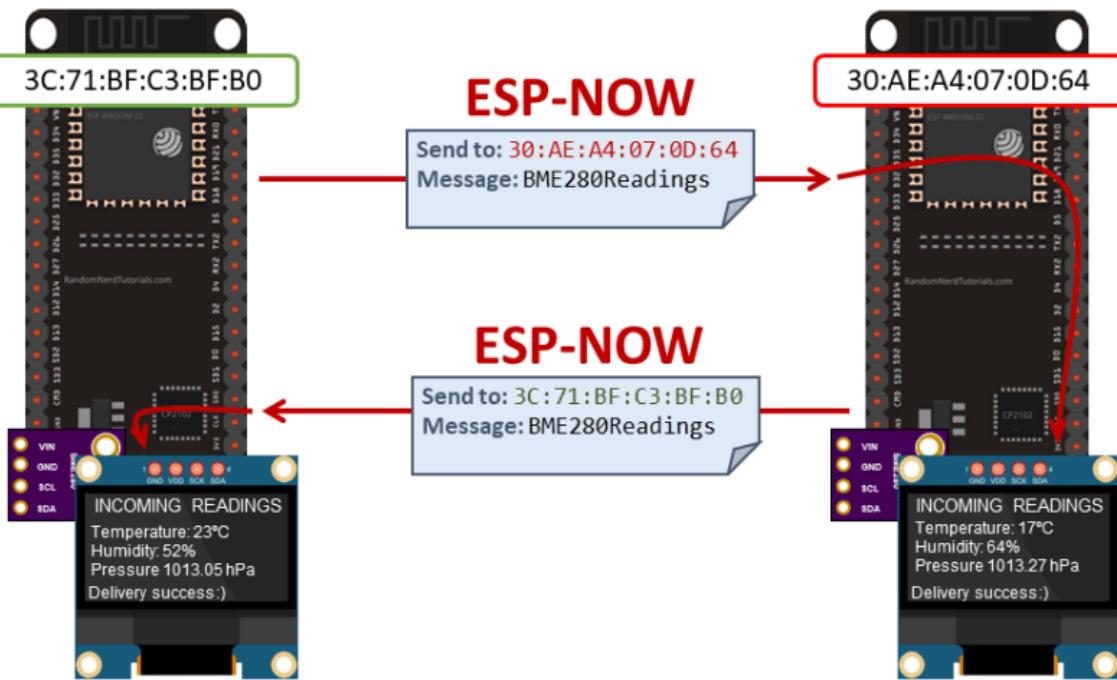
10.2 - ESP-NOW Two-Way Communication Between ESP32



In this Unit, you'll learn to establish a two-way communication between two ESP32 boards using ESP-NOW communication protocol. As an example, two ESP32 boards will exchange sensor readings (with a range in open field up to 220 meters ~ 722 feet).

Project Overview

The following diagram shows a high-level overview of the project we'll build.



- In this project, we'll have two ESP32 boards. Each board is connected to an OLED display and a BME280 sensor;
- Each board gets temperature, humidity, and pressure readings from their corresponding sensors;
- Each board sends its readings to the other board via ESP-NOW;
- When a board receives the readings, it displays them on the OLED display;
- After sending the readings, the board displays on the OLED if the message was sent successfully;
- Each board needs to know the other board's MAC address to send the message.
- In this example, we're using a two-way communication between two boards, but you can add more boards to this setup, and have all boards communicating with each other.

Install Libraries

Install the following libraries in your Arduino IDE if you haven't already. These libraries can be installed through the Arduino Library Manager. Go to **Sketch > Include Library > Manage Libraries** and search for the libraries' names.

- OLED libraries: [Adafruit_SSD1306 library](#) and [Adafruit_GFX library](#)

- BME280 libraries: [Adafruit_BME280_library](#) and [Adafruit Unified Sensor library](#)

Parts Required

For this tutorial, you need the following parts:

- 2x [ESP32 development boards](#)
- 2x [BME280 sensors](#)
- 2x [0.96 inch OLED displays](#)
- [Breadboard](#)
- [Jumper wires](#)

Getting the Boards MAC Address

To send messages between each board, we need to know their MAC addresses.

Each board has a unique MAC address.

Upload the following code to each of your boards to get their MAC address.

- [Click here to download the code.](#)

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

void readMacAddress(){
    uint8_t baseMac[6];
    esp_err_t ret = esp_wifi_get_mac(WIFI_IF_STA, baseMac);
    if (ret == ESP_OK) {
        Serial.printf("%02x:%02x:%02x:%02x:%02x:%02x\n",
                      baseMac[0], baseMac[1], baseMac[2],
                      baseMac[3], baseMac[4], baseMac[5]);
    }
    else {
        Serial.println("Failed to read MAC address");
    }
}

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

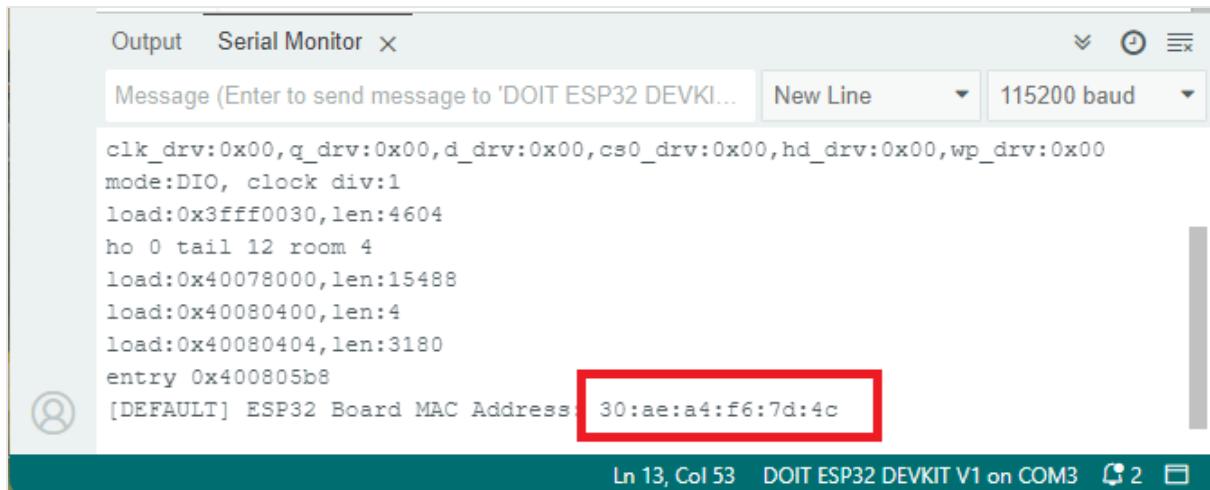
    WiFi.mode(WIFI_STA);
    WiFi.STA.begin();

    Serial.print("[DEFAULT] ESP32 Board MAC Address: ");
    readMacAddress();
}
```

```
void loop(){
```

```
}
```

After uploading the code, open the serial monitor at a baud rate of 115200 and press the ESP32 RST/EN button. The MAC address will be printed as follows:

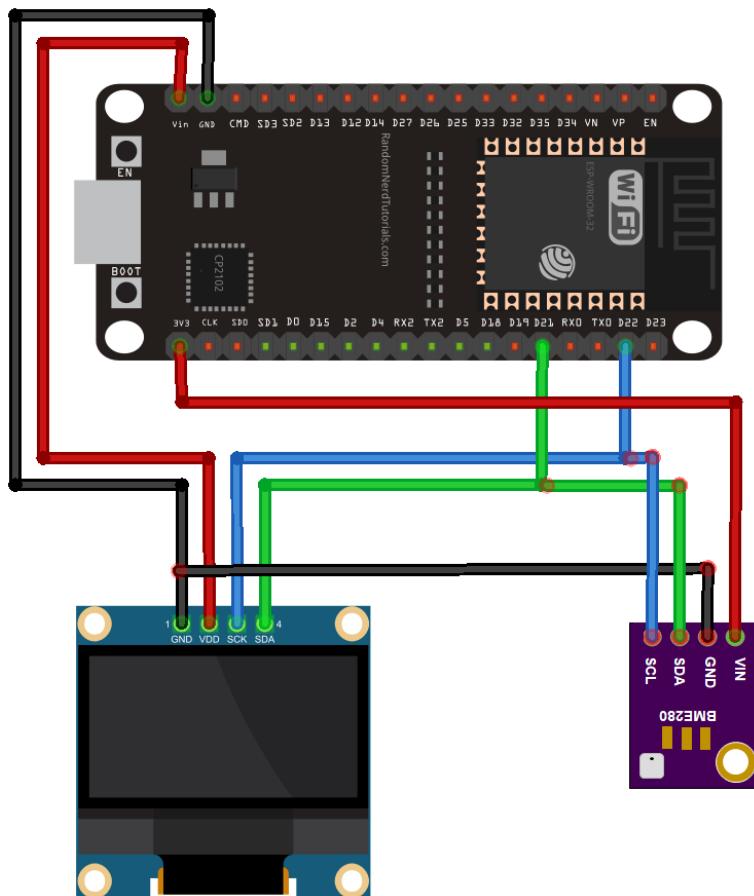


Write down the MAC address of each board to clearly identify them. For example:



Wiring the Circuit

Wire an OLED display and a BME280 sensor to each ESP32 board. Follow the next schematic diagram.



You can use the following table as a reference when wiring the BME280 sensor.

BME280	ESP32
VIN	3.3V
GND	GND
SCL	GPIO 22
SDA	GPIO 21

You can also follow the next table to wire the OLED display to the ESP32.

OLED Display	ESP32
GND	GND

VCC	VIN
SCL	GPIO 22
SDA	GPIO 21

Two-Way Communication ESP-NOW Code

Upload the following code to each of your boards. Before uploading the code, you need to enter the MAC address of the other board (the board you're sending data to).

- [Click here to download the code.](#)

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

#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels

// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

Adafruit_BME280 bme;

// REPLACE WITH THE MAC Address of your receiver
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

// Define variables to store BME280 readings to be sent
float temperature;
float humidity;
float pressure;

// Define variables to store incoming readings
float incomingTemp;
float incomingHum;
float incomingPres;

// Variable to store if sending data was successful
String success;

//Structure example to send data
//Must match the receiver structure
typedef struct struct_message {
    float temp;
    float hum;
```

```

    float pres;
} struct_message;

// Create a struct_message called BME280Readings to hold sensor readings
struct_message BME280Readings;

// Create a struct_message to hold incoming sensor readings
struct_message incomingReadings;

esp_now_peer_info_t peerInfo;

// Callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
    Serial.print("\r\nLast Packet Send Status:\t");
    Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" : "Delivery
Fail");
    if (status ==0){
        success = "Delivery Success :)";
    }
    else{
        success = "Delivery Fail :(";
    }
}

// Callback when data is received
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
    memcpy(&incomingReadings, incomingData, sizeof(incomingReadings));
    Serial.print("Bytes received: ");
    Serial.println(len);
    incomingTemp = incomingReadings.temp;
    incomingHum = incomingReadings.hum;
    incomingPres = incomingReadings.pres;
}

void setup() {
    // Init Serial Monitor
    Serial.begin(115200);

    // Init BME280 sensor
    bool status = bme.begin(0x76);
    if (!status) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }

    // Init OLED display
    if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
        Serial.println(F("SSD1306 allocation failed"));
        for(;;);
    }

    // Set device as a Wi-Fi Station
    WiFi.mode(WIFI_STA);

    // Init ESP-NOW
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    // Once ESPNow is successfully Init, we will register for Send CB to
}

```

```

// get the status of Trasnmitted packet
esp_now_register_send_cb(OnDataSent);

// 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;
}
// Register for a callback function that will be called when data is received
esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv));
}

void loop() {
    getReadings();

    // Set values to send
    BME280Readings.temp = temperature;
    BME280Readings.hum = humidity;
    BME280Readings.pres = pressure;

    // Send message via ESP-NOW
    esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &BME280Readings,
                                    sizeof(BME280Readings));

    if (result == ESP_OK) {
        Serial.println("Sent with success");
    }
    else {
        Serial.println("Error sending the data");
    }
    updateDisplay();
    delay(10000);
}

void getReadings(){
    temperature = bme.readTemperature();
    humidity = bme.readHumidity();
    pressure = (bme.readPressure() / 100.0F);
}

void updateDisplay(){
    // Display Readings on OLED Display
    display.clearDisplay();
    display.setTextSize(1);
    display.setTextColor(WHITE);
    display.setCursor(0, 0);
    display.println("INCOMING READINGS");
    display.setCursor(0, 15);
    display.print("Temperature: ");
    display.print(incomingTemp);
    display.cp437(true);
    display.write(248);
    display.print("C");
    display.setCursor(0, 25);
    display.print("Humidity: ");
    display.print(incomingHum);
    display.print("%");
}

```

```

display.setCursor(0, 35);
display.print("Pressure: ");
display.print(incomingPres);
display.print("hPa");
display.setCursor(0, 56);
display.print(success);
display.display();

// Display Readings in Serial Monitor
Serial.println("INCOMING READINGS");
Serial.print("Temperature: ");
Serial.print(incomingReadings.temp);
Serial.println(" °C");
Serial.print("Humidity: ");
Serial.print(incomingReadings.hum);
Serial.println(" %");
Serial.print("Pressure: ");
Serial.print(incomingReadings.pres);
Serial.println(" hPa");
Serial.println();
}

```

How Does the Code Work?

We've covered in great detail how to interact with the OLED display and with the BME280 sensor in previous units. Here, we'll just take a look at the relevant parts when it comes to ESP-NOW.

The code is well commented so that you understand what each line of code does.

To use ESP-NOW, you need to include the next libraries.

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

In the next line, insert the MAC address of the receiver board:

```
// REPLACE WITH THE MAC Address of your receiver
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
```

Create a variable of type `esp_now_peer_info_t` to store the information about the peer (the ESP32 board that will receive the messages).

```
esp_now_peer_info_t peerInfo;
```

Create variables to store temperature, humidity and pressure readings from the BME280 sensor. These readings will be sent to the other board:

```
// Define variables to store BME280 readings to be sent
float temperature;
float humidity;
```

```
float pressure;
```

Create variables to store the sensor readings coming from the other board:

```
float incomingTemp;
float incomingHum;
float incomingPres;
```

The following variable will store a success message if the readings are sent successfully.

```
String success;
```

Create a structure that stores humidity, temperature and pressure readings.

```
typedef struct struct_message {
    float temp;
    float hum;
    float pres;
} struct_message;
```

Then, you need to create two instances of that structure—one to receive the readings and another to store the readings to be sent.

The `BME280Readings` will store the readings to be sent.

```
struct_message BME280Readings;
```

The `incomingReadings` will store the data coming from the other board.

```
struct_message incomingReadings;
```

Then, we need to create two callback functions. One will be called when data is sent, and another will be called when data is received.

OnDataSent() callback function

The `OnDataSent()` function will be called when new data is sent. This function simply prints if the message was successfully sent or not. If the message is sent successfully, the status variable returns 0, so we can set our success message to “Sent Success”:

```
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
    Serial.print("\r\nLast Packet Send Status:\t");
    Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Sent Success" : "Sent Fail");
    if (status == 0){
        success = "Delivery Success :)" ;
```

```
    }
    else{
        success = "Delivery Fail :(";
    }
}
```

OnDataRecv() callback function

The OnDataRecv() function will be called when a new packet arrives.

```
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
```

We save the new packet in the incomingReadings structure we've created previously:

```
memcpy(&incomingReadings, incomingData, sizeof(incomingReadings));
```

We print the message length on the serial monitor. You can only send 250 bytes in each packet.

```
Serial.print("Bytes received: ");
Serial.println(len);
```

Then, store the incoming readings in their corresponding variables. To access the temperature variable inside incomingReadings structure, you just need to do call incomingReadings.temp as follows:

```
incomingTemp = incomingReadings.temp;
```

The same process is done for the other variables:

```
incomingHum = incomingReadings.hum;
incomingPres = incomingReadings.pres;
```

setup()

In the setup(), initialize ESP-NOW.

```
if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
}
```

Then, register for the OnDataSent() callback function.

```
esp_now_register_send_cb(OnDataSent);
```

In order to send data to another board, you need to pair it as a peer. The following lines register and add a new peer.

```
// 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;
}
```

Register for the `OnDataRecv()` callback function.

```
esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv));
```

loop()

In the `loop()`, we call the `getReadings()` function that is responsible for getting new temperature readings from the sensor. That function is created after the `loop()`.

```
getReadings();
```

After getting new temperature, humidity and pressure readings, we update our `BME280Reading` structure with those new values:

```
BME280Readings.temp = temperature;
BME280Readings.hum = humidity;
BME280Readings.pres = pressure;
```

Then, we can send the `BME280Readings` structure via ESP-NOW:

```
esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &BME280Readings,
                                 sizeof(BME280Readings));

if (result == ESP_OK) {
    Serial.println("Sent with success");
}
else {
    Serial.println("Error sending the data");
}
```

Finally, call the `updateDisplay()` function that will update the OLED display with the readings coming from the other ESP32 board.

```
updateDisplay();
```

The `loop()` is executed every 10 seconds.

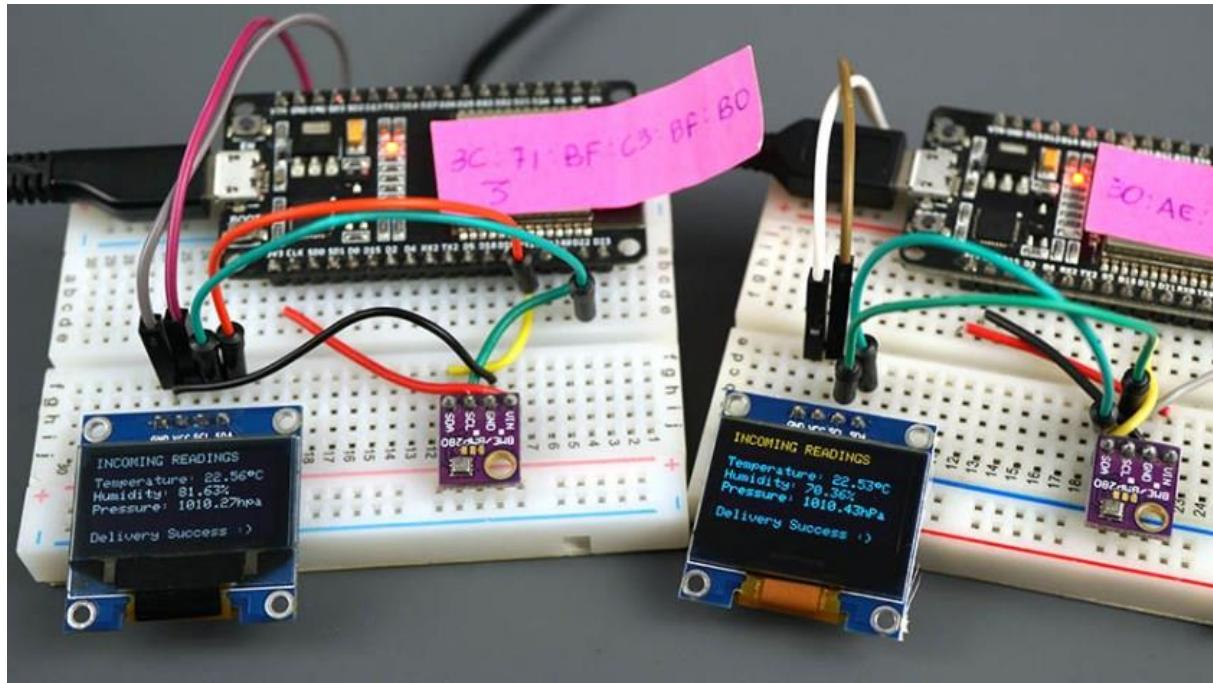
```
delay(10000);
```

That's pretty much how the code works.

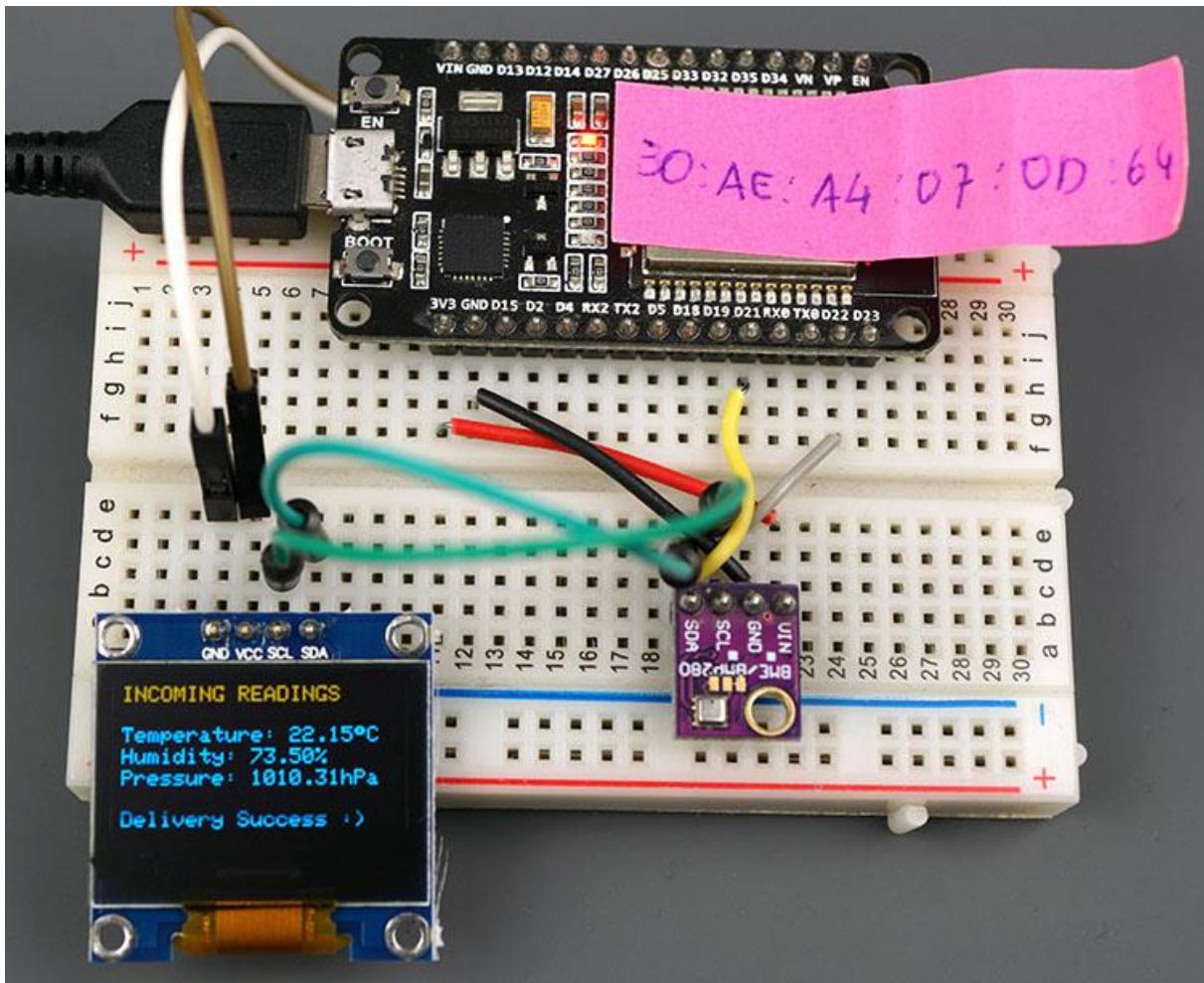
Now, upload the code to both of your boards. You just need to modify the code with the MAC address of the board you're sending data to.

Demonstration

After uploading the code to both boards, you should see the OLED displaying the sensor readings from the other board and a success delivery message.



As you can see, it's working as expected:



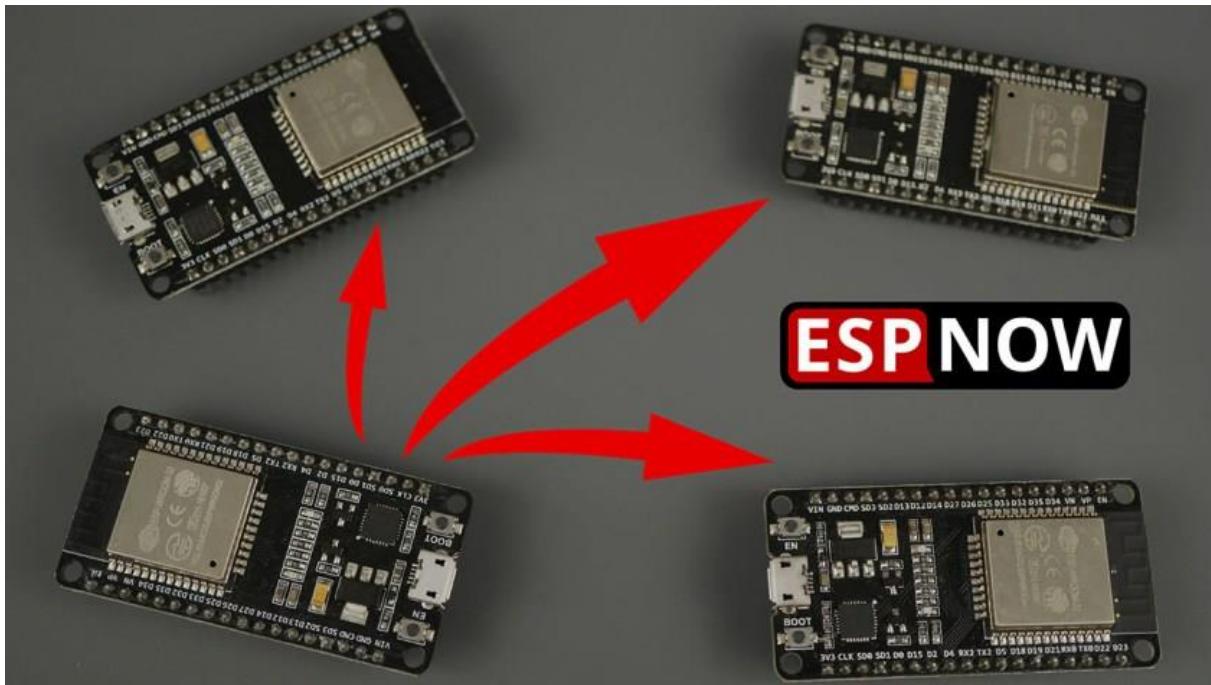
Wrapping Up

In this unit, we've shown you how to establish a two-way communication with two ESP32 boards using ESP-NOW. This is a very versatile communication protocol that can be used to send packets with up to 250 bytes.

As an example, we've shown you the interaction between two boards, but you can add many boards to your setup. You just need to know the MAC address of the board you're sending data to.

In the next Units, you'll learn how to receive data from multiple boards (many-to-one) and how to send data to multiple boards (one-to-many).

10.3 - ESP-NOW Send Data to Multiple Boards (one-to-many)



In this Unit, you'll learn how to use ESP-NOW communication protocol to send data from one ESP32 to multiple ESP32 (one-to-many configuration).

Project Overview

Here's an overview of the project we'll build:

- One ESP32 acts as a sender;
- Multiple ESP32 boards act as receivers. We tested this setup with three ESP32 boards simultaneously. You should be able to add more boards to your setup;
- The ESP32 sender receives an acknowledge message if the messages are successfully delivered. You know which boards received the message and which boards didn't;
- As an example, we'll exchange random values between the boards. You should modify this example to send commands or sensor readings, for example;
- This tutorial covers these two different scenarios:

- sending the same message to all boards;
- sending a different message to each board.

Getting the Boards' MAC Address

If you followed previous units, you know that you need to know the receiver boards' MAC address to send messages via ESP-NOW.

Upload the following code to each of your receiver boards to get their MAC addresses.

- [Click here to download the code.](#)

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

void readMacAddress(){
    uint8_t baseMac[6];
    esp_err_t ret = esp_wifi_get_mac(WIFI_IF_STA, baseMac);
    if (ret == ESP_OK) {
        Serial.printf("%02x:%02x:%02x:%02x:%02x:%02x\n",
                      baseMac[0], baseMac[1], baseMac[2],
                      baseMac[3], baseMac[4], baseMac[5]);
    } else {
        Serial.println("Failed to read MAC address");
    }
}

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

    WiFi.mode(WIFI_STA);
    WiFi.STA.begin();

    Serial.print("[DEFAULT] ESP32 Board MAC Address: ");
    readMacAddress();
}

void loop(){}
```

After uploading the code, open the serial monitor at a baud rate of 115200 and press the ESP32 RST/EN button. The MAC address will be printed as follows:



We suggest that you write down the boards' MAC addresses on a label to identify each board.

ESP32 Sender Code (ESP-NOW)

The following code sends data to multiple (three) ESP boards via ESP-NOW. You should modify the code with your receiver boards' MAC address. You should also add or delete lines of code depending on the number of receiver boards.

- [Click here to download the code.](#)

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

// REPLACE WITH YOUR ESP RECEIVER'S MAC ADDRESS
uint8_t broadcastAddress1[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
uint8_t broadcastAddress2[] = {0xFF, , , , , };
uint8_t broadcastAddress3[] = {0xFF, , , , , };

// Variable to add info about peer
esp_now_peer_info_t peerInfo;

typedef struct test_struct {
    int x;
    int y;
} test_struct;

test_struct test;

// callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
    char macStr[18];
    Serial.print("Packet to: ");
    // Copies the sender mac address to a string
    sprintf(macStr, sizeof(macStr), "%02x:%02x:%02x:%02x:%02x:%02x",
            mac_addr[0], mac_addr[1], mac_addr[2],
            mac_addr[3], mac_addr[4], mac_addr[5]);
    Serial.print(macStr);
```

```

Serial.print(" send status:\t");
Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" :
              "Delivery Fail");
}

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

  WiFi.mode(WIFI_STA);

  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }

  esp_now_register_send_cb(OnDataSent);

  // register peer
  peerInfo.channel = 0;
  peerInfo.encrypt = false;
  // register first peer
  memcpy(peerInfo.peer_addr, broadcastAddress1, 6);
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }
  // register second peer
  memcpy(peerInfo.peer_addr, broadcastAddress2, 6);
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }
  /// register third peer
  memcpy(peerInfo.peer_addr, broadcastAddress3, 6);
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }
}

void loop() {
  test.x = random(0,20);
  test.y = random(0,20);

  esp_err_t result = esp_now_send(0, (uint8_t *) &test, sizeof(test_struct));

  if (result == ESP_OK) {
    Serial.println("Sent with success");
  }
  else {
    Serial.println("Error sending the data");
  }
  delay(2000);
}

```

How Does the Code Work?

First, include the `esp_now.h` and `WiFi.h` libraries.

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

Receivers' MAC Address

Insert the receivers' MAC address. For example:

```
uint8_t broadcastAddress1[] = {0x3C, 0x71, 0xBF, 0xC3, 0xBF, 0xB0};
uint8_t broadcastAddress2[] = {0x24, 0x0A, 0xC4, 0xAE, 0xAE, 0x44};
uint8_t broadcastAddress3[] = {0x80, 0x7D, 0x3A, 0x58, 0xB4, 0xB0};
```

Then, create a structure that contains the data we want to send. We called this structure `test_struct` and it contains two integer variables. You can change this to send whatever variable types you want.

```
typedef struct test_struct {
    int x;
    int y;
} test_struct;
```

Create a new variable of type `test_struct` that's called `test` that will store the variables' values.

```
test_struct test;
```

OnDataSent() callback function

Next, define the `OnDataSent()` function. This is a callback function that will be executed when a message is sent. In this case, this function prints if the message was successfully delivered or not and for which MAC address. So, you know which boards received the message or and which boards didn't.

```
// callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
    char macStr[18];
    Serial.print("Packet to: ");
    // Copies the sender mac address to a string
    sprintf(macStr, sizeof(macStr), "%02x:%02x:%02x:%02x:%02x:%02x",
            mac_addr[0], mac_addr[1], mac_addr[2],
            mac_addr[3], mac_addr[4], mac_addr[5]);
    Serial.print(macStr);
    Serial.print(" send status:\t");
    Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" :
                  "Delivery Fail");
}
```

setup()

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, register for the `OnDataSent()` function created previously.

```
esp_now_register_send_cb(OnDataSent);
```

Add peers

After that, we need to pair with other ESP-NOW devices to send data. That's what we do in the next lines – register peers:

```
peerInfo.channel = 0;
peerInfo.encrypt = false;
// register first peer
memcpy(peerInfo.peer_addr, broadcastAddress1, 6);
if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
}
// register second peer
memcpy(peerInfo.peer_addr, broadcastAddress2, 6);
if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
}
/// register third peer
memcpy(peerInfo.peer_addr, broadcastAddress3, 6);
if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
}
```

If you want to add more peers you just need to multiply these lines and pass the peer MAC address:

```
memcpy(peerInfo.peer_addr, broadcastAddress3, 6);
if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
```

loop()

In the `loop()`, we'll send a message via ESP-NOW every 2 seconds (you can change this delay time).

Assign a value to each variable:

```
test.x = random(0,20);
test.y = random(0,20);
```

Remember that `test` is a structure. Here assign the values that you want to send inside the structure. In this case, we're just sending random values. In a practical application, these should be replaced with commands or sensor readings, for example.

Send the same data to multiple boards

Finally, send the message as follows:

```
esp_err_t result = esp_now_send(0, (uint8_t *) &test, sizeof(test_struct));
```

The first argument of the `esp_now_send()` function is the receiver's MAC address. If you pass `0` as an argument, it will send the same message to all registered peers. If you want to send a different message to each peer, follow the next section.

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);
```

Send different data to each board

The code to send a different message to each board is very similar to the previous one. So, we'll just take a look at the differences.

If you want to send a different message to each board, you need to create a data structure for each of your boards, for example:

```
test_struct test;
test_struct test2;
test_struct test3;
```

In this case, we're sending the same structure type (`test_struct`). You can send a different structure type as long as the receiver code is prepared to receive that type of structure.

Then, assign different values to the variables of each structure. In this example, we're just setting them to random numbers.

```
test.x = random(0,20);
test.y = random(0,20);
test2.x = random(0,20);
test2.y = random(0,20);
test3.x = random(0,20);
test3.y = random(0,20);
```

Finally, you need to call the `esp_now_send()` function for each receiver.

For example, send the `test` structure to the board whose MAC address is `broadcastAddress1`.

```
esp_err_t result1 = esp_now_send(
    broadcastAddress1,
    (uint8_t *) &test,
    sizeof(test_struct));

if (result1 == ESP_OK) {
    Serial.println("Sent with success");
}
else {
    Serial.println("Error sending the data");
}
delay(500);
```

Do the same for the other boards. For the second board send the `test2` structure:

```
esp_err_t result2 = esp_now_send(
    broadcastAddress2,
    (uint8_t *) &test2,
    sizeof(test_struct));

if (result2 == ESP_OK) {
    Serial.println("Sent with success");
}
else {
    Serial.println("Error sending the data");
```

```
}

delay(500);
```

And finally, for the third board, send the test3 structure:

```
esp_err_t result2 = esp_now_send(
    broadcastAddress2,
    (uint8_t *) &test2,
    sizeof(test_struct));

if (result2 == ESP_OK) {
    Serial.println("Sent with success");
}
else {
    Serial.println("Error sending the data");
}

delay(500);
```

Here's the complete code that sends a different message to each board.

- [Click here to download the code.](#)

ESP32 Receiver Code (ESP-NOW)

Upload the next code to the receiver boards (in our example, we used three receiver boards).

- [Click here to download the code.](#)

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

//Structure example to receive data
//Must match the sender structure
typedef struct test_struct {
    int x;
    int y;
} test_struct;

//Create a struct_message called myData
test_struct myData;

//callback function that will be executed when data is received
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
    memcpy(&myData, incomingData, sizeof(myData));
    Serial.print("Bytes received: ");
    Serial.println(len);
    Serial.print("x: ");
    Serial.println(myData.x);
    Serial.print("y: ");
    Serial.println(myData.y);
    Serial.println();
}
```

```

void setup() {
    //Initialize Serial Monitor
    Serial.begin(115200);

    //Set device as a Wi-Fi Station
    WiFi.mode(WIFI_STA);

    //Init ESP-NOW
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    // Once ESPNow is successfully Init, we will register for recv CB to
    // get recv packer info
    esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv));
}

void loop() {
}

```

How Does the Code Work?

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 test_struct {
    int x;
    int y;
} test_struct;
```

Create a `test_struct` variable called `myData`.

```
test_struct myData;
```

Create a callback function that is 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, int len) {
```

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 sender ESP32. To access variable `x`, for example, call `myData.x`.

In this example, we print the received data, but in a practical application you can print the data on an OLED display, for example.

```
Serial.print("Bytes received: ");
Serial.println(len);
Serial.print("x: ");
Serial.println(myData.x);
Serial.print("y: ");
Serial.println(myData.y);
Serial.println();
```

In the `setup()`, initialize the Serial Monitor.

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

Set the device as a Wi-Fi Station.

```
WiFi.mode(WIFI_STA);
```

Initialize ESP-NOW:

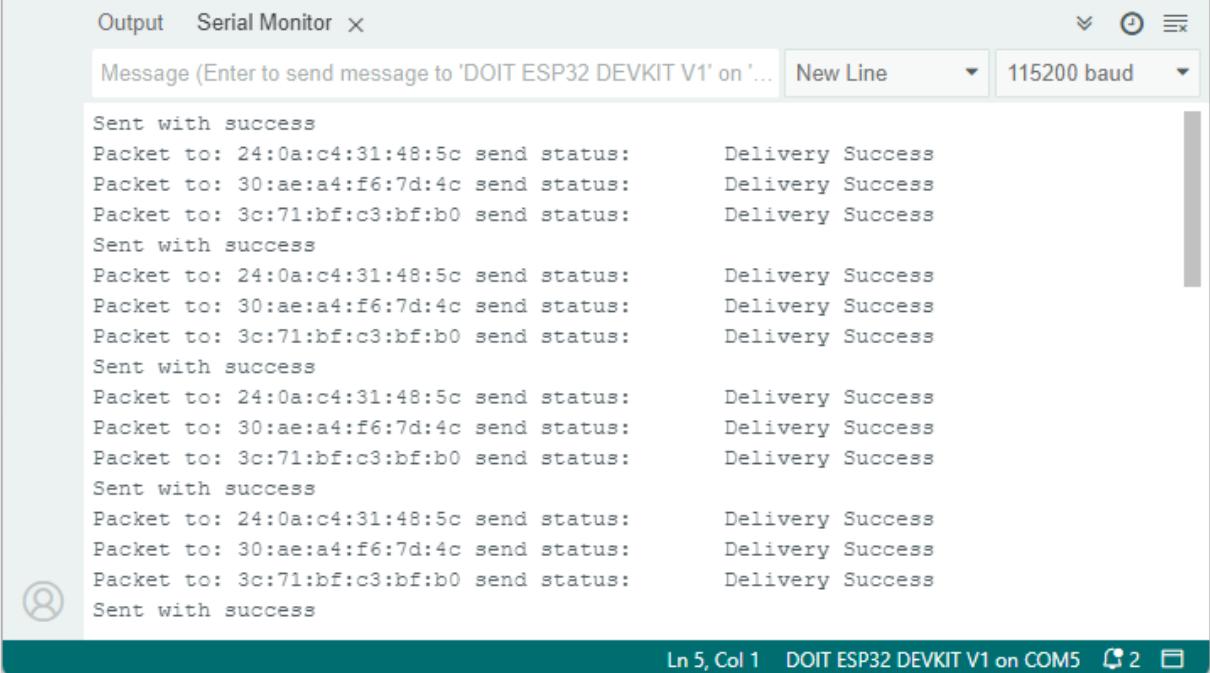
```
//Init ESP-NOW
if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
}
```

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(esp_now_recv_cb_t(OnDataRecv));
```

Demonstration

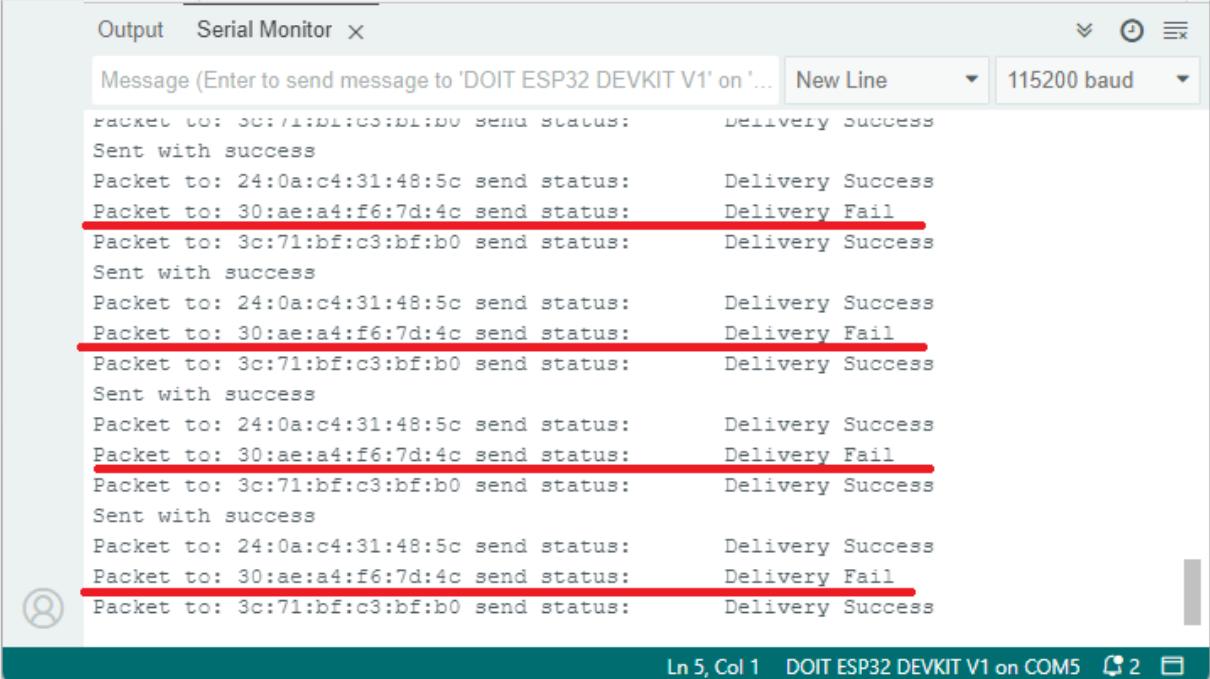
Having all your boards powered on, open the Arduino IDE Serial Monitor for the COM port the sender is connected to. You should start receiving “Delivery Success” messages with the corresponding receiver’s MAC address in the Serial Monitor.



The Serial Monitor window shows the following output:

```
Output Serial Monitor ×
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on ... New Line 115200 baud
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Success
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Success
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Success
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Success
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
```

If you remove power from one of the boards, you'll receive a "Delivery Fail" message for that specific board. So, you can identify which board didn't receive the message.

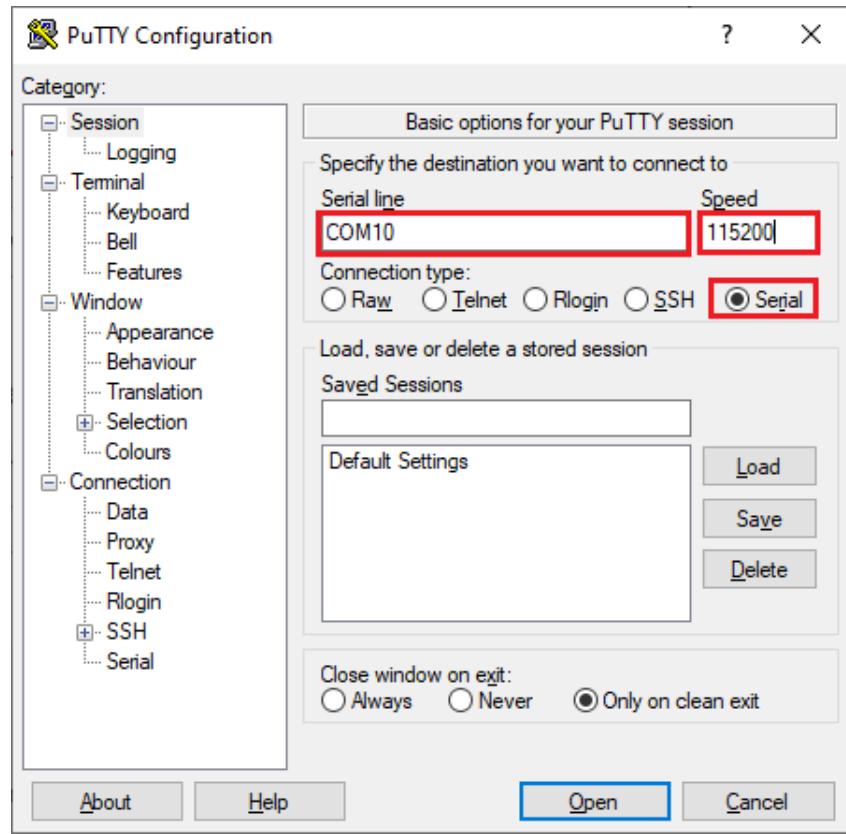


The Serial Monitor window shows the following output, with failed messages highlighted by red lines:

```
Output Serial Monitor ×
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on ... New Line 115200 baud
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Fail
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Fail
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Fail
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Fail
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Sent with success
Packet to: 24:0a:c4:31:48:5c send status: Delivery Success
Packet to: 30:ae:a4:f6:7d:4c send status: Delivery Fail
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
```

If you want to check if the boards are actually receiving the messages, you can open the Serial Monitor for the COM port they are connected to, or you can use [PuTTY](#) to establish a serial communication with your boards.

If you're using PuTTY, select Serial communication, write the COM port number and the baud rate (115200) as shown below and click Open.



Then, you should see the messages being received.

The screenshot shows the PuTTY terminal window titled 'COM10 - PuTTY'. The window displays a series of messages indicating data reception and processing. The messages are as follows:

```
x: 14  
y: 15  
  
Bytes received: 8  
x: 6  
y: 6  
  
Bytes received: 8  
x: 5  
y: 3  
  
Bytes received: 8  
x: 0  
y: 13  
  
Bytes received: 8  
x: 16  
y: 12  
  
Bytes received: 8  
x: 19  
y: 1
```

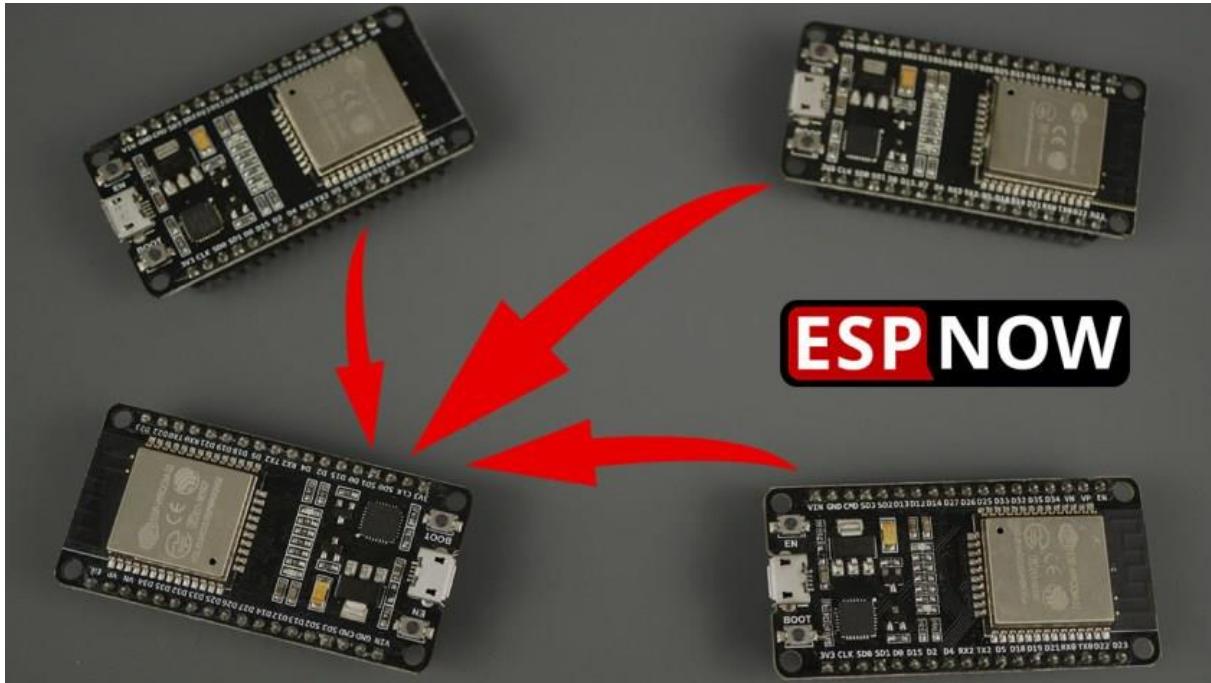
Open a serial communication for each of your boards and check that they are receiving the messages.

Wrapping Up

In this tutorial, you've learned how to send data to multiple ESP32 boards from a single ESP32 using ESP-NOW (one-to-many communication). You can send the same data to all boards or send specific data to each board.

In the next Unit, you'll learn how to receive data from multiple boards.

10.4 - ESP-NOW Receive Data from Multiple Boards (many-to-one)



This Unit shows how to set up an ESP32 board to receive data from multiple ESP32 boards via ESP-NOW communication protocol (many-to-one configuration). This configuration is ideal if you want to collect data from several sensors' nodes into one ESP32 board.

Project Overview

Here's a quick overview of the project we'll build:

- One ESP32 board acts as a receiver/slave;
- Multiple ESP32 boards act as senders/masters. We tested this example with 5 ESP32 sender boards and it worked fine. You should be able to add more boards to your setup;
- The sender board receives an acknowledge message indicating if the message was successfully delivered or not;
- The ESP32 receiver board receives the messages from all senders and identifies which board sent the message;

- As an example, we'll exchange random values between the boards. You should modify this example to send commands or sensor readings.



Getting the Receiver Board's MAC Address

If you followed previous units, you know that to send messages via ESP-NOW, you need to know the receiver board's MAC address.

Upload the following code to get its MAC address.

- [Click here to download the code.](#)

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

void readMacAddress(){
    uint8_t baseMac[6];
    esp_err_t ret = esp_wifi_get_mac(WIFI_IF_STA, baseMac);
    if (ret == ESP_OK) {
        Serial.printf("%02x:%02x:%02x:%02x:%02x:%02x\n",
                     baseMac[0], baseMac[1], baseMac[2],
                     baseMac[3], baseMac[4], baseMac[5]);
    } else {
        Serial.println("Failed to read MAC address");
    }
}

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

    WiFi.mode(WIFI_STA);
    WiFi.STA.begin();
```

```
Serial.print("[DEFAULT] ESP32 Board MAC Address: ");
readMacAddress();
}

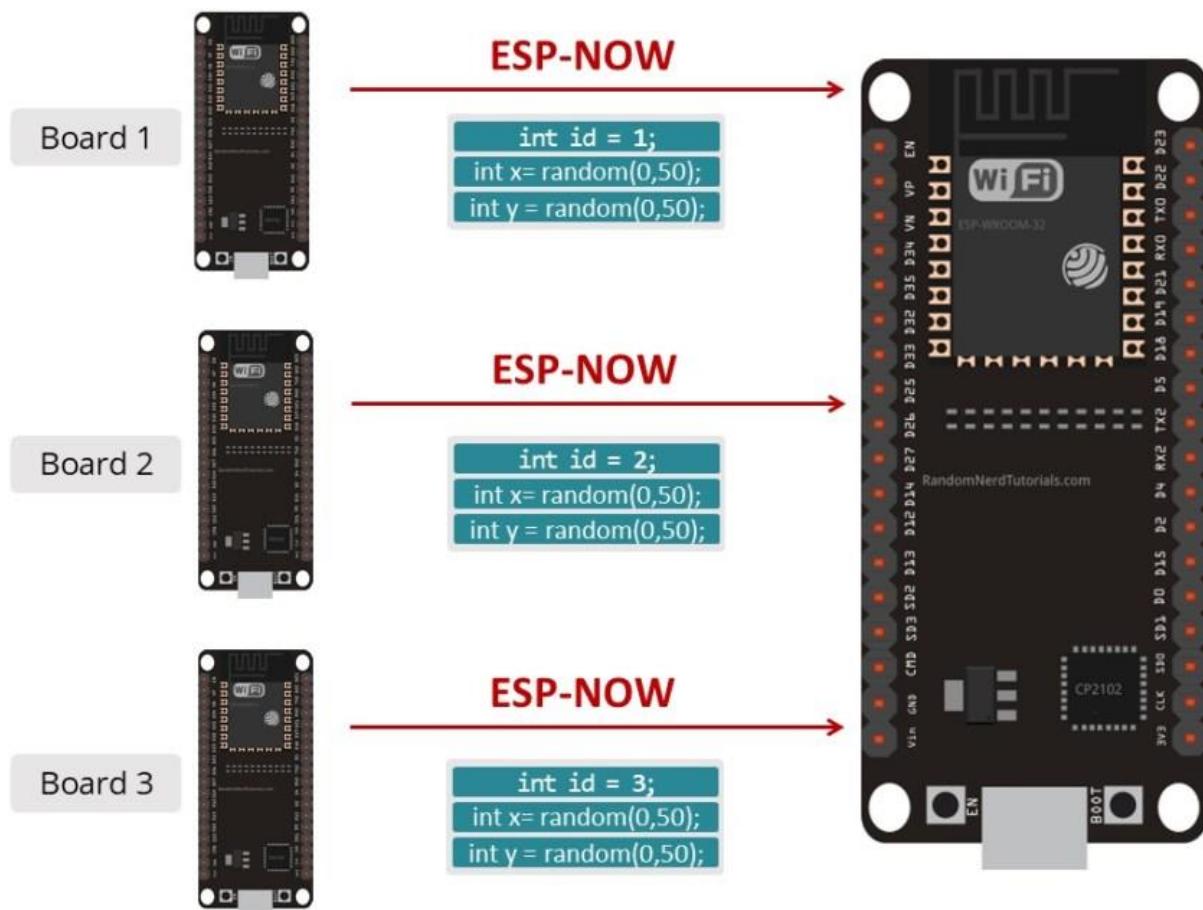
void loop(){
}
```

ESP32 Sender Code (ESP-NOW)

The receiver can identify each sender by its unique MAC address. However, dealing with different MAC addresses on the Receiver side to identify which board sent which message can be tricky.

So, to make things easier, we'll identify each board with a unique number (`id`) that starts at 1. If you have three boards, one will have ID number 1, the other number 2, and finally number 3. The ID will be sent to the receiver alongside the other variables.

As an example, we'll exchange a structure that contains the board `id` number and two random numbers `x` and `y` as shown in the figure below.



Upload the following code to each of your sender boards. Don't forget to increment the `id` number for each sender board.

- [Click here to download the code.](#)

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

// REPLACE WITH THE RECEIVER'S MAC Address
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

// Create a peer interface
esp_now_peer_info_t peerInfo;

// Structure example to send data
// Must match the receiver structure
typedef struct struct_message {
    int id; // must be unique for each sender board
    int x;
    int y;
} struct_message;

//Create a struct_message called myData
struct_message myData;

// callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
    Serial.print("\r\nLast Packet Send Status:\t");
    Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" :
                  "Delivery Fail");
}

void setup() {
    // Init Serial Monitor
    Serial.begin(115200);

    // Set device as a Wi-Fi Station
    WiFi.mode(WIFI_STA);

    // Init ESP-NOW
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    // Once ESPNow is successfully Init, we will register for Send CB to
    // get the status of Trasnmitted packet
    esp_now_register_send_cb(OnDataSent);

    // 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;
    }
}
```

```

}

void loop() {
    // Set values to send
    myData.id = 1;
    myData.x = random(0,50);
    myData.y = random(0,50);

    // Send message via ESP-NOW
    esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &myData,
                                    sizeof(myData));

    if (result == ESP_OK) {
        Serial.println("Sent with success");
    }
    else {
        Serial.println("Error sending the data");
    }
    delay(10000);
}

```

How the Code Works

Include the WiFi and esp_now libraries.

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

Insert the receiver's MAC address on the following line.

```
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
```

Create a variable of type esp_now_peer_info_t called peerInfo that will then hold all the information about the peer (the other device we want to connect to).

```
// Create a peer interface
esp_now_peer_info_t peerInfo;
```

Then, create a structure that contains the data we want to send. We called this structure struct_message and it contains three integer variables: the board id, x and y. You can change this to send whatever variable types you want (but don't forget to change that on the receiver side too).

```
typedef struct struct_message {
    int id; // must be unique for each sender board
    int x;
    int y;
} struct_message;
```

Create a new variable of type struct_message that is called myData that will store the variables' values.

```
struct_message myData;
```

OnDataSent() callback function

Next, define the `OnDataSent()` function. This is a callback function that will be executed when a message is sent. In this case, this function prints if the message was successfully delivered or not.

```
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
    Serial.print("\r\nLast Packet Send Status:\t");
    Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" :
                  "Delivery Fail");
}
```

setup()

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, register for the `OnDataSent()` function created previously.

```
esp_now_register_send_cb(OnDataSent);
```

Add peer device

To send data to another board (the receiver), you need to pair it as a peer. The following lines register and add a new peer.

```
// 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;  
}
```

loop()

In the `loop()`, we'll send a message via ESP-NOW every 10 seconds (you can change this delay time).

Assign a value to each variable.

```
// Set values to send  
myData.id = 1;  
myData.x = random(0,50);  
myData.y = random(0,50);
```

Don't forget to change the `id` variable for each sender board.

Remember that `myData` is a structure. Here assign the values that you want to send inside the structure. In this case, we're just sending the id and random values `x` and `y`. In a practical application, these should be replaced with commands or sensor readings, for example.

Send ESP-NOW message

Finally, send the message via ESP-NOW.

```
// Send message via ESP-NOW  
esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &myData,  
                                sizeof(myData));  
  
if (result == ESP_OK) {  
    Serial.println("Sent with success");  
}  
else {  
    Serial.println("Error sending the data");  
}
```

ESP32 Receiver Code (ESP-NOW)

Upload the following code to your ESP32 receiver board. The code is prepared to receive data from three different boards. You can easily modify the code to receive data from a different number of boards.

- [Click here to download the code.](#)

```

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

// Structure example to receive data
// Must match the sender structure
typedef struct struct_message {
    int id;
    int x;
    int y;
}struct_message;

// Create a struct_message called myData
struct_message myData;

// Create a structure to hold the readings from each board
struct_message board1;
struct_message board2;
struct_message board3;

// Create an array with all the structures
struct_message boardsStruct[3] = {board1, board2, board3};

// callback function that will be executed when data is received
void OnDataRecv(const uint8_t * mac_addr, const uint8_t *incomingData, int len) {
    memcpy(&myData, incomingData, sizeof(myData));
    Serial.printf("Board ID %u: %u bytes\n", myData.id, len);
    // Update the structures with the new incoming data
    boardsStruct[myData.id-1].x = myData.x;
    boardsStruct[myData.id-1].y = myData.y;
    Serial.printf("x value: %d \n", boardsStruct[myData.id-1].x);
    Serial.printf("y value: %d \n", boardsStruct[myData.id-1].y);
    Serial.println();
}

void setup() {
    //Initialize Serial Monitor
    Serial.begin(115200);

    //Set device as a Wi-Fi Station
    WiFi.mode(WIFI_STA);

    //Init ESP-NOW
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    // Once ESPNow is successfully Init, we will register for recv CB to
    // get recv packer info
    esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv));
}

void loop() {
    // Acess the variables for each board
    /*int board1X = boardsStruct[0].x;
    int board1Y = boardsStruct[0].y;
    int board2X = boardsStruct[1].x;
    int board2Y = boardsStruct[1].y;
    int board3X = boardsStruct[2].x;
    int board3Y = boardsStruct[2].y;*/
```

```
    delay(10000);
}
```

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 {
    int id;
    int x;
    int y;
}struct_message;
```

Create a `struct_message` variable called `myData` that will hold the data received.

```
struct_message myData;
```

Then, create a `struct_message` variable for each board, so that we can assign the received data to the corresponding board. Here we're creating structures for three sender boards. If you have more sender boards, you need to create more structures.

```
struct_message board1;
struct_message board2;
struct_message board3;
```

Create an array that contains all the board structures. If you're using a different number of boards you need to change that.

```
struct_message boardsStruct[3] = {board1, board2, board3};
```

onDataRecv()

Create a callback function that is 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_addr, const uint8_t *incomingData, int len) {
    memcpy(&myData, incomingData, sizeof(myData));
    Serial.printf("Board ID %u: %u bytes\n", myData.id, len);
```

```

// Update the structures with the new incoming data
boardsStruct[myData.id-1].x = myData.x;
boardsStruct[myData.id-1].y = myData.y;
Serial.printf("x value: %d \n", boardsStruct[myData.id-1].x);
Serial.printf("y value: %d \n", boardsStruct[myData.id-1].y);
Serial.println();
}

```

Copy the content of the `incomingData` data variable into the `myData` variable.

```
memcpy(&myData, incomingData, sizeof(myData));
```

Now, the `myData` structure contains several variables with the values sent by one of the ESP32 senders. We can identify which board send the packet by its ID: `myData.id`.

This way, we can assign the values received to the corresponding boards on the `boardsStruct` array:

```

boardsStruct[myData.id-1].x = myData.x;
boardsStruct[myData.id-1].y = myData.y;

```

For example, imagine you receive a packet from board with `id` 2. The value of `myData.id`, is 2.

So, you want to update the values of the `board2` structure. The `board2` structure is the element with index 1 on the `boardsStruct` array. That's why we subtract 1, because arrays in C have 0 indexing. It may help if you take a look at the following image.

boardsStruct Array		
boardsStruct[0]	boardsStruct[1]	boardsStruct[2]
structure board1 <pre> int id = 1; int x= random(0,50); int y = random(0,50); </pre>	structure board2 <pre> int id = 2; int x= random(0,50); int y = random(0,50); </pre>	structure board3 <pre> int id = 3; int x= random(0,50); int y = random(0,50); </pre>

setup()

In the `setup()`, initialize 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(esp_now_recv_cb_t(OnDataRecv));
```

The following lines commented on the `loop()` exemplify what you need to do if you want to access the variables of each board structure. For example, to access the `x` value of board1:

```
int board1X = boardsStruct[0].x;
```

Demonstration

Upload the sender code to each of your sender boards. Don't forget to give a different ID to each board.

Upload the receiver code to the ESP32 receiver board. Don't forget to modify the structure to match the number of sender boards.

On the senders' Serial Monitor, you should get a "Delivery Success" message if the messages are delivered correctly.

```
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' New Line ▾ 115200 baud ▾  
Sent with success  
  
Last Packet Send Status: Delivery Success  
Sent with success  
  
Last Packet Send Status: Delivery Success  
Sent with success  
  
Last Packet Send Status: Delivery Success  
Sent with success  
  
Last Packet Send Status: Delivery Success  
Sent with success  
  
Last Packet Send Status: Delivery Success  
Sent with success
```



On the receiver board, you should be receiving the packets from all the other boards.

```
Board ID 2: 12 bytes  
x value: 7  
y value: 1  
  
Board ID 3: 12 bytes  
x value: 43  
y value: 14  
  
Board ID 1: 12 bytes  
x value: 21  
y value: 19  
  
Board ID 2: 12 bytes  
x value: 21  
y value: 27
```



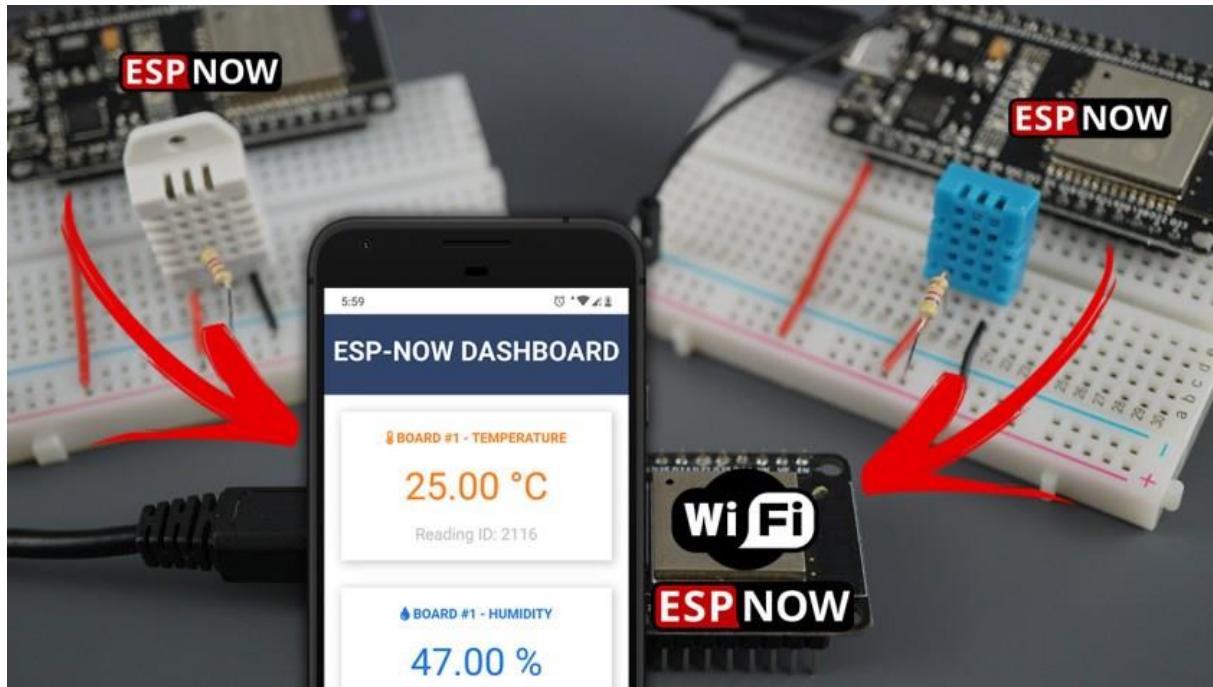
Wrapping Up

In this tutorial, you've learned how to set up an ESP32 to receive data from multiple ESP32 boards using ESP-NOW (many-to-one configuration).

As an example, we've exchanged random numbers. In a real application, those can be replaced with actual sensor readings or commands. This is ideal if you want to collect data from several sensor nodes.

You can take this project further and create a web server on the receiver board to display the received messages – that's what we're going to do in the next Unit.

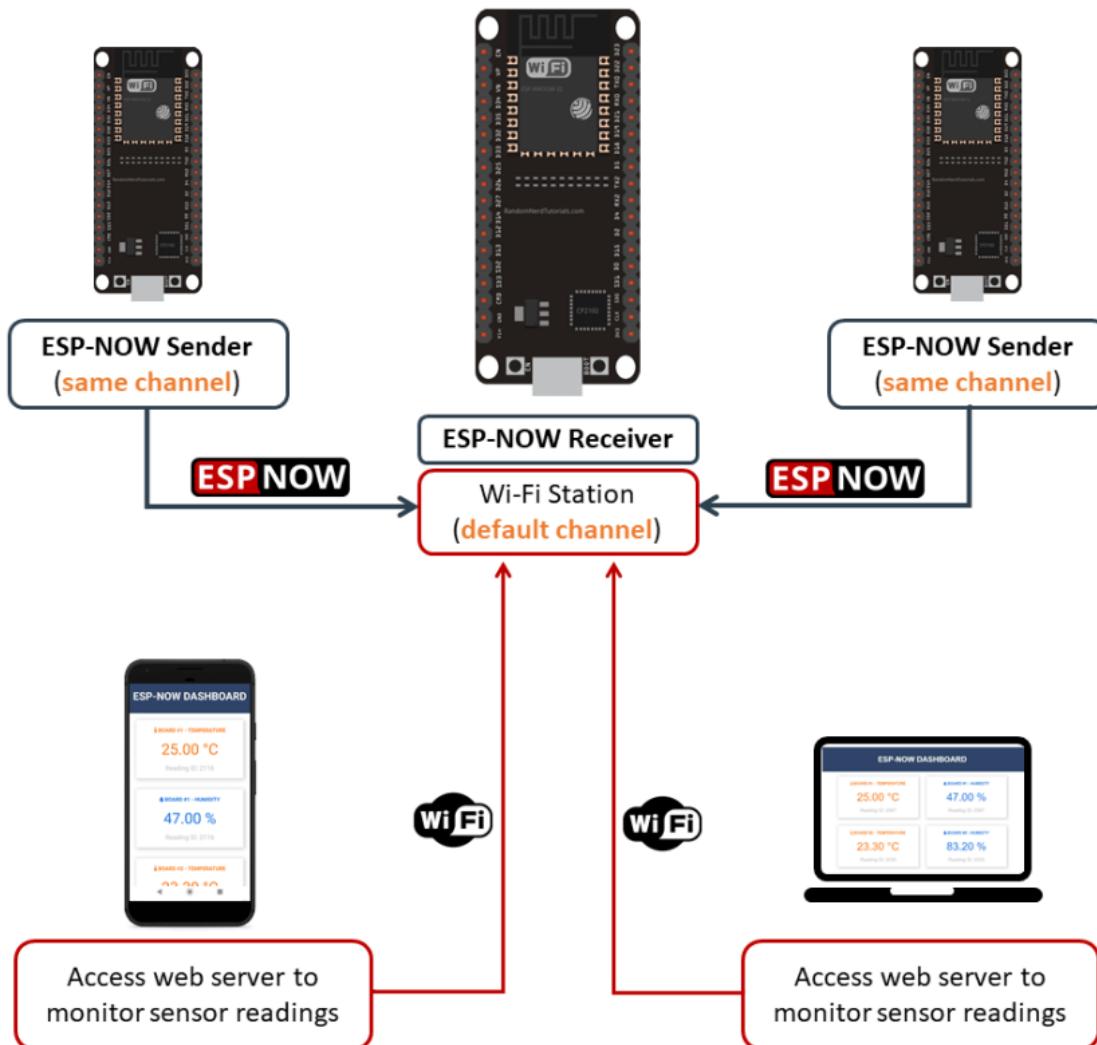
10.5 - ESP-NOW Web Server Sensor Dashboard (ESP-NOW + Wi-Fi)



In this Unit, you'll learn how to host an ESP32 web server and use ESP-NOW communication protocol at the same time. You can have several ESP32 boards sending sensor readings via ESP-NOW to one ESP32 receiver that displays all readings on a web server.

(continues on next page ...)

Using ESP-NOW and Wi-Fi Simultaneously

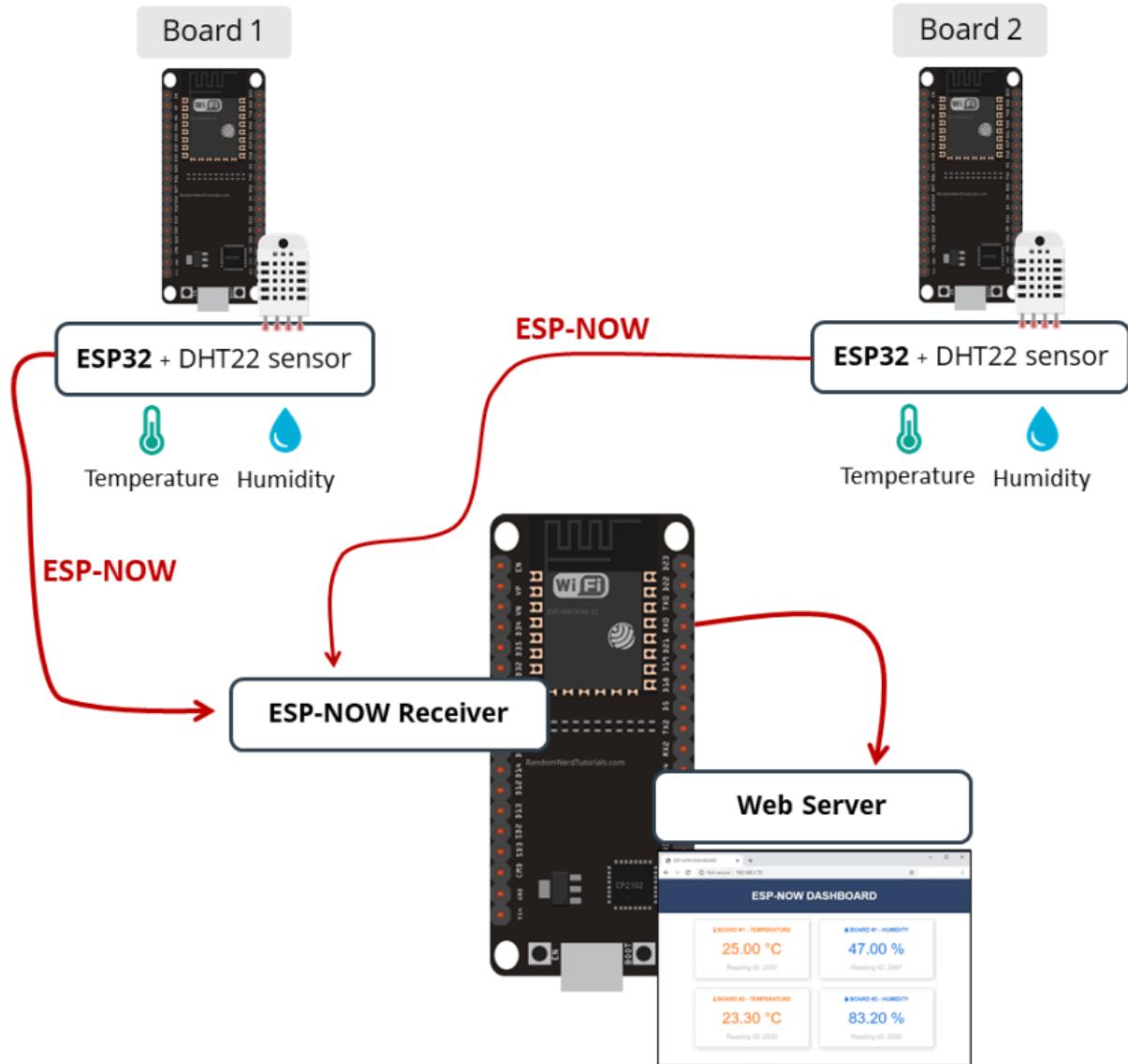


There are a few things you need to take into account if you want to use Wi-Fi to host a web server and use ESP-NOW simultaneously to receive sensor readings from other boards:

- The ESP32 sender boards must use the same Wi-Fi channel of the receiver board.
- The Wi-Fi channel of the receiver board is automatically assigned by your Wi-Fi router.
- The Wi-Fi mode of the receiver board must be an access point and a station (`WIFI_AP_STA`).
- You can set up the same Wi-Fi channel manually or add a simple spinet of code on the sender to set its Wi-Fi channel to the same of the receiver board.

Project Overview

The following diagram shows a high-level overview of the project we'll build.



- There are two ESP32 sender boards that send DHT22 temperature and humidity readings via ESP-NOW to one ESP32 receiver board (ESP-NOW many-to-one configuration);
- The ESP32 receiver board receives the packets and displays the readings on a web server;
- The web server is updated automatically every time it receives a new reading using Server-Sent Events (SSE).

DHT Library

The ESP32 sender board will send temperature and humidity readings from a DHT22 sensor. To read from the DHT sensor, we'll use the [DHT library from Adafruit](#). Follow the next instructions to install the library.

1. Open your Arduino IDE and go to **Sketch > Include Library > Manage Libraries**. The Library Manager should open.
2. Search for “**DHT**” on the Search box and install the DHT library from Adafruit. Install any other required dependencies.

Async Web Server Libraries

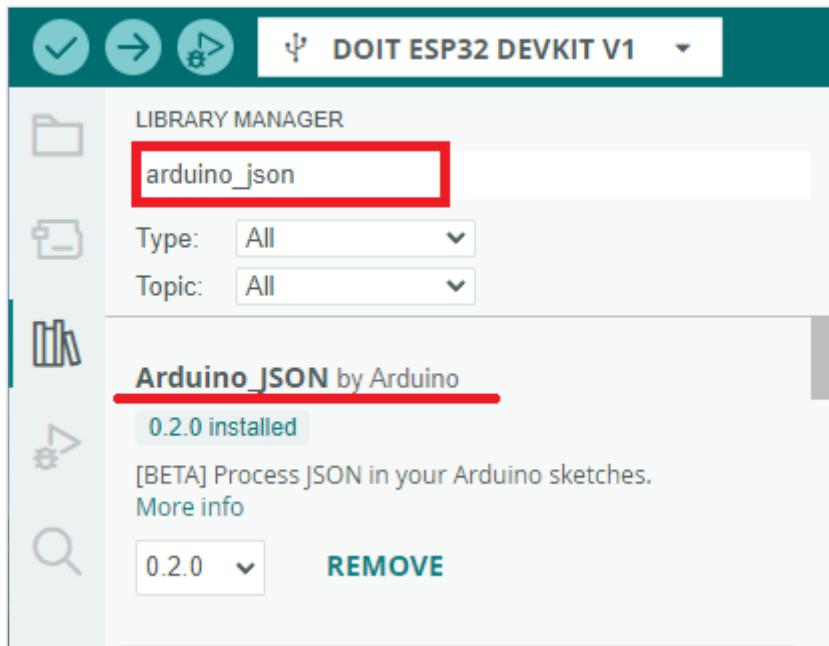
To build the web server you need to install the following libraries:

- ESPAsyncWebServer – [click here to download the .ZIP folder](#)
- AsyncTCP – [click here to download the .ZIP folder](#)

These libraries aren’t available to install through the Arduino Library Manager, so in your Arduino IDE, go to **Sketch > Include Library > Add .zip Library** and select the libraries you’ve just downloaded.

Arduino_JSON Library

You need to install the `Arduino_JSON` library. You can install this library in the Arduino IDE Library Manager. Just go to **Sketch > Include Library > Manage Libraries** and search for the library name as follows:



We're using the **Arduino_JSON library by Arduino**.

Parts Required

To follow this tutorial, you need multiple ESP32 boards. Here's the complete list of parts:

- 3x [ESP32](#) (read [Best ESP32 development boards](#))
- 2x [DHT22 temperature and humidity sensor](#) – [DHT guide for ESP32](#)
- 2x [4.7k Ohm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)

Getting the Receiver Board MAC Address

If you followed previous units, you know that to send messages via ESP-NOW, you need to get the receiver board's MAC address. Upload the following code to get its MAC address.

- [Click here to download the code.](#)

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

void readMacAddress(){
    uint8_t baseMac[6];
    esp_err_t ret = esp_wifi_get_mac(WIFI_IF_STA, baseMac);
```

```

if (ret == ESP_OK) {
    Serial.printf("%02x:%02x:%02x:%02x:%02x:%02x\n",
                  baseMac[0], baseMac[1], baseMac[2],
                  baseMac[3], baseMac[4], baseMac[5]);
} else {
    Serial.println("Failed to read MAC address");
}
}

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

    WiFi.mode(WIFI_STA);
    WiFi.STA.begin();

    Serial.print("[DEFAULT] ESP32 Board MAC Address: ");
    readMacAddress();
}

void loop(){
}

```

After uploading the code, press the RST/EN button, and the MAC address should be displayed on the Serial Monitor.

ESP32 Receiver (ESP-NOW + Web Server)

The ESP32 receiver board receives the packets from the sender boards and hosts a web server to display the latest received readings. Upload the following code to your receiver board – the code is prepared to receive readings from two different boards.

- [Click here to download the code.](#)

```

#include <esp_now.h>
#include <WiFi.h>
#include "ESPAsyncWebServer.h"
#include <Arduino_JSON.h>

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

// Structure example to receive data
// Must match the sender structure
typedef struct struct_message {
    int id;
    float temp;
    float hum;
    unsigned int readingId;
} struct_message;

```

```

struct_message incomingReadings;

JSONVar board;

AsyncWebServer server(80);
AsyncEventSource events("/events");

// callback function that will be executed when data is received
void OnDataRecv(const uint8_t * mac_addr, const uint8_t *incomingData, int len) {
    // Copies the sender mac address to a string
    char macStr[18];
    Serial.print("Packet received from: ");
    snprintf(macStr, sizeof(macStr), "%02x:%02x:%02x:%02x:%02x:%02x",
              mac_addr[0], mac_addr[1], mac_addr[2],
              mac_addr[3], mac_addr[4], mac_addr[5]);
    Serial.println(macStr);
    memcpy(&incomingReadings, incomingData, sizeof(incomingReadings));

    board["id"] = incomingReadings.id;
    board["temperature"] = incomingReadings.temp;
    board["humidity"] = incomingReadings.hum;
    board["readingId"] = String(incomingReadings.readingId);
    String jsonString = JSON.stringify(board);
    events.send(jsonString.c_str(), "new_readings", millis());

    Serial.printf("Board ID %u: %u bytes\n", incomingReadings.id, len);
    Serial.printf("t value: %4.2f \n", incomingReadings.temp);
    Serial.printf("h value: %4.2f \n", incomingReadings.hum);
    Serial.printf("readingID value: %d \n", incomingReadings.readingId);
    Serial.println();
}

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE HTML><html>
<head>
    <title>ESP-NOW DASHBOARD</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css"
integrity="sha384-fnmOCqbTlWljl8LyTjo7mOUSTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr"
crossorigin="anonymous">
    <link rel="icon" href="data:, ">
    <style>
        html {font-family: Arial; display: inline-block; text-align: center;}
        p { font-size: 1.2rem; }
        body { margin: 0; }
        .topnav { overflow: hidden; background-color: #2f4468; color: white; font-size: 1.7rem; }
        .content { padding: 20px; }
            .card { background-color: white; box-shadow: 2px 2px 12px 1px
rgba(140,140,140,.5); }
        .cards { max-width: 700px; margin: 0 auto; display: grid; grid-gap: 2rem;
grid-template-columns: repeat(auto-fit, minmax(300px, 1fr)); }
        .reading { font-size: 2.8rem; }
        .packet { color: #bebebe; }
        .card.temperature { color: #fd7e14; }
        .card.humidity { color: #1b78e2; }
    </style>
</head>
<body>

```

```

<div class="topnav">
    <h3>ESP-NOW DASHBOARD</h3>
</div>
<div class="content">
    <div class="cards">
        <div class="card temperature">
            <h4><i class="fas fa-thermometer-half"></i> BOARD #1 - TEMPERATURE</h4><p><span class="reading"><span id="t1"></span>&deg;C</span></p><p class="packet">Reading ID: <span id="rt1"></span></p>
        </div>
        <div class="card humidity">
            <h4><i class="fas fa-tint"></i> BOARD #1 - HUMIDITY</h4><p><span class="reading"><span id="h1"></span>&percnt;</span></p><p class="packet">Reading ID: <span id="rh1"></span></p>
        </div>
        <div class="card temperature">
            <h4><i class="fas fa-thermometer-half"></i> BOARD #2 - TEMPERATURE</h4><p><span class="reading"><span id="t2"></span>&deg;C</span></p><p class="packet">Reading ID: <span id="rt2"></span></p>
        </div>
        <div class="card humidity">
            <h4><i class="fas fa-tint"></i> BOARD #2 - HUMIDITY</h4><p><span class="reading"><span id="h2"></span>&percnt;</span></p><p class="packet">Reading ID: <span id="rh2"></span></p>
        </div>
    </div>
</div>
<script>
if (!window.EventSource) {
    var source = new EventSource('/events');

    source.addEventListener('open', function(e) {
        console.log("Events Connected");
    }, false);
    source.addEventListener('error', function(e) {
        if (e.target.readyState != EventSource.OPEN) {
            console.log("Events Disconnected");
        }
    }, false);

    source.addEventListener('message', function(e) {
        console.log("message", e.data);
    }, false);

    source.addEventListener('new_readings', function(e) {
        console.log("new_readings", e.data);
        var obj = JSON.parse(e.data);
        document.getElementById("t"+obj.id).innerHTML = obj.temperature.toFixed(2);
        document.getElementById("h"+obj.id).innerHTML = obj.humidity.toFixed(2);
        document.getElementById("rt"+obj.id).innerHTML = obj.readingId;
        document.getElementById("rh"+obj.id).innerHTML = obj.readingId;
    }, false);
}
</script>
</body>
</html>)rawliteral";

void setup() {
    // Initialize Serial Monitor
    Serial.begin(115200);

```

```

// Set the device as a Station and Soft Access Point simultaneously
WiFi.mode(WIFI_AP_STA);

// Set device as a Wi-Fi Station
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Setting as a Wi-Fi Station..");
}
Serial.print("Station IP Address: ");
Serial.println(WiFi.localIP());
Serial.print("Wi-Fi Channel: ");
Serial.println(WiFi.channel());

// Init ESP-NOW
if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
}

// Once ESPNow is successfully Init, we will register for recv CB to
// get recv packer info
esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv));

server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
    request->send_P(200, "text/html", index_html);
});

events.onConnect([](AsyncEventSourceClient *client){
    if(client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n",
                      client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);
server.begin();
}

void loop() {
    static unsigned long lastEventTime = millis();
    static const unsigned long EVENT_INTERVAL_MS = 5000;
    if ((millis() - lastEventTime) > EVENT_INTERVAL_MS) {
        events.send("ping",NULL,millis());
        lastEventTime = millis();
    }
}

```

How Does the Code Work?

First, include the necessary libraries.

```

#include <esp_now.h>
#include <WiFi.h>
#include "ESPAsyncWebServer.h"
#include <Arduino_JSON.h>

```

The [Arduino JSON library](#) is needed because we'll create a JSON variable with the data received from each board. This JSON variable will be used to send all the required information to the web page, as you'll see later in this project.

Insert your network credentials on the following lines so that the ESP32 can connect to your local network.

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

Data Structure

Then, create a structure that contains the data we'll receive. We called this structure `struct_message`, and it contains the board ID, temperature and humidity readings, and the reading ID.

```
typedef struct struct_message {
    int id;
    float temp;
    float hum;
    unsigned int readingId;
} struct_message;
```

Create a new variable of type `struct_message` that is called `incomingReadings` that will store the variables values.

```
struct_message incomingReadings;
```

Create a JSON variable called `board`.

```
JSONVar board;
```

Create an Async Web Server on port 80.

```
AsyncWebServer server(80);
```

Create Event Source

To automatically display the information on the web server when a new reading arrives, we'll use Server-Sent Events (SSE). The following line creates a new event source on `/events`.

```
AsyncEventSource events("/events");
```

Server-Sent Events allow a web page (client) to get updates from a server. We'll use this to automatically display new readings on the web server page when a new ESP-NOW packet arrives.

OnDataRecv() function

The OnDataRecv() function will be executed when you receive a new ESP-NOW packet.

```
void OnDataRecv(const uint8_t * mac_addr, const uint8_t *incomingData, int len)
```

Copy the information in the incomingData variable into the incomingReadings structure variable.

```
memcpy(&incomingReadings, incomingData, sizeof(incomingReadings));
```

Then, create a JSON String variable with the received information (jsonString variable):

```
board["id"] = incomingReadings.id;
board["temperature"] = incomingReadings.temp;
board["humidity"] = incomingReadings.hum;
board["readingId"] = String(incomingReadings.readingId);
String jsonString = JSON.stringify(board);
```

Here's an example on how the jsonString variable may look like after receiving the readings:

```
board = {
  "id": "1",
  "temperature": "24.32",
  "humidity": "65.85",
  "readingId": "2"
}
```

After gathering all the received data on the jsonString variable, send that information to the browser as an event ("new_readings").

```
events.send(jsonString.c_str(), "new_readings", millis());
```

Later, we'll see how to handle these events on the client side.

Finally, print the received information on the Arduino IDE Serial Monitor for debugging purposes:

```
Serial.printf("Board ID %u: %u bytes\n", incomingReadings.id, len);
Serial.printf("t value: %4.2f \n", incomingReadings.temp);
Serial.printf("h value: %4.2f \n", incomingReadings.hum);
Serial.printf("readingID value: %d \n", incomingReadings.readingId);
Serial.println();
```

Building the Web Page

The `index_html` variable contains all the HTML, CSS and JavaScript to build the web page. We won't go into details on how the HTML and CSS works. We'll just take a look at how to handle the events sent by the server.

Handle Events

Create a new `EventSource` object and specify the URL of the page sending the updates. In our case, it's `/events`.

```
if (!!window.EventSource) {
  var source = new EventSource('/events');
```

Once you've instantiated an event source, you can start listening for messages from the server with `addEventListener()`.

These are the default event listeners, as shown here in the [AsyncWebServer documentation](#).

```
source.addEventListener('open', function(e) {
  console.log("Events Connected");
}, false);
source.addEventListener('error', function(e) {
  if (e.target.readyState != EventSource.OPEN) {
    console.log("Events Disconnected");
  }
}, false);

source.addEventListener('message', function(e) {
  console.log("message", e.data);
}, false);
```

Then, add the event listener for "new_readings".

```
source.addEventListener('new_readings', function(e) {
```

When the ESP32 receives a new packet, it sends a JSON string with the readings as an event ("new_readings") to the client. The following lines handle what happens when the browser receives that event.

```
console.log("new_readings", e.data);
var obj = JSON.parse(e.data);
document.getElementById("t"+obj.id).innerHTML = obj.temperature.toFixed(2);
document.getElementById("h"+obj.id).innerHTML = obj.humidity.toFixed(2);
document.getElementById("rt"+obj.id).innerHTML = obj.readingId;
document.getElementById("rh"+obj.id).innerHTML = obj.readingId;
```

Print the new readings on the browser console, and put the received data into the elements with the corresponding id on the web page.

setup()

In the `setup()`, set the ESP32 receiver as an access point and Wi-Fi station:

```
WiFi.mode(WIFI_AP_STA);
```

The following lines connect the ESP32 to your local network and print the IP address and the Wi-Fi channel:

```
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Setting as a Wi-Fi Station..");
}
Serial.print("Station IP Address: ");
Serial.println(WiFi.localIP());
Serial.print("Wi-Fi Channel: ");
Serial.println(WiFi.channel());
```

Initialize ESP-NOW.

```
// Init ESP-NOW
if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
}
```

Register for the `OnDataRecv()` callback function, so that it is executed when a new ESP-NOW packet arrives.

```
esp_now_register_recv_cb(esp_now_recv_cb_t(OnDataRecv));
```

Handle Requests

When you access the ESP32 IP address on the root / URL, send the text that is stored on the `index_html` variable to build the web page.

```
server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
    request->send_P(200, "text/html", index_html);
```

```
});
```

Server Event Source

Set up the event source on the server.

```
events.onConnect([](AsyncEventSourceClient *client){  
    if(client->lastId()){  
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n",  
client->lastId());  
    }  
    // send event with message "hello!", id current millis  
    // and set reconnect delay to 1 second  
    client->send("hello!", NULL, millis(), 10000);  
});  
server.addHandler(&events);
```

Finally, start the server.

```
server.begin();
```

loop()

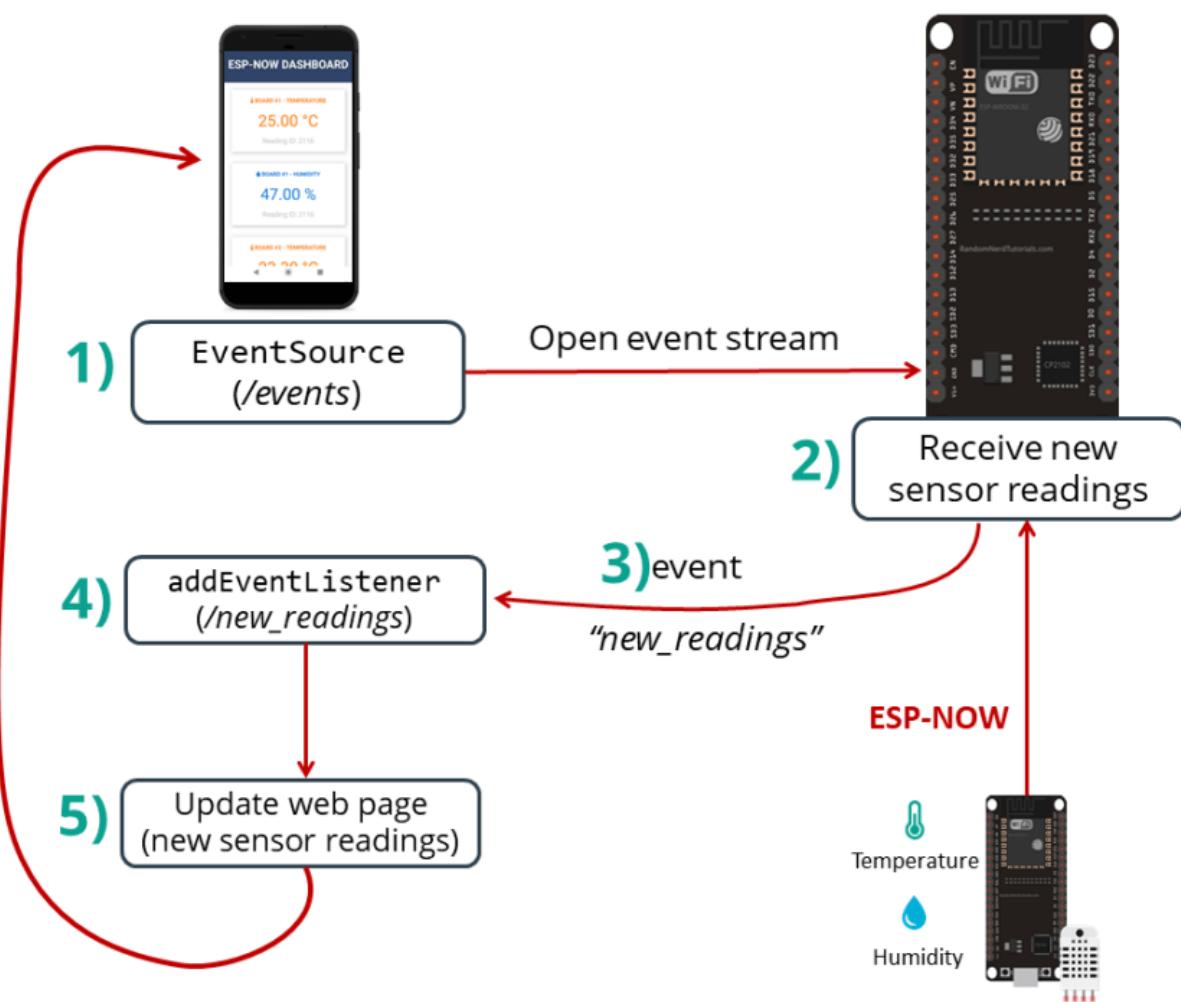
In the `loop()`, send a ping every 5 seconds. This is used to check on the client side, if the server is still running.

```
static unsigned long lastEventTime = millis();  
static const unsigned long EVENT_INTERVAL_MS = 5000;  
if ((millis() - lastEventTime) > EVENT_INTERVAL_MS) {  
    events.send("ping",NULL,millis());  
    lastEventTime = millis();  
}
```

The following diagram summarizes how the Server-Sent Events work on this project.

Client (browser)

Server (ESP32)



Demonstration

After uploading the code to the receiver board, press the on-board EN/RST button.

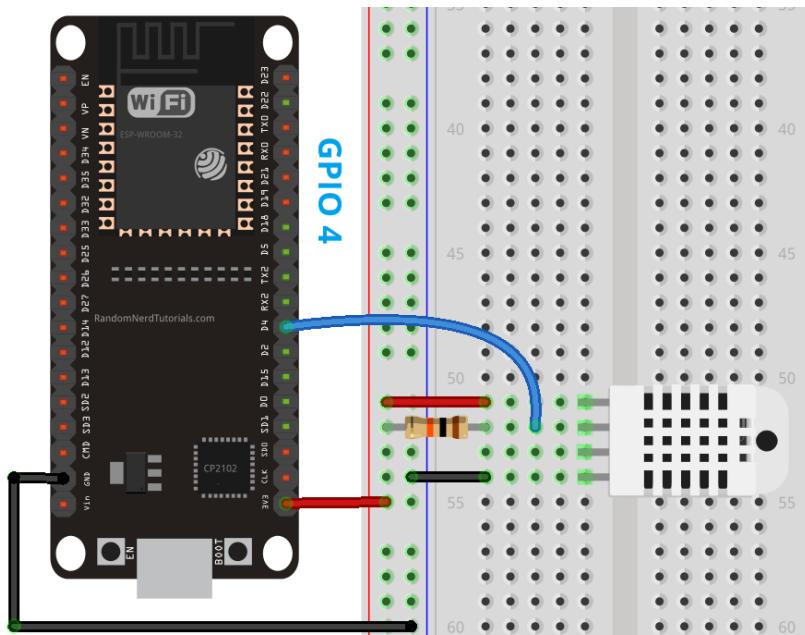
The ESP32 IP address should be printed on the Serial Monitor as well as the Wi-Fi channel.

```
rst:0x1 (POWERON_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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Setting as a Wi-Fi Station..
Setting as a Wi-Fi Station..
Setting as a Wi-Fi Station..
Station IP Address: 192.168.1.75
Wi-Fi Channel: 12
```

Ln 13, Col 35 DOIT ESP32 DEVKIT V1 on COM3

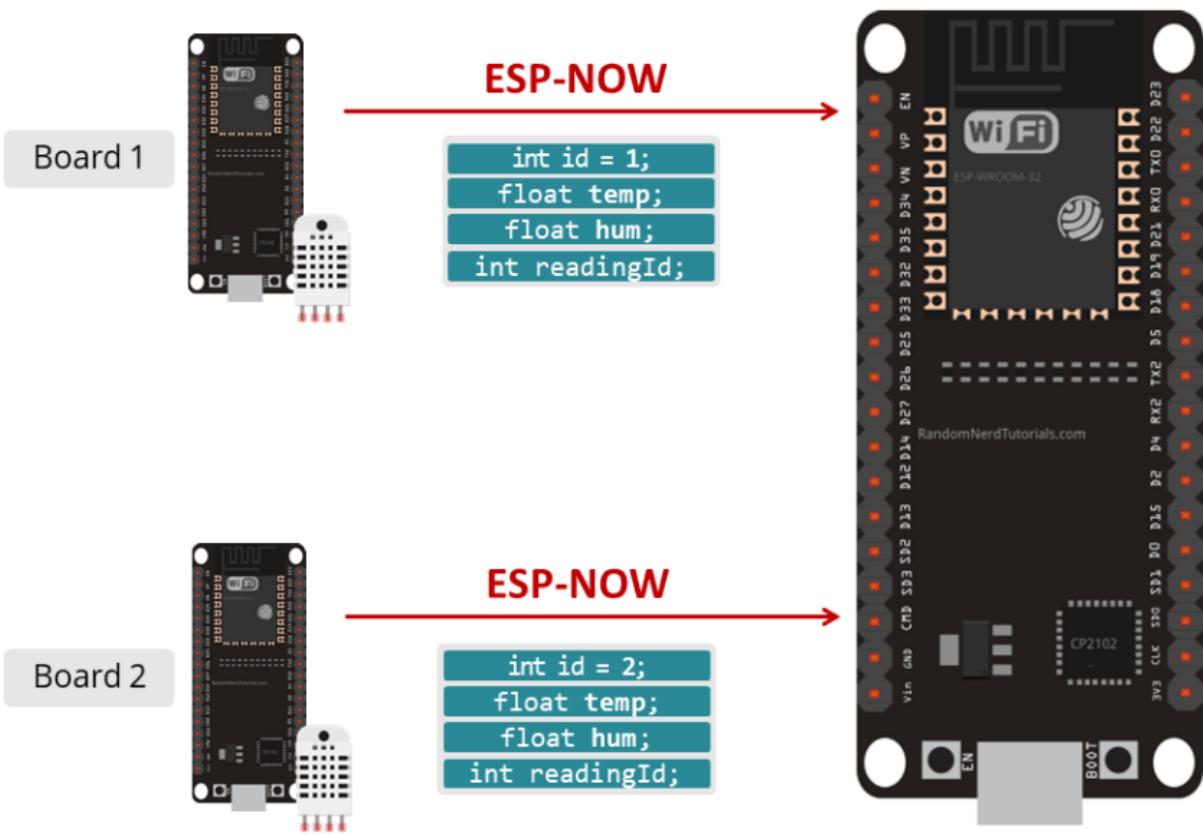
ESP32 Sender Circuit

The ESP32 sender boards are connected to a DHT22 temperature and humidity sensor. The data pin is connected to GPIO 4. You can choose any other suitable GPIO. Follow the next schematic diagram to wire the circuit.



ESP32 Sender Code (ESP-NOW)

Each sender board will send a structure via ESP-NOW that contains the board ID (so that you can identify which board sent the readings), the temperature, the humidity, and the reading ID. The reading ID is an int number to know how many messages were sent.



Upload the following code to each of your sender boards. Don't forget to increment the id number for each sender board.

- [Click here to download the code.](#)

```
#include <esp_now.h>
#include <esp_wifi.h>
#include <WiFi.h>
#include <Adafruit_Sensor.h>
#include <DHT.h>

// Set your Board ID (ESP32 Sender #1 = BOARD_ID 1, ESP32 Sender #2 = BOARD_ID 2
#define BOARD_ID 1

// Digital pin connected to the DHT sensor
#define DHTPIN 4

// Uncomment the type of sensor in use:
//#define DHTTYPE DHT11      // DHT 11
#define DHTTYPE DHT22      // DHT 22 (AM2302)
//#define DHTTYPE DHT21      // DHT 21 (AM2301)

DHT dht(DHTPIN, DHTTYPE);

//MAC Address of the receiver
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

// Variable to add info about peer
esp_now_peer_info_t peerInfo;

//Structure example to send data
//Must match the receiver structure
```

```

typedef struct struct_message {
    int id;
    float temp;
    float hum;
    int readingId;
} struct_message;

//Create a struct_message called myData
struct_message myData;

unsigned long previousMillis = 0; // Stores last time temperature was published
const long interval = 10000; // Interval at which to publish sensor readings

unsigned int readingId = 0;

// Insert your SSID
constexpr char WIFI_SSID[] = "REPLACE_WITH_YOUR_SSID";

int32_t getWiFiChannel(const char *ssid) {
    if (int32_t n = WiFi.scanNetworks()) {
        for (uint8_t i=0; i<n; i++) {
            if (!strcmp(ssid, WiFi.SSID(i).c_str())) {
                return WiFi.channel(i);
            }
        }
    }
    return 0;
}

float readDHTTemperature() {
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
    // Read temperature as Celsius (the default)
    float t = dht.readTemperature();
    // Read temperature as Fahrenheit (isFahrenheit = true)
    //float t = dht.readTemperature(true);
    // Check if any reads failed and exit early (to try again).
    if (isnan(t)) {
        Serial.println("Failed to read from DHT sensor!");
        return 0;
    }
    else {
        Serial.println(t);
        return t;
    }
}

float readDHTHumidity() {
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
    float h = dht.readHumidity();
    if (isnan(h)) {
        Serial.println("Failed to read from DHT sensor!");
        return 0;
    }
    else {
        Serial.println(h);
        return h;
    }
}

// callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {

```

```

Serial.print("\r\nLast Packet Send Status:\t");
Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" :
              "Delivery Fail");
}

void setup() {
    //Init Serial Monitor
    Serial.begin(115200);

    dht.begin();

    // Set device as a Wi-Fi Station and set channel
    WiFi.mode(WIFI_STA);

    int32_t channel = getWiFiChannel(WIFI_SSID);

    WiFi.printDiag(Serial); // Uncomment to verify channel number before
    esp_wifi_set_promiscuous(true);
    esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
    esp_wifi_set_promiscuous(false);
    WiFi.printDiag(Serial); // Uncomment to verify channel change after

    //Init ESP-NOW
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    // Once ESPNow is successfully Init, we will register for Send CB to
    // get the status of Trasnmited packet
    esp_now_register_send_cb(OnDataSent);

    //Register peer
    memcpy(peerInfo.peer_addr, broadcastAddress, 6);
    peerInfo.encrypt = false;

    //Add peer
    if (esp_now_add_peer(&peerInfo) != ESP_OK){
        Serial.println("Failed to add peer");
        return;
    }
}

void loop() {
    unsigned long currentMillis = millis();
    if (currentMillis - previousMillis >= interval) {
        // Save the last time a new reading was published
        previousMillis = currentMillis;
        //Set values to send
        myData.id = BOARD_ID;
        myData.temp = readDHTTemperature();
        myData.hum = readDHTHumidity();
        myData.readingId = readingId++;

        //Send message via ESP-NOW
        esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &myData,
                                       sizeof(myData));
        if (result == ESP_OK) {
            Serial.println("Sent with success");
        }
        else {

```

```
        Serial.println("Error sending the data");
    }
}
```

How Does the Code Work?

Start by importing the required libraries:

```
#include <esp_now.h>
#include <esp_wifi.h>
#include <WiFi.h>
#include <Adafruit_Sensor.h>
#include <DHT.h>
```

Set Board ID

Define the ESP32 sender board ID, for example set BOARD_ID 1 for ESP32 Sender #1, etc...

```
// Set your Board ID (ESP32 Sender #1 = BOARD_ID 1, ESP32 Sender #2 = BOARD_ID 2
#define BOARD_ID 1
```

DHT Sensor

Define the pin the DHT sensor is connected to. In our example, it is connected to GPIO 4.

```
#define DHTPIN 4
```

Select the DHT sensor type you're using. We're using the DHT22.

```
//#define DHTTYPE DHT11 // DHT 11
#define DHTTYPE DHT22 // DHT 22 (AM2302)
//#define DHTTYPE DHT21 // DHT 21 (AM2301)
```

Create a DHT object on the pin and type defined earlier.

```
DHT dht(DHTPIN, DHTTYPE);
```

Receiver's MAC Address

Insert the receiver's MAC address on the next line (for example):

```
uint8_t broadcastAddress[] = {0x3C, 0x71, 0xBF, 0xC3, 0xBF, 0xB0};
```

Data Structure

Then, create a structure that contains the data we want to send. The `struct_message` contains the board ID, temperature reading, humidity reading, and the reading ID.

```
typedef struct struct_message {
    int id;
    float temp;
    float hum;
    unsigned int readingId;
} struct_message;
```

Create a new variable of type `struct_message` that is called `myData` that stores the variables' values.

```
struct_message myData;
```

Timer Interval

Create some auxiliary timer variables to publish the readings every 10 seconds.

You can change the delay time on the `interval` variable.

```
unsigned long previousMillis = 0; // Stores last time temperature was published
const long interval = 10000; // Interval at which to publish sensor readings
```

Initialize the `readingId` variable – it keeps track of the number of readings sent.

```
unsigned int readingId = 0;
```

Changing Wi-Fi channel

Now, we'll get the receiver's Wi-Fi channel. This is useful because it allows us to automatically assign the same Wi-Fi channel to the sender board.

To do that, you must insert your SSID in the following line:

```
constexpr char WIFI_SSID[] = "REPLACE_WITH_YOUR_SSID";
```

Then, the `getWiFiChannel()` function scans for your network and gets its channel.

```
int32_t getWiFiChannel(const char *ssid) {
    if (int32_t n = WiFi.scanNetworks()) {
        for (uint8_t i=0; i<n; i++) {
            if (!strcmp(ssid, WiFi.SSID(i).c_str())) {
                return WiFi.channel(i);
            }
        }
    }
}
```

```
    return 0;
}
```

Reading Temperature

The `readDHTTemperature()` function reads and returns the temperature from the DHT sensor. In case it is not able to get temperature readings, it returns 0.

```
float readDHTTemperature() {
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
    // Read temperature as Celsius (the default)
    float t = dht.readTemperature();
    // Read temperature as Fahrenheit (isFahrenheit = true)
    //float t = dht.readTemperature(true);
    // Check if any reads failed and exit early (to try again).
    if (isnan(t)) {
        Serial.println("Failed to read from DHT sensor!");
        return 0;
    }
    else {
        Serial.println(t);
        return t;
    }
}
```

Reading Humidity

The `readDHTHumidity()` function reads and returns the humidity from the DHT sensor. In case it is not able to get humidity readings, it returns 0.

```
float readDHTHumidity() {
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
    float h = dht.readHumidity();
    if (isnan(h)) {
        Serial.println("Failed to read from DHT sensor!");
        return 0;
    }
    else {
        Serial.println(h);
        return h;
    }
}
```

OnDataSent Callback Function

The `OnDataSent()` callback function will be executed when a message is sent. In this case, this function prints if the message was successfully delivered or not.

```
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
    Serial.print("\r\nLast Packet Send Status:\t");
    Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" :
                  "Delivery Fail");
}
```

setup()

Initialize the Serial Monitor.

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

Set the ESP32 as a Wi-Fi station.

```
WiFi.mode(WIFI_STA);
```

Get the Wi-Fi channel of the network (the Wi-Fi channel the receiver board is using) and set this board to use the same Wi-Fi channel.

```
int32_t channel = getWiFiChannel(WIFI_SSID);  
  
WiFi.printDiag(Serial); // Uncomment to verify channel number before  
esp_wifi_set_promiscuous(true);  
esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);  
esp_wifi_set_promiscuous(false);  
WiFi.printDiag(Serial); // Uncomment to verify channel change after
```

Initialize ESP-NOW.

```
//Init 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, register for the `OnDataSent()` function created previously.

```
esp_now_register_send_cb(OnDataSent);
```

Add peer

To send data to another board (the receiver), you need to pair it as a peer. The following lines register and add the receiver as a peer.

```
//Register peer  
memcpy(peerInfo.peer_addr, broadcastAddress, 6);  
peerInfo.encrypt = false;  
  
//Add peer  
if (esp_now_add_peer(&peerInfo) != ESP_OK){  
    Serial.println("Failed to add peer");  
    return;  
}
```

loop()

In the `loop()`, check if it is time to get and send new readings.

```
void loop() {
    unsigned long currentMillis = millis();
    if (currentMillis - previousMillis >= interval) {
        // Save the last time a new reading was published
        previousMillis = currentMillis;
```

Send ESP-NOW Message

Update the data structure with the current sensor readings, board ID and reading ID.

```
//Set values to send
myData.id = BOARD_ID;
myData.temp = readDHTTemperature();
myData.hum = readDTHumidity();
myData.readingId = readingId++;
```

Finally, send the data structure via ESP-NOW.

```
//Send message via ESP-NOW
esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &myData,
                                sizeof(myData));
if (result == ESP_OK) {
    Serial.println("Sent with success");
}
else {
    Serial.println("Error sending the data");
}
```

Demonstration

Upload the code to your sender boards. You should notice that the boards change their Wi-Fi channel to the channel of the receiver board. In our case, the boards changed their Wi-Fi channel number to 12.

Output Serial Monitor

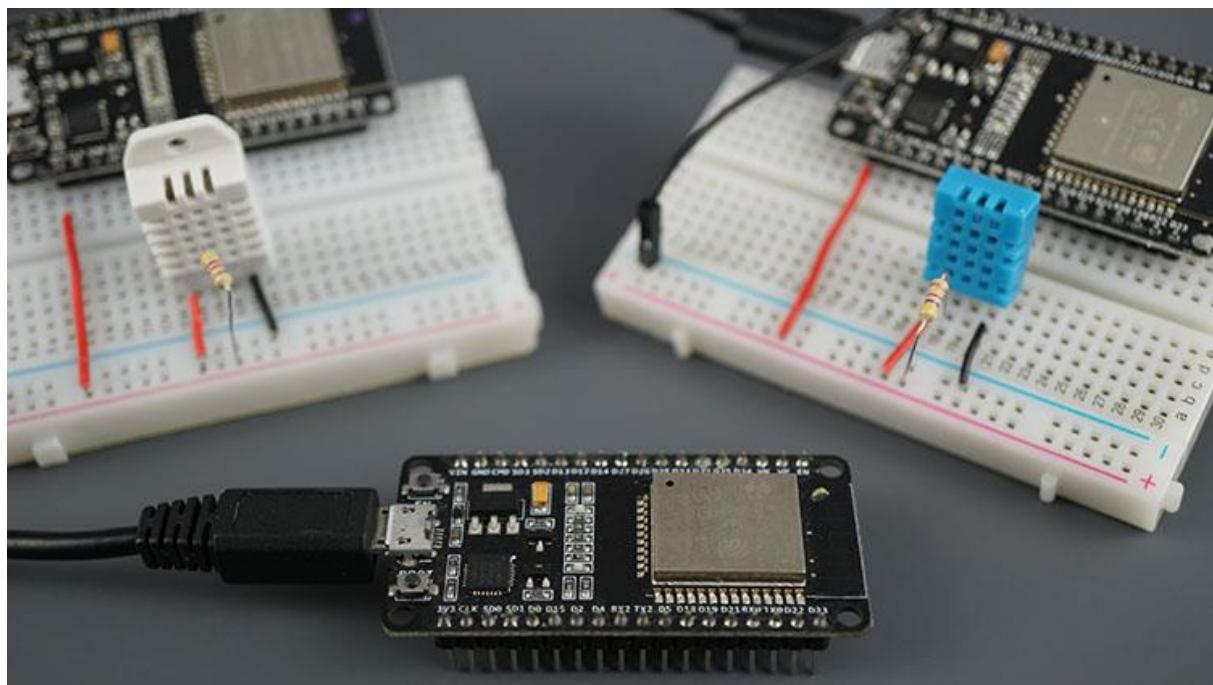
Message (Enter to send message to 'DOIT ESP32 DEVKIT ...')

New Line 115200 baud

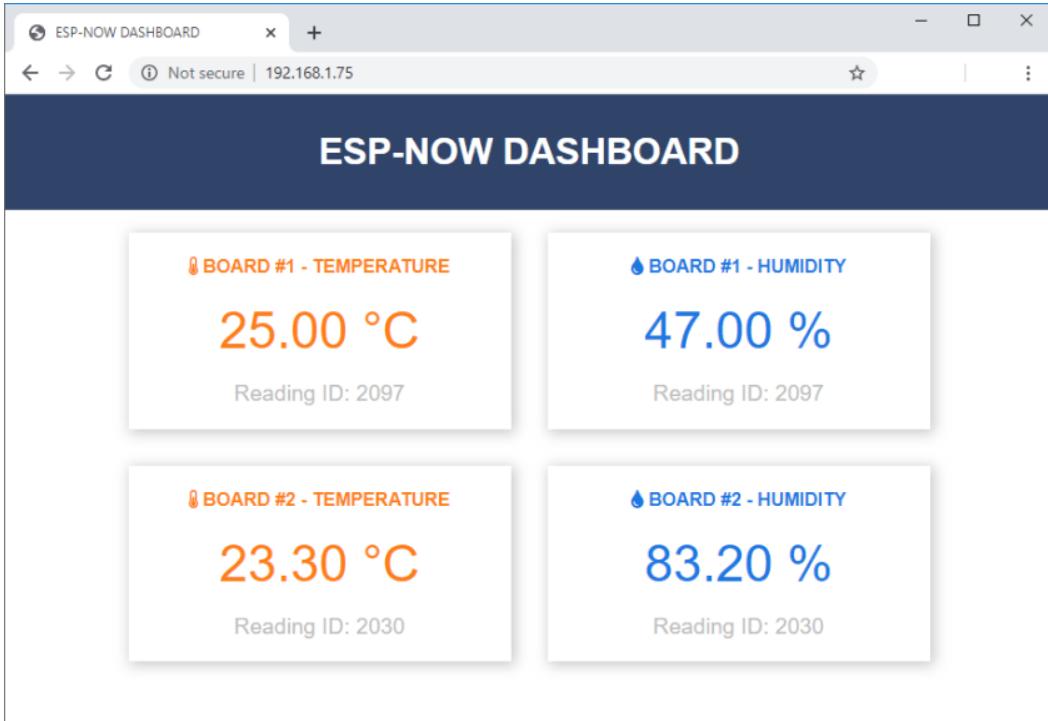
```
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Mode: STA
Channel: 1
SSID (0):
Passphrase (0):
BSSID set: 0
Mode: STA
Channel: 12
SSID (0):
Passphrase (0):
BSSID set: 0
Mode: STA
```

Ln 48, Col 1 DOIT ESP32 DEVKIT V1 on COM9 2

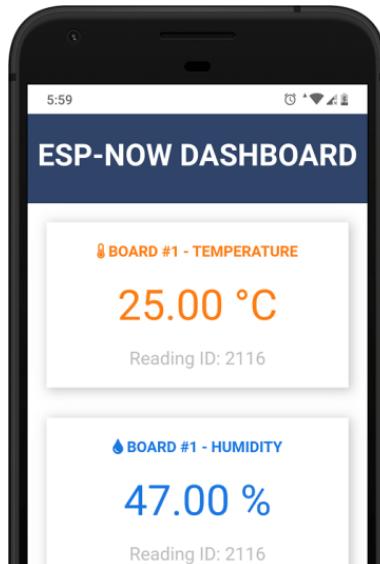
If everything goes as expected, the ESP32 receiver board should start receiving sensor readings from the other boards.



Open a browser on your local network and type the ESP32 receiver IP address.



It should load the temperature, humidity, and reading IDs for each board. Upon receiving a new packet, your web page updates automatically without refreshing the web page.



Wrapping Up

In this tutorial you've learned how to use ESP-NOW and Wi-Fi to set up a web server to receive ESP-NOW packets from multiple boards (many-to-one configuration).

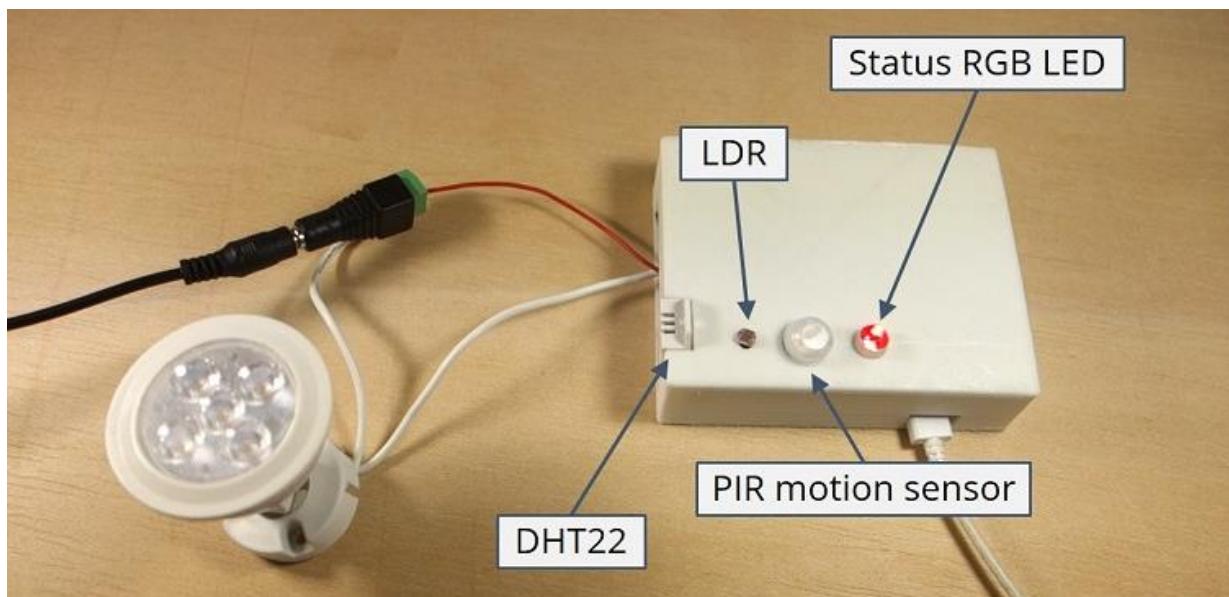
Additionally, you also used Server-Sent Events to automatically update the web page every time a new packet is received without refreshing the web page.

PROJECT 1

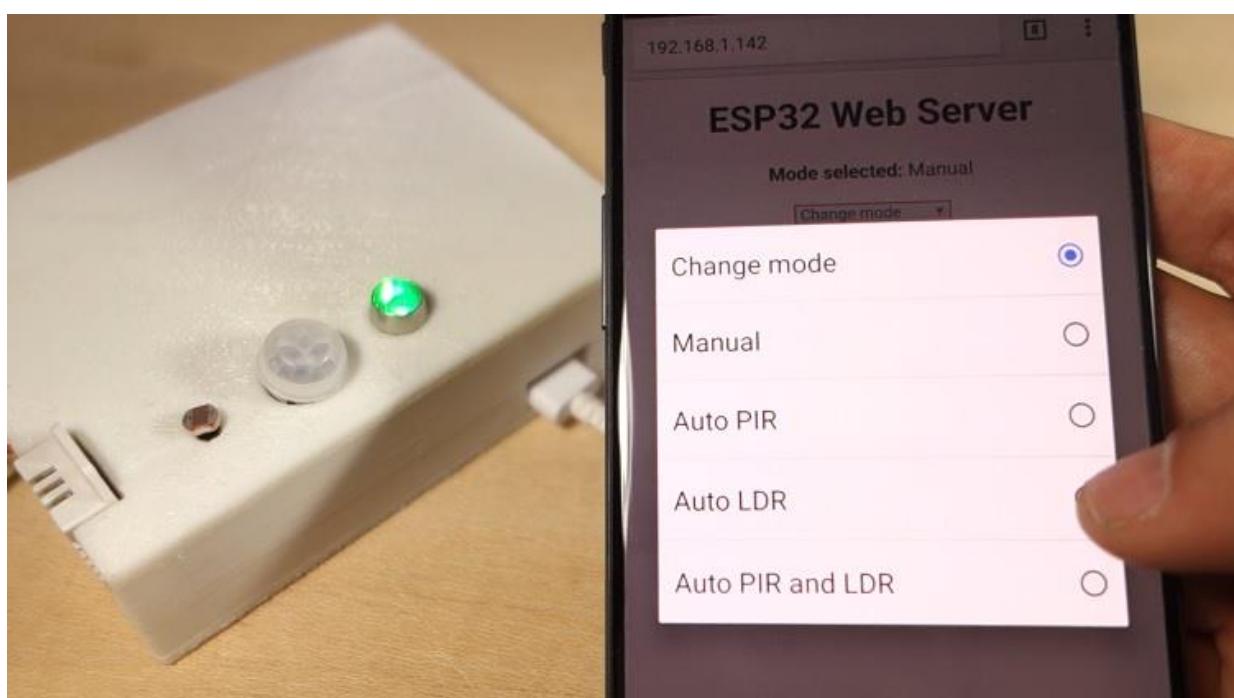
**ESP32 Wi-Fi Multisensor:
Temperature, Humidity, Motion,
Luminosity, and Relay Control**

Unit 1 - ESP32 Wi-Fi Multisensor: Temperature, Humidity, Motion, Luminosity, and Relay Control

In this project, we're going to show you how to create an ESP32 Wi-Fi Multisensor. This device consists of a PIR motion sensor, a light-dependent resistor (LDR), a DHT22 temperature and humidity sensor, a relay, and a status RGB LED.



We'll build a web server that allows you to monitor your sensors and control your relay based on several configurable conditions.



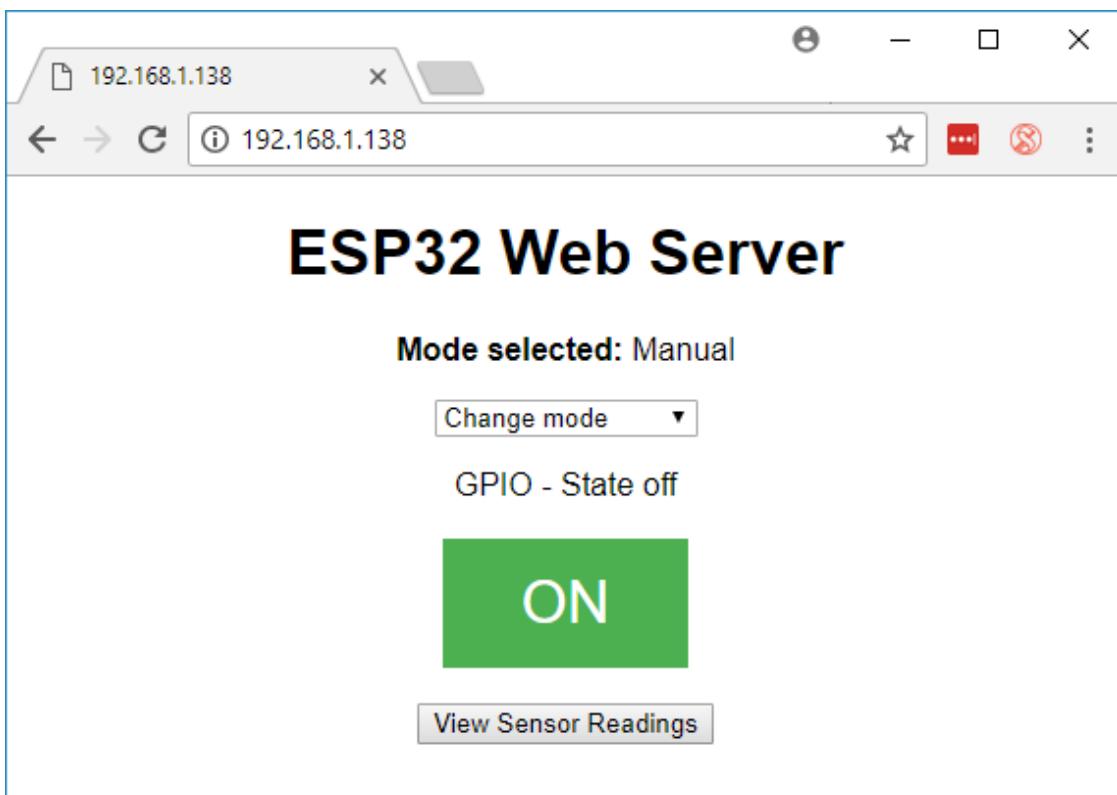
The code for this project is quite long, so we'll divide it into two Units:

- In this Unit, we'll show you how to build the complete Wi-Fi Multisensor step-by-step.
- In the next Unit, we'll explain how the code works.

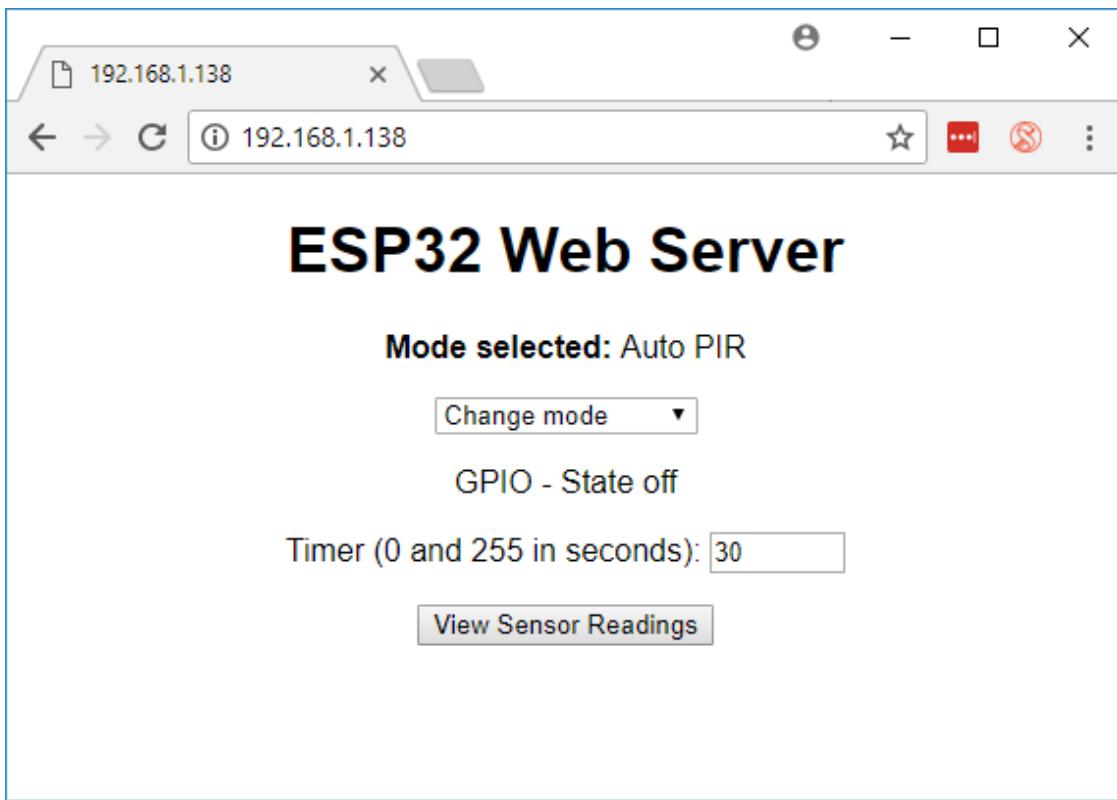
Project Overview

The web server for this project allows you to choose between 4 different modes to control the relay:

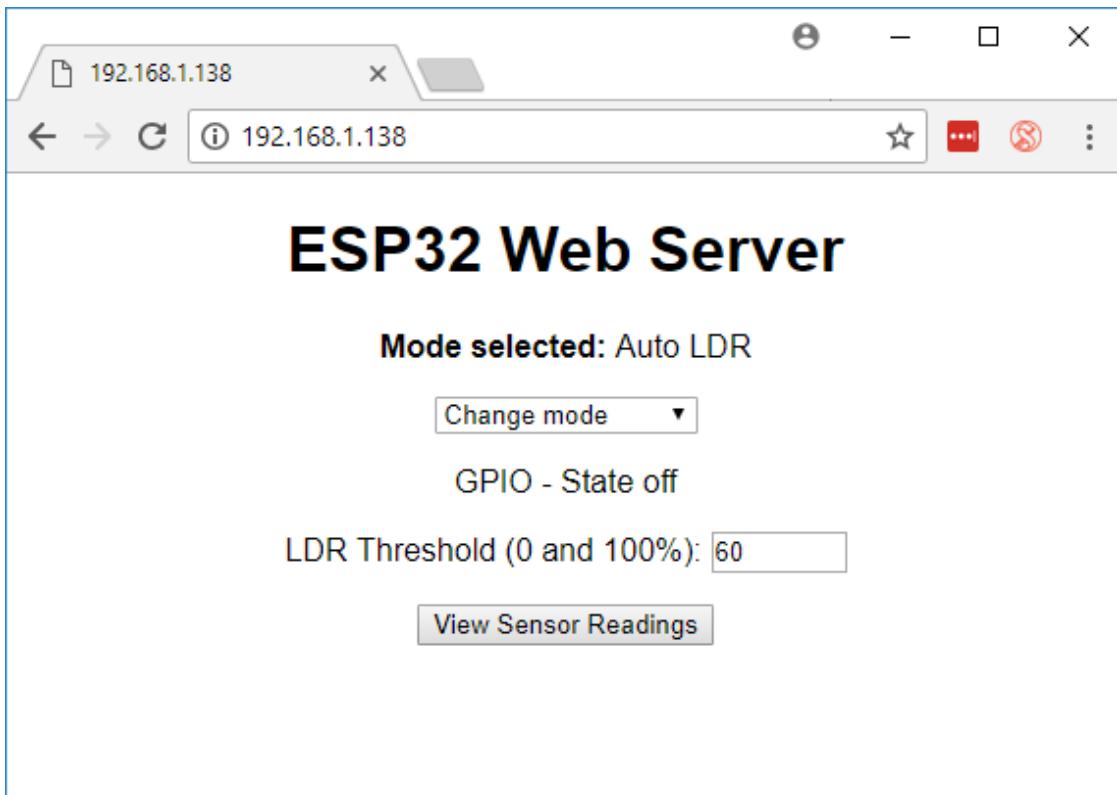
- **Manual** (mode 0): you have a button to turn a relay on and off.



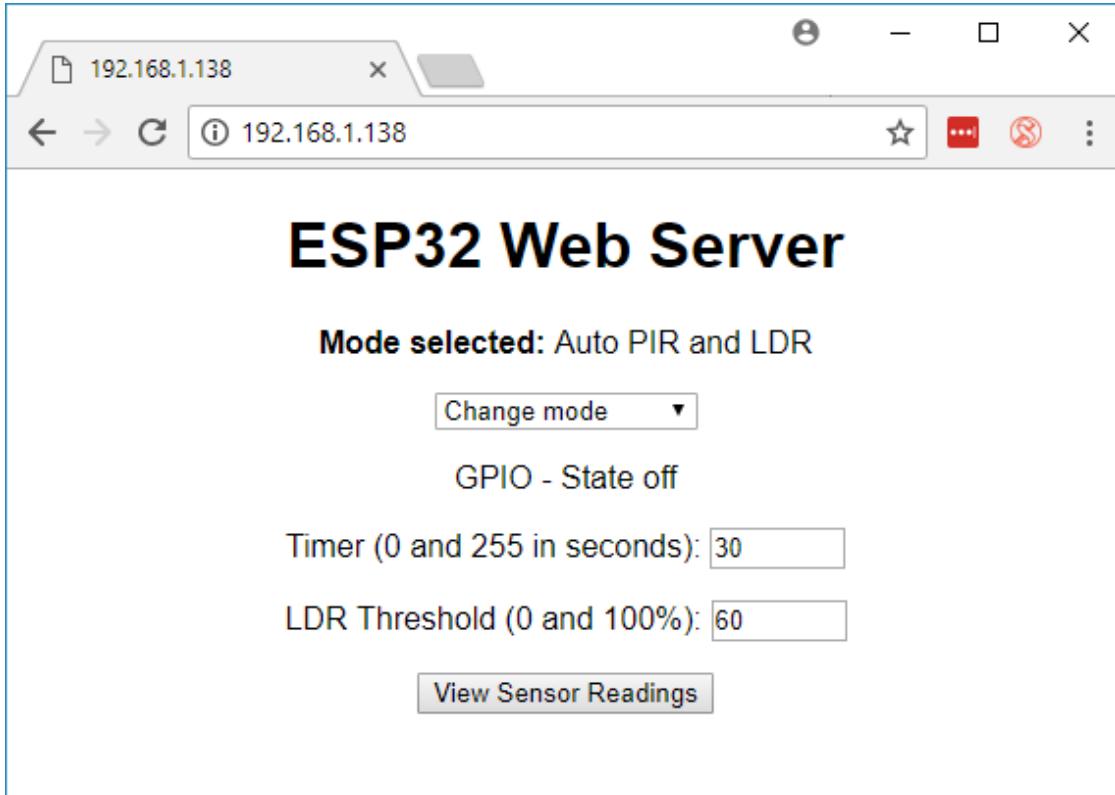
- **Auto PIR** (mode 1): turns the relay on when motion is detected. In this mode, there is a field in which you can set the number of seconds the output will be on after motion is detected.



- **LDR** (mode 2): the relay turns on when the luminosity goes below a certain threshold. You can set an LDR threshold value between 0 and 100%.

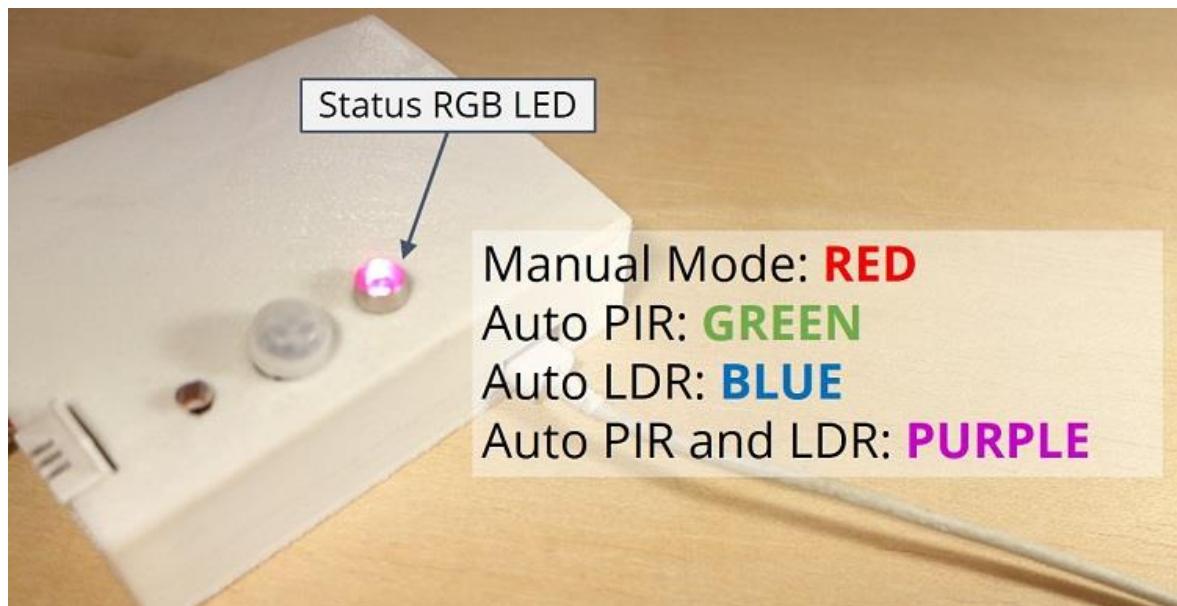


- **Auto PIR and LDR** (mode 3): this mode combines the PIR motion sensor and the LDR. When this mode is selected, the relay turns on when the PIR sensor detects motion and if the luminosity value is below the threshold. In this mode, you can set the timer and the LDR threshold value.



Status indicator

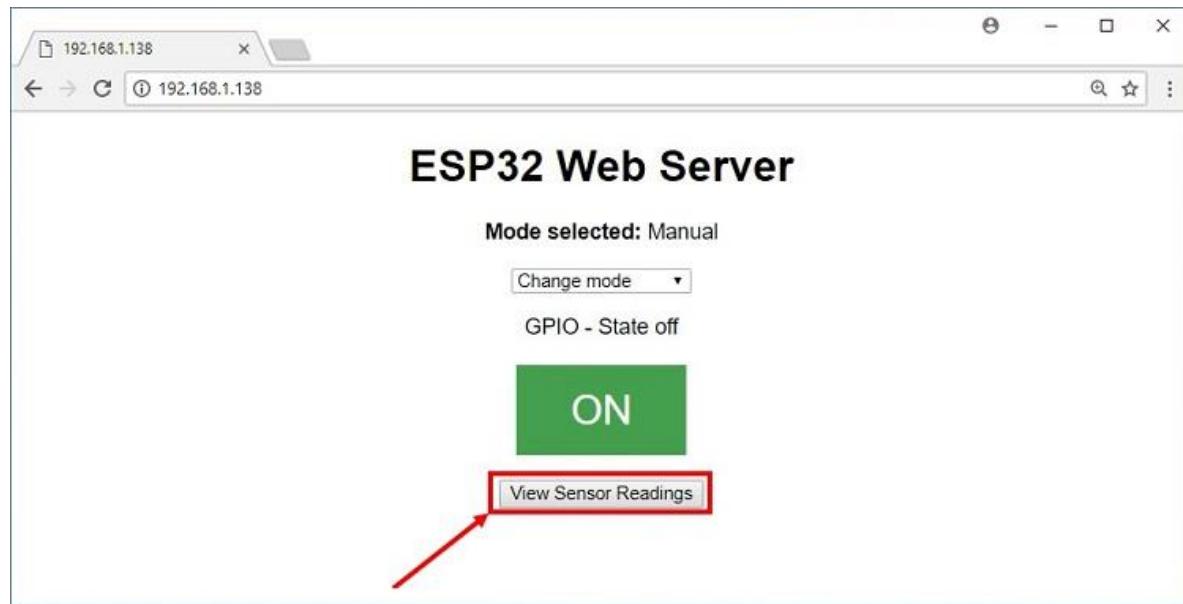
The device contains an RGB LED that changes color accordingly to the selected mode:



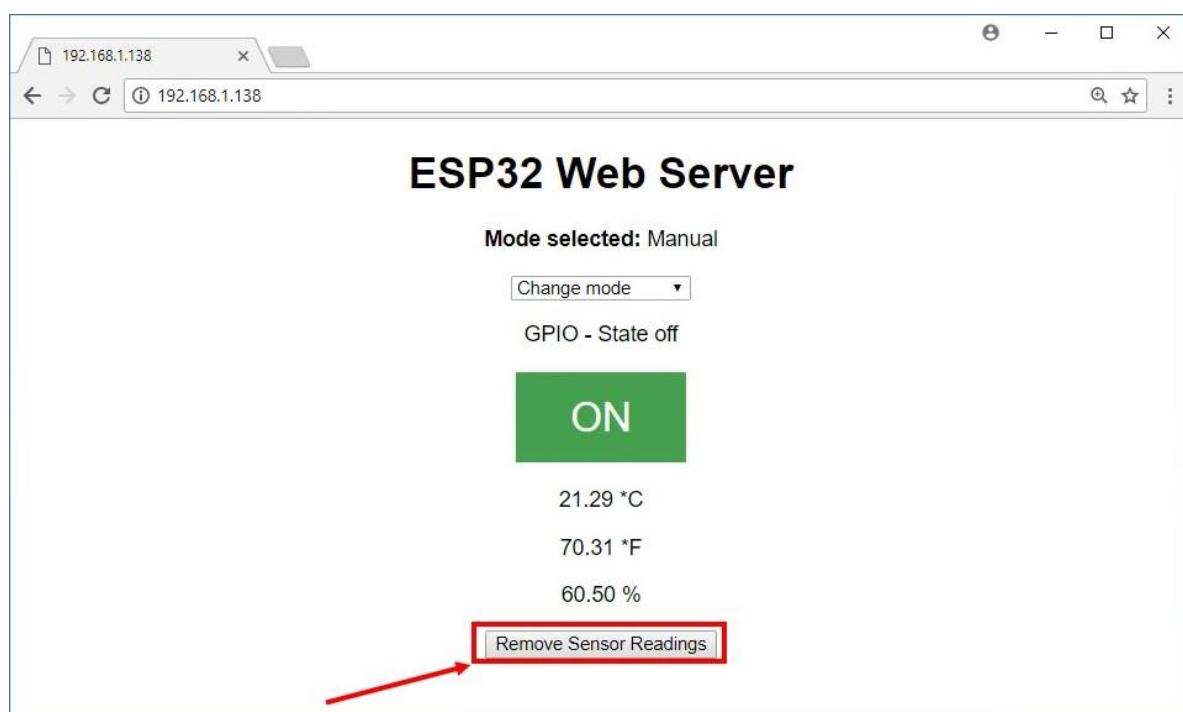
- Manual: red
- Auto PIR: green
- Auto LDR: blue
- Auto PIR and LDR: purple

Sensor readings

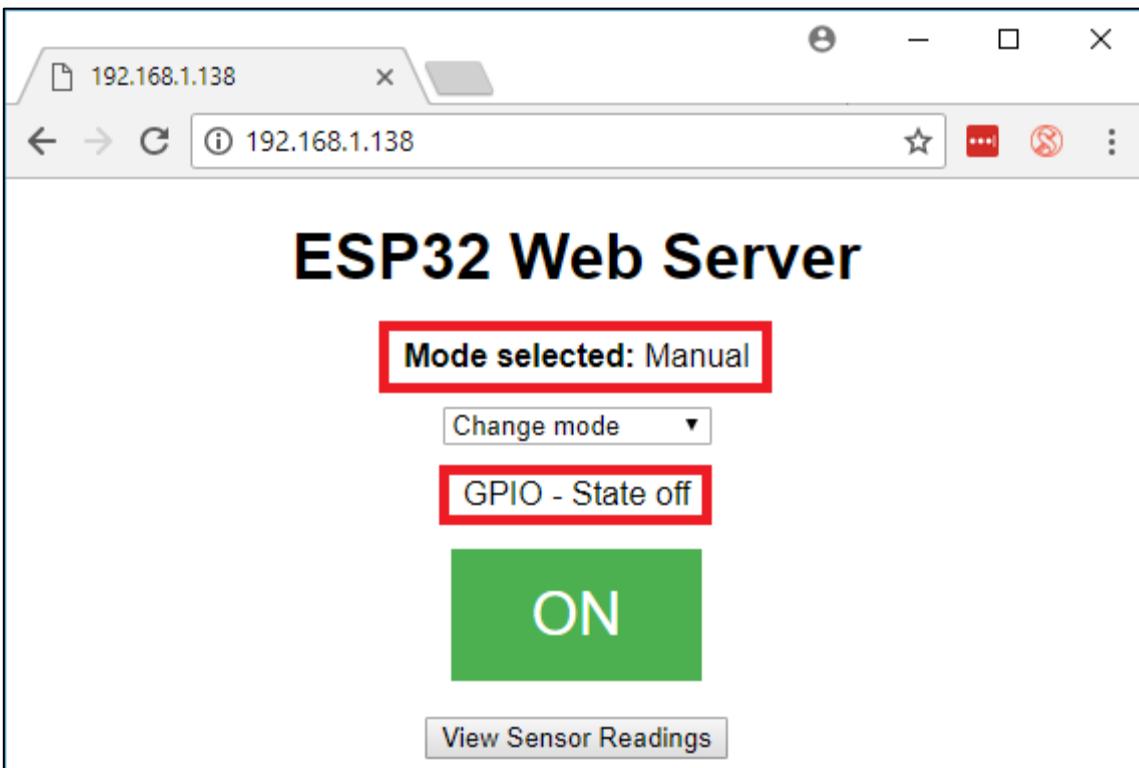
In the web server, there's also a button that you can press to request the temperature and humidity readings.



You can press the “Remove Sensor Readings” button to hide the readings.



In every mode, there's a label that shows the selected mode and the current output state as highlighted in the following figure.



We want the ESP32 to remember the last output state and the settings, in case it resets or suddenly loses power. So, we need to save those parameters in the ESP32 flash memory.

Building the Sensor Node

Now that you know what you're going to build, let's create the ESP32 Wi-Fi Multisensor.

Parts Required:

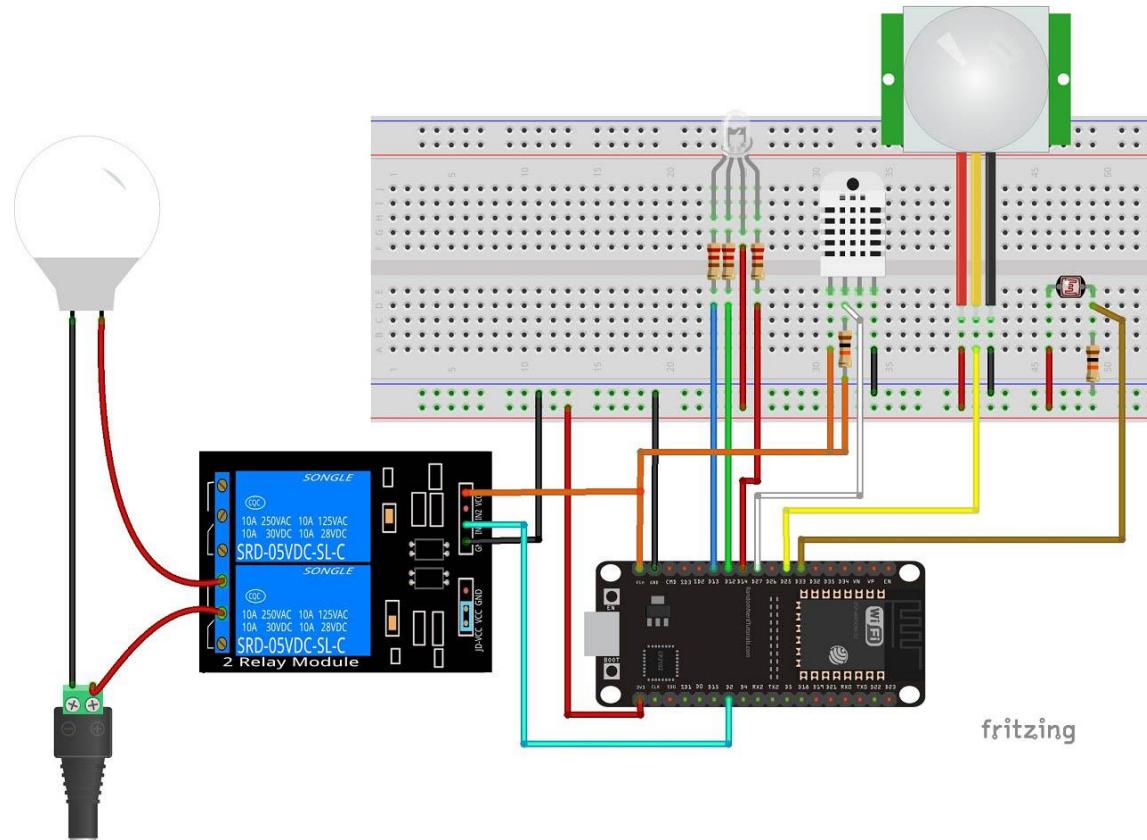
Here's a list of the parts required to build this project:

- [ESP32 DOIT DEVKIT V1 board](#)
- [DHT22 temperature and humidity sensor](#)
- [Mini PIR motion sensor](#)
- [Light dependent resistor \(LDR\)](#)
- [RGB LED common anode](#)
- [2x 10K Ohm resistor](#)

- [3x 220 Ohm resistor \(or similar values\)](#)
- [1x LED holder](#)
- [Relay module](#)
- 12V Lamp
- [12V power source](#)
- [Breadboard](#)
- [Prototyping circuit board](#)
- [Jumper wires](#)
- [Project box enclosure](#) (or 3D print your own case)

Start by wiring the circuit as shown in the following schematic diagram. Wire the PIR motion sensor data pin to GPIO 25. We'll read the value from the LDR on GPIO 33. The temperature and humidity sensor data pin goes to GPIO 27, and the RGB LED leads go to GPIOs 12, 13, and 14.

Our multisensor controls an output connected to GPIO 2. We're using a relay to control a lamp, but you can connect any other output that best suits your project needs.



For better guidance, you can take a look at the following tables:

PIR Motion Sensor Wiring

PIR Motion Sensor	ESP32
VCC	3.3V (or 5V depending on your PIR Motion sensor)
Data	GPIO 25
GND	GND

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.

RGB LED Wiring

RGB LED	ESP32
Red lead	GPIO 14
Common Anode	3.3V
Green lead	GPIO 12
Blue lead	GPIO 13

DHT22 Wiring

DHT22	ESP32
Pin 1	Vin (5V)
Pin 2	GPIO 27 and 10K pull-up resistor
Pin 3	Don't connect
Pin 4	GND

LDR Wiring

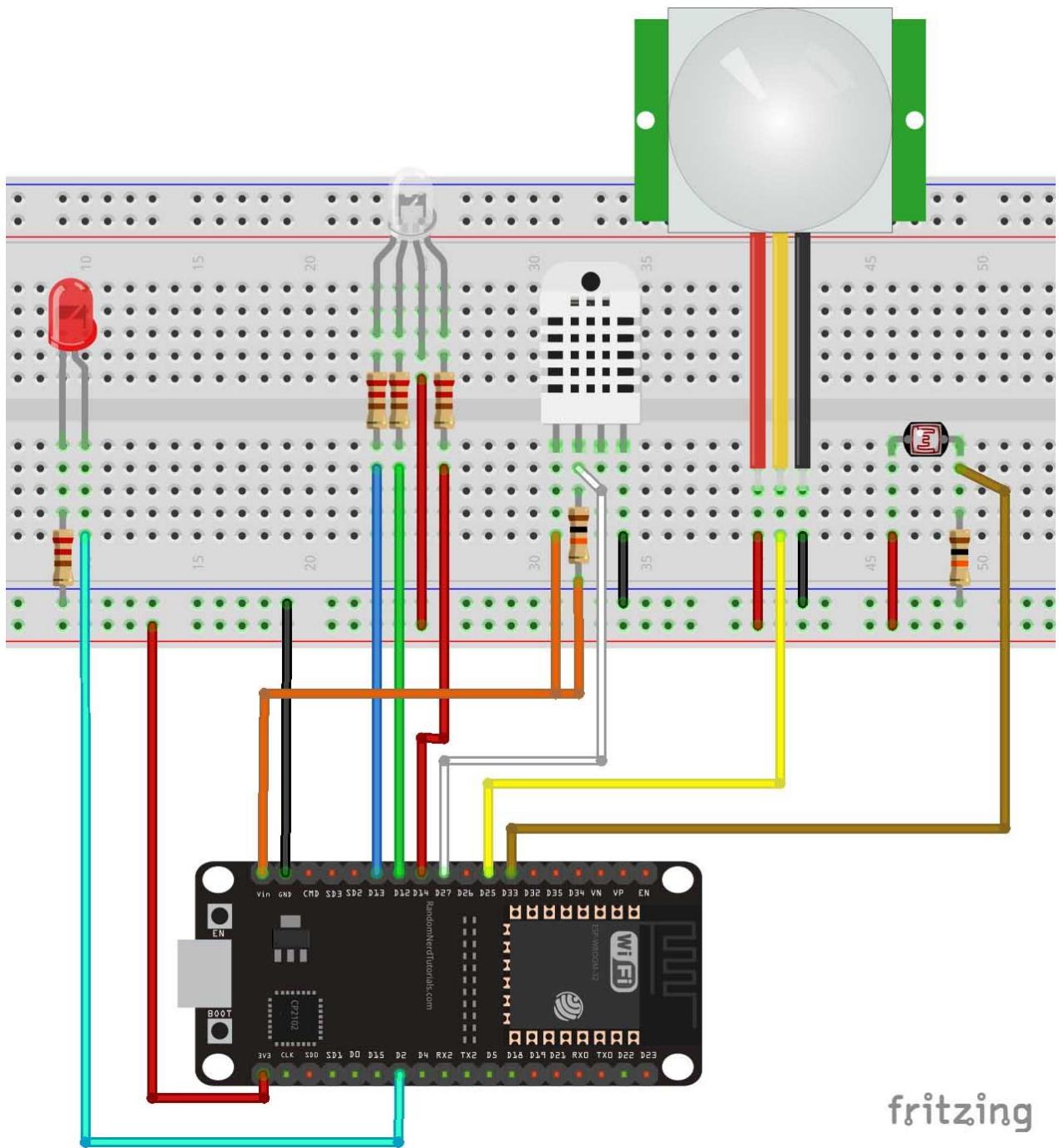
LDR	ESP32
Pin 1	3.3V
Pin 2	GPIO 33 and 10K pull-down resistor

Relay Wiring

Relay	ESP32
VCC	Vin (5V)
IN1	GPIO 2
IN2	Don't connect
GND	GND

Note: for testing purposes, you can use an LED as an output. However, note that we'll be using a relay, so the logic in our code is inverted. If you try to turn the LED on, it will turn off and vice-versa.

The following figure shows the circuit diagram you need to follow to test the setup with an LED.



Installing the Adafruit DHT Library

Before uploading the code, you need to install the Adafruit_DHT library in your Arduino IDE. Follow the next instructions to install it if you haven't already:

1. Go **Sketch > Include Library > Manage Libraries**, and type "DHT sensor library" to search for the library. Then install the DHT sensor library by Adafruit.
2. A pop-up window will open asking you to install the Adafruit Unified Sensor library. Install that library and any other dependencies if you haven't already.

Uploading the Sensor Node Code

After installing the required libraries, copy the following code to your Arduino IDE.

- [Click here to download the code.](#)

```
// Load libraries
#include <WiFi.h>
#include <Preferences.h>
#include "DHT.h"
#include <Adafruit_Sensor.h>

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

// Uncomment one of the lines below for whatever DHT sensor type you're using!
//#define DHTTYPE DHT11    // DHT 11
//#define DHTTYPE DHT21    // DHT 21 (AM2301)
#define DHTTYPE DHT22    // DHT 22  (AM2302), AM2321

// DHT Sensor
const int DHTPin = 27;
// Initialize DHT sensor.
DHT dht(DHTPin, DHTTYPE);

// Temporary variables for temperature and humidity
static char celsiusTemp[7];
static char fahrenheitTemp[7];
static char humidityTemp[7];

// Preferences object
Preferences preferences;

// Set GPIOs for: output variable, RGB LED, PIR Motion Sensor, and LDR
const int output = 2;
const int redRGB = 14;
const int greenRGB = 12;
```

```

const int blueRGB = 13;
const int motionSensor = 25;
const int ldr = 33;
// Store the current output state
String outputState = "off";

// Timers - Auxiliary variables
long now = millis();
long lastMeasure = 0;
boolean startTimer = false;

// Auxiliary variables to store selected mode and settings
int selectedMode = 0;
int timer = 0;
int ldrThreshold = 0;
int armMotion = 0;
int armLdr = 0;
String modes[4] = { "Manual", "Auto PIR", "Auto LDR", "Auto PIR and LDR" };

// Decode HTTP GET value
String valueString = "0";
int pos1 = 0;
int pos2 = 0;
// Variable to store the HTTP request
String header;
// Set web server port number to 80
WiFiServer server(80);

// Current time
unsigned long currentTime = millis();
// Previous time
unsigned long previousTime = 0;
// Define timeout time in milliseconds (example: 2000ms = 2s)
const long timeoutTime = 2000;

void setup() {
    // initialize the DHT sensor
    dht.begin();

    // Serial port for debugging purposes
    Serial.begin(115200);

    // PIR Motion Sensor mode, then set interrupt function and RISING mode
    pinMode(motionSensor, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(motionSensor),
                    detectsMovement, RISING);

    Serial.println("start...");

    // Initialize Preferences
    preferences.begin("settings", false);

    // Initialize the output variable and RGB pins as OUTPUTS
    pinMode(output, OUTPUT);
    pinMode(redRGB, OUTPUT);
    pinMode(greenRGB, OUTPUT);
    pinMode(blueRGB, OUTPUT);

    // Read from flash memory on start and store the values in auxiliary variables
    // Set output to last state (saved in the flash memory)
    if(preferences.getInt("outputState", 0) == 0) {

```

```

        outputState = "off";
        digitalWrite(output, HIGH);
    }
    else {
        outputState = "on";
        digitalWrite(output, LOW);
    }

    selectedMode = preferences.getInt("selectedMode", 0);
    timer = preferences.getInt("timer", 0);
    ldrThreshold = preferences.getInt("ldrThreshold", 0);
    configureMode();

    // 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();
}

void loop() {
    WiFiClient client = server.available(); // Listen for incoming clients
    if (client) { // If a new client connects,
        currentTime = millis();
        previousTime = currentTime;
        Serial.println("New Client."); // print a message out in the serial port
        String currentLine = ""; // make a String to hold incoming data from the client
        // loop while the client's connected
        while (client.connected() && currentTime - previousTime <= timeoutTime) {
            currentTime = millis();
            if (client.available()) { // if there's bytes to read from the client,
                char c = client.read(); // read a byte, then
                Serial.write(c); // print it out the serial monitor
                header += c;
                if (c == '\n') { // if the byte is a newline character
                    // if the current line is blank, you got two newline characters in a row
                    // that's the end of the client HTTP request, so send a response:
                    if (currentLine.length() == 0) {
                        // HTTP headers always start with a response code (e.g. HTTP/1.1 200 OK)
                        // and a content-type so the client knows what's coming, then a blank line:
                        client.println("HTTP/1.1 200 OK");
                        client.println("Content-type:text/html");
                        client.println("Connection: close");
                        client.println();
                        // Display the HTML web page
                        client.println("<!DOCTYPE html><html>");
                        client.println("<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">");
                        client.println("<link rel=\"icon\" href=\"data:,\">");
                        // CSS to style the on/off buttons
                        // Feel free to change the background-color and font-size attributes to fit your preferences
                        client.println("<style>html { font-family: Helvetica;
```

```

        display: inline-block; margin: 0px auto; text-align: center;}");
client.println(".button { background-color: #4CAF50; border: none;
                      color: white; padding: 16px 40px;}");
client.println("text-decoration: none; font-size: 30px; margin: 2px;
                      cursor: pointer;}");
client.println(".button2 {background-color: #555555;}"
                </style></head>");

// Request example: GET /?mode=0& HTTP/1.1 - sets mode to Manual (0)
if(header.indexOf("GET /?mode=") >= 0) {
    pos1 = header.indexOf('=');
    pos2 = header.indexOf('&');
    valueString = header.substring(pos1+1, pos2);
    selectedMode = valueString.toInt();
    preferences.putInt("selectedMode", selectedMode);
    configureMode();
}
// Change the output state - turn GPIOs on and off
else if(header.indexOf("GET /?state=on") >= 0) {
    outputOn();
}
else if(header.indexOf("GET /?state=off") >= 0) {
    outputOff();
}
// Set timer value
else if(header.indexOf("GET /?timer=") >= 0) {
    pos1 = header.indexOf('=');
    pos2 = header.indexOf('&');
    valueString = header.substring(pos1+1, pos2);
    timer = valueString.toInt();
    preferences.putInt("timer", timer);
    Serial.println(valueString);
}
// Set LDR Threshold value
else if(header.indexOf("GET /?ldrthreshold=") >= 0) {
    pos1 = header.indexOf('=');
    pos2 = header.indexOf('&');
    valueString = header.substring(pos1+1, pos2);
    ldrThreshold = valueString.toInt();
    preferences.putInt("ldrThreshold", ldrThreshold);
    Serial.println(valueString);
}

// Web Page Heading
client.println("<body><h1>ESP32 Web Server</h1>");
// Drop down menu to select mode
client.println("<p><strong>Mode selected:</strong> " +
                  + modes[selectedMode] + "</p>");
client.println("<select id=\"mySelect\" " +
                  onchange=\"setMode(this.value)\">");
client.println("<option>Change mode");
client.println("<option value=\"0\">Manual");
client.println("<option value=\"1\">Auto PIR");
client.println("<option value=\"2\">Auto LDR");
client.println("<option value=\"3\">Auto PIR and LDR</select>");

// Display current state, and ON/OFF buttons for output
client.println("<p>GPIO - State " + outputState + "</p>");
// If the output is off, it displays the ON button
if(selectedMode == 0) {
    if(outputState == "off") {

```

```

        client.println("<p><button class=\"button\""
                      onclick=\"outputOn()\">ON</button></p>");
    }
    else {
        client.println("<p><button class=\"button button2\""
                      onclick=\"outputOff()\">OFF</button></p>");
    }
}
else if(selectedMode == 1) {
    client.println("<p>Timer (0 and 255 in seconds): <input"
                  type="number" name="txt" value="" +
                  String(preferences.getInt("timer", 0)) + "<" +
                  onchange="setTimer(this.value)" min="0" max="255"></p>");
}
else if(selectedMode == 2) {
    client.println("<p>LDR Threshold (0 and 100%):" +
                  <input type="number" name="txt" value="" +
                  String(preferences.getInt("ldrThreshold", 0)) + "<" +
                  onchange="setThreshold(this.value)" min="0" max="100"></p>");
}
else if(selectedMode == 3) {
    client.println("<p>Timer (0 and 255 in seconds):" +
                  <input type="number" name="txt" value="" +
                  String(preferences.getInt("timer", 0)) +
                  "\<" onchange="setTimer(this.value)" min="0" max="255"></p>");

    client.println("<p>LDR Threshold (0 and 100%):" +
                  <input type="number" name="txt" value="" +
                  String(preferences.getInt("ldrThreshold", 0)) +
                  "\<" onchange="setThreshold(this.value)" min="0" max="100"></p>");

}
// Get and display DHT sensor readings
if(header.indexOf("GET /?sensor") >= 0) {
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
    float h = dht.readHumidity();
    // Read temperature as Celsius (the default)
    float t = dht.readTemperature();
    // Read temperature as Fahrenheit (isFahrenheit = true)
    float f = dht.readTemperature(true);
    // Check if any reads failed and exit early (to try again).
    if (isnan(h) || isnan(t) || isnan(f)) {
        Serial.println("Failed to read from DHT sensor!");
        strcpy(celsiusTemp, "Failed");
        strcpy(fahrenheitTemp, "Failed");
        strcpy(humidityTemp, "Failed");
    }
    else {
        // Computes temperature values in Celsius + Fahrenheit and Humidity
        dtostrf(t, 6, 2, celsiusTemp);
        dtostrf(f, 6, 2, fahrenheitTemp);
        dtostrf(h, 6, 2, humidityTemp);
    }
    client.println("<p>");
    client.println(celsiusTemp);
    client.println("*C</p><p>");
    client.println(fahrenheitTemp);
    client.println("*F</p></div><p>");
    client.println(humidityTemp);
    client.println("%</p></div>");
    client.println("<p><a href=\"/\"><button>Remove Sensor Readings
                  </button></a></p>");
```

```

    }
    else {
        client.println("<p><a href=\"?sensor\"><button>View Sensor
                      Readings</button></a></p>");
    }
    client.println("<script> function setMode(value) { var xhr = new
                    XMLHttpRequest();";
    client.println("xhr.open('GET', '/?mode=" +
                  value + "&", true);"
    client.println("xhr.send(); setInterval(function(){
                    location.reload(true); }, 1500); } ");
    client.println("function setTimer(value) { var xhr = new
                    XMLHttpRequest();";
    client.println("xhr.open('GET', '/?timer=" + value
                  + "&", true);"
    client.println("xhr.send(); setInterval(function(){
                    location.reload(true); }, 1500); } ");
    client.println("function setThreshold(value) { var xhr = new
                    XMLHttpRequest();";
    client.println("xhr.open('GET', '/?ldrthreshold=" + value
                  + "&", true);"
    client.println("xhr.send(); setInterval(function(){
                    location.reload(true); }, 1500); } ");
    client.println("function outputOn() { var xhr = new
                    XMLHttpRequest();";
    client.println("xhr.open('GET', '/?state=on", true);"
    client.println("xhr.send(); setInterval(function(){
                    location.reload(true); }, 1500); } ");
    client.println("function outputOff() { var xhr = new
                    XMLHttpRequest();";
    client.println("xhr.open('GET', '/?state=off", true);"
    client.println("xhr.send(); setInterval(function(){
                    location.reload(true); }, 1500); } ");
    client.println("function updateSensorReadings() { var xhr = new
                    XMLHttpRequest();";
    client.println("xhr.open('GET', '/?sensor", true);"
    client.println("xhr.send(); setInterval(function(){
                    location.reload(true); }, 1500); }</script></body></html>");

    // The HTTP response ends with another blank line
    client.println();
    // Break out of the while loop
    break;
} else { // if you got a newline, then clear currentLine
    currentLine = "";
}
} else if (c != '\r') { // if you got anything else but a carriage return character,
    currentLine += c;      // add it to the end of the currentLine
}
}

// Clear the header variable
header = "";
// Close the connection
client.stop();
Serial.println("Client disconnected.");
}

// Starts a timer to turn on/off the output according to the time value or LDR reading
now = millis();
// Mode selected (1): Auto PIR
if(startTimer && armMotion && !armLdr) {

```

```

    if(outputState == "off") {
        outputOn();
    }
    else if((now - lastMeasure > (timer * 1000))) {
        outputOff();
        startTimer = false;
    }
}

// Mode selected (2): Auto LDR
// Read current LDR value and turn the output accordingly
if(armLdr && !armMotion) {
    int ldrValue = map(analogRead(ldr), 0, 4095, 0, 100);
    //Serial.println(ldrValue);
    if(ldrValue > ldrThreshold && outputState == "on") {
        outputOff();
    }
    else if(ldrValue < ldrThreshold && outputState == "off") {
        outputOn();
    }
    delay(100);
}

// Mode selected (3): Auto PIR and LDR
if(startTimer && armMotion && armLdr) {
    int ldrValue = map(analogRead(ldr), 0, 4095, 0, 100);
    //Serial.println(ldrValue);
    if(ldrValue > ldrThreshold) {
        outputOff();
        startTimer = false;
    }
    else if(ldrValue < ldrThreshold && outputState == "off") {
        outputOn();
    }
    else if(now - lastMeasure > (timer * 1000)) {
        outputOff();
        startTimer = false;
    }
}
}

// Checks if motion was detected and the sensors are armed. Then, starts a timer.
void detectsMovement() {
    if(armMotion || (armMotion && armLdr)) {
        Serial.println("MOTION DETECTED!!!");
        startTimer = true;
        lastMeasure = millis();
    }
}
void configureMode() {
    // Mode: Manual
    if(selectedMode == 0) {
        armMotion = 0;
        armLdr = 0;
        // RGB LED color: red
        digitalWrite(redRGB, LOW);
        digitalWrite(greenRGB, HIGH);
        digitalWrite(blueRGB, HIGH);
    }
    // Mode: Auto PIR
    else if(selectedMode == 1) {

```

```

    outputOff();
    armMotion = 1;
    armLdr = 0;
    // RGB LED color: green
    digitalWrite(redRGB, HIGH);
    digitalWrite(greenRGB, LOW);
    digitalWrite(blueRGB, HIGH);
}
// Mode: Auto LDR
else if(selectedMode == 2) {
    armMotion = 0;
    armLdr = 1;
    // RGB LED color: blue
    digitalWrite(redRGB, HIGH);
    digitalWrite(greenRGB, HIGH);
    digitalWrite(blueRGB, LOW);
}
// Mode: Auto PIR and LDR
else if(selectedMode == 3) {
    outputOff();
    armMotion = 1;
    armLdr = 1;
    // RGB LED color: purple
    digitalWrite(redRGB, LOW);
    digitalWrite(greenRGB, HIGH);
    digitalWrite(blueRGB, LOW);
}
}

// Change output pin to on or off
void outputOn() {
    Serial.println("GPIO on");
    outputState = "on";
    digitalWrite(output, LOW);
    preferences.putInt("outputState", 1);
}
void outputOff() {
    Serial.println("GPIO off");
    outputState = "off";
    digitalWrite(output, HIGH);
    preferences.putInt("outputState", 0);
}

```

You can insert your network credentials in the following variables and the code will work straight away.

```

const char* ssid      = "REPLACE_WITH_YOUR_SSID";
const char* password = " REPLACE_WITH_YOUR_PASSWORD";

```

To learn how the code works, read the next Unit.

After inserting your network credentials, you can upload the code to your ESP32.

Make sure you have the right board and COM port selected.

Testing the Wi-Fi Multisensor Node

Open the Serial Monitor at a baud rate of 115200.

Press the ESP32 enable button to print the ESP32 IP address.

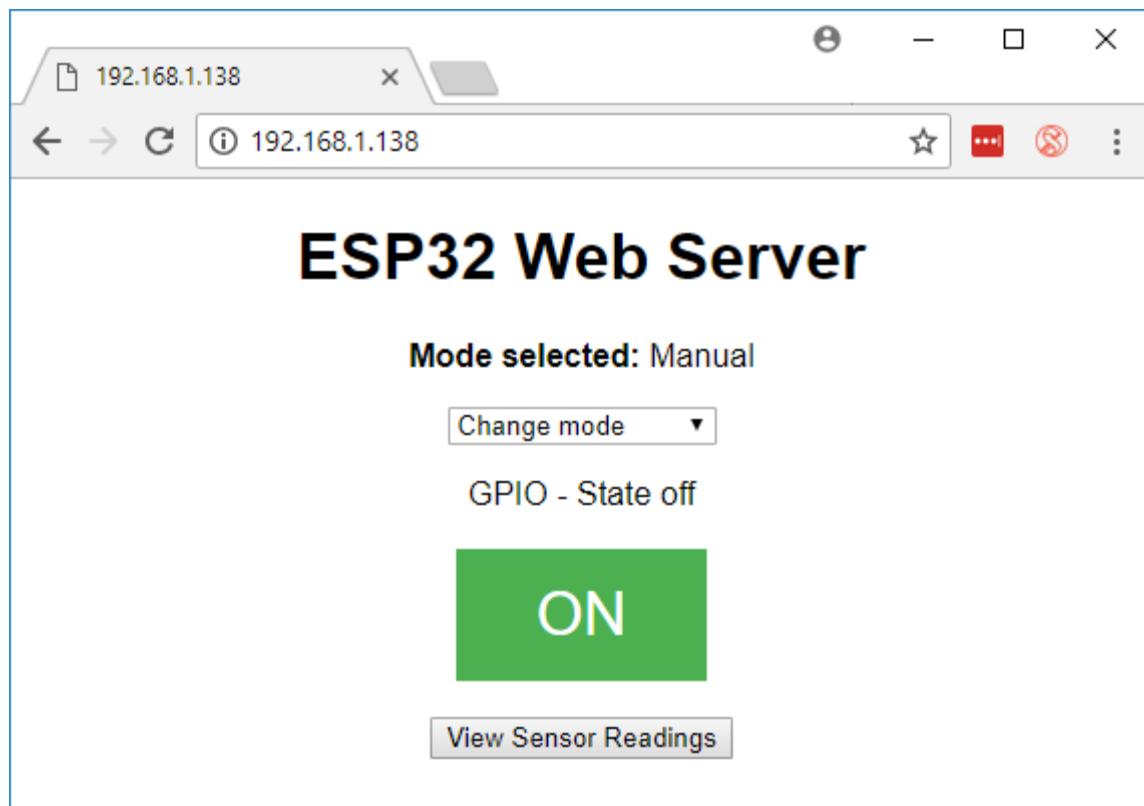


The screenshot shows the Arduino Serial Monitor window titled "Serial Monitor". The text area displays the following log:

```
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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
start...
Connecting to MEO-D32A40
.
WiFi connected.
IP address:
192.168.1.138
```

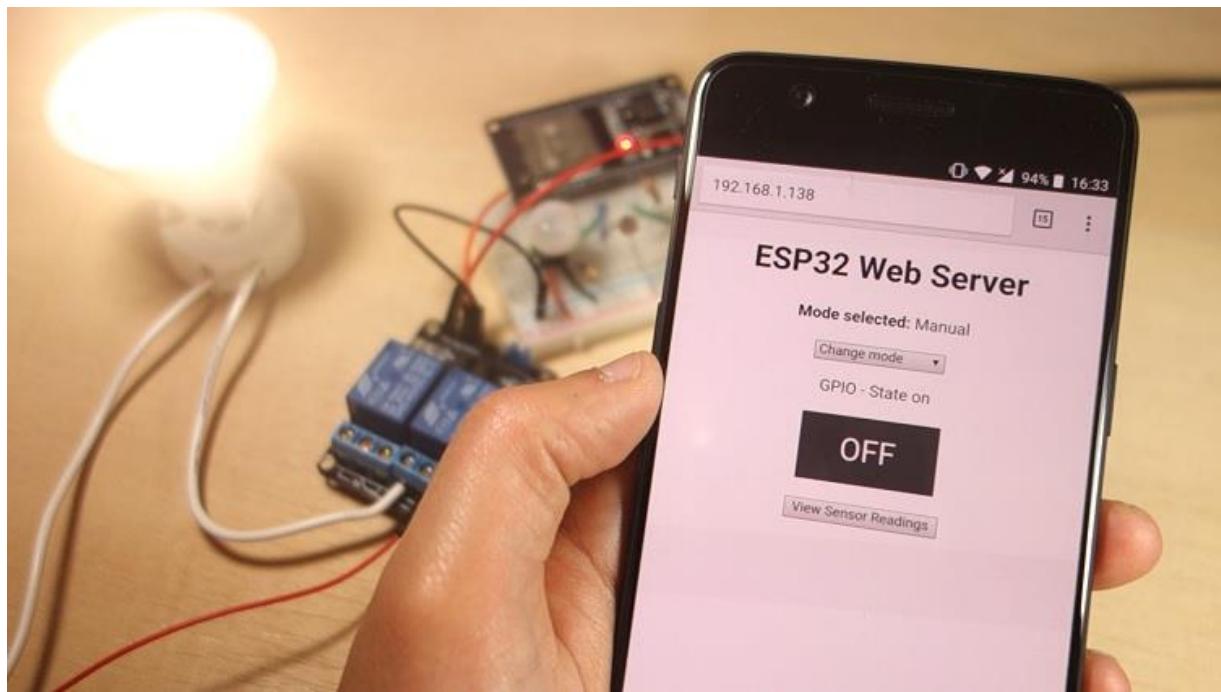
The status bar at the bottom indicates "Ln 419, Col 2 DOIT ESP32 DEVKIT V1 on COM9".

Open your browser and type the ESP32 IP address. The following page should load.

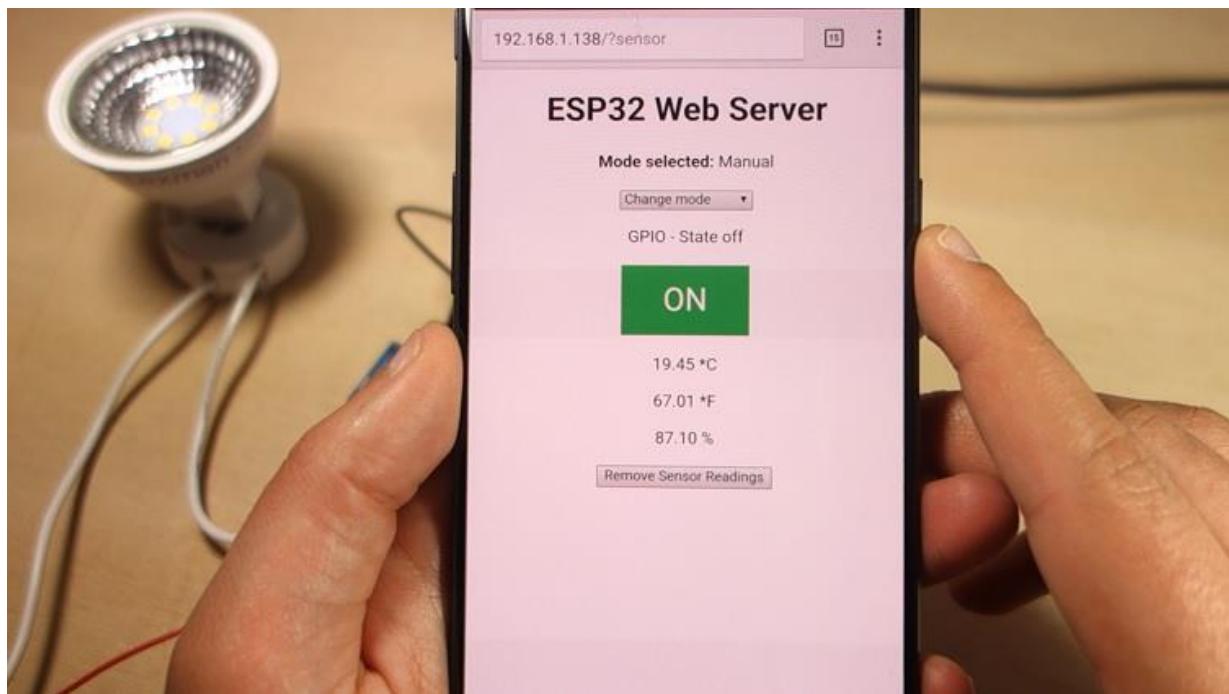


Now, select each mode, try to set different settings to check if everything is working properly.

For example, select Manual mode and turn the lamp on and off.



Click the "View Sensor Readings" button to request the latest sensor readings.



Select other modes and see the RGB LED color changing. Configure the multisensor with your own settings to check how it works.

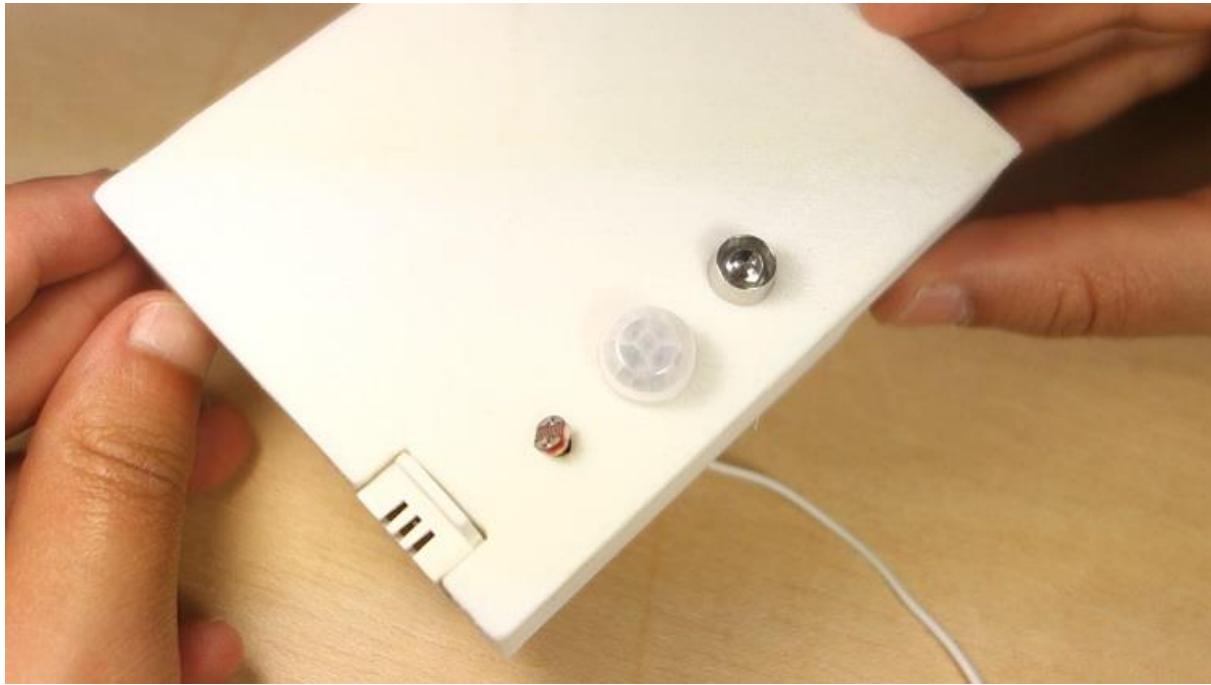
Building an Enclosure

After making sure everything is working properly, you can build a more permanent solution. We've 3D printed an enclosure using the Creality CR-10 3D printer to accommodate the circuit (read our [CR-10 3D Printer review](#)).

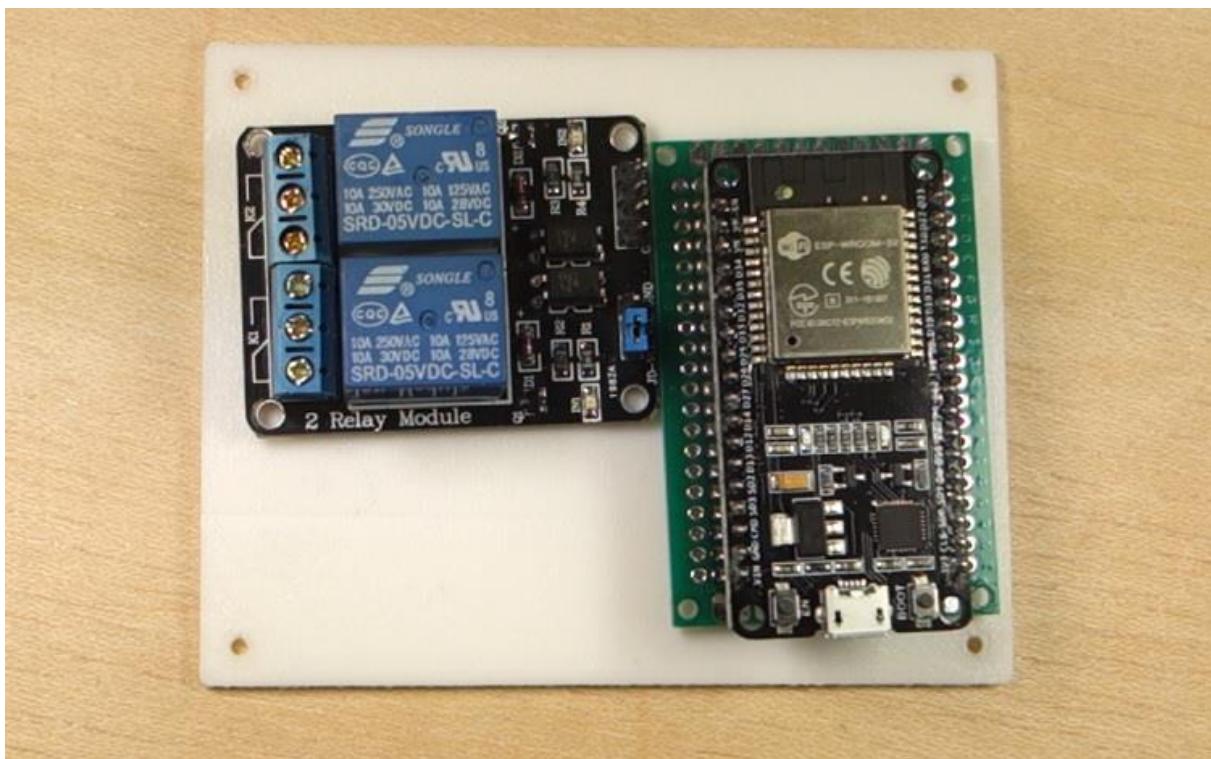
The enclosure consists of two pieces: the bottom and the lid. The lid has three holes on top.



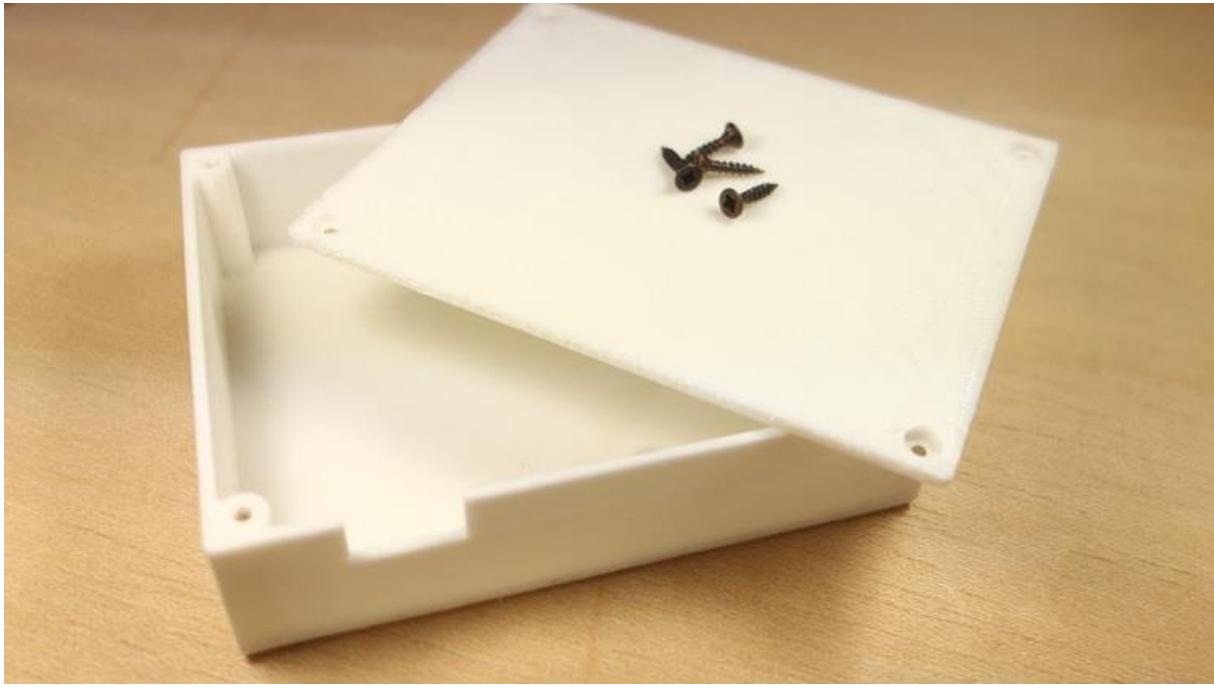
One to place the LDR, another to place the mini PIR motion sensor and another for the RGB LED holder. There's a slot at the side to place the DHT. There's a space for the relay wires and another for the USB cable to go through and power the ESP32.



At the bottom, you should place the ESP32 and the relay. You can use some hot glue or tape to fix the components more securely.



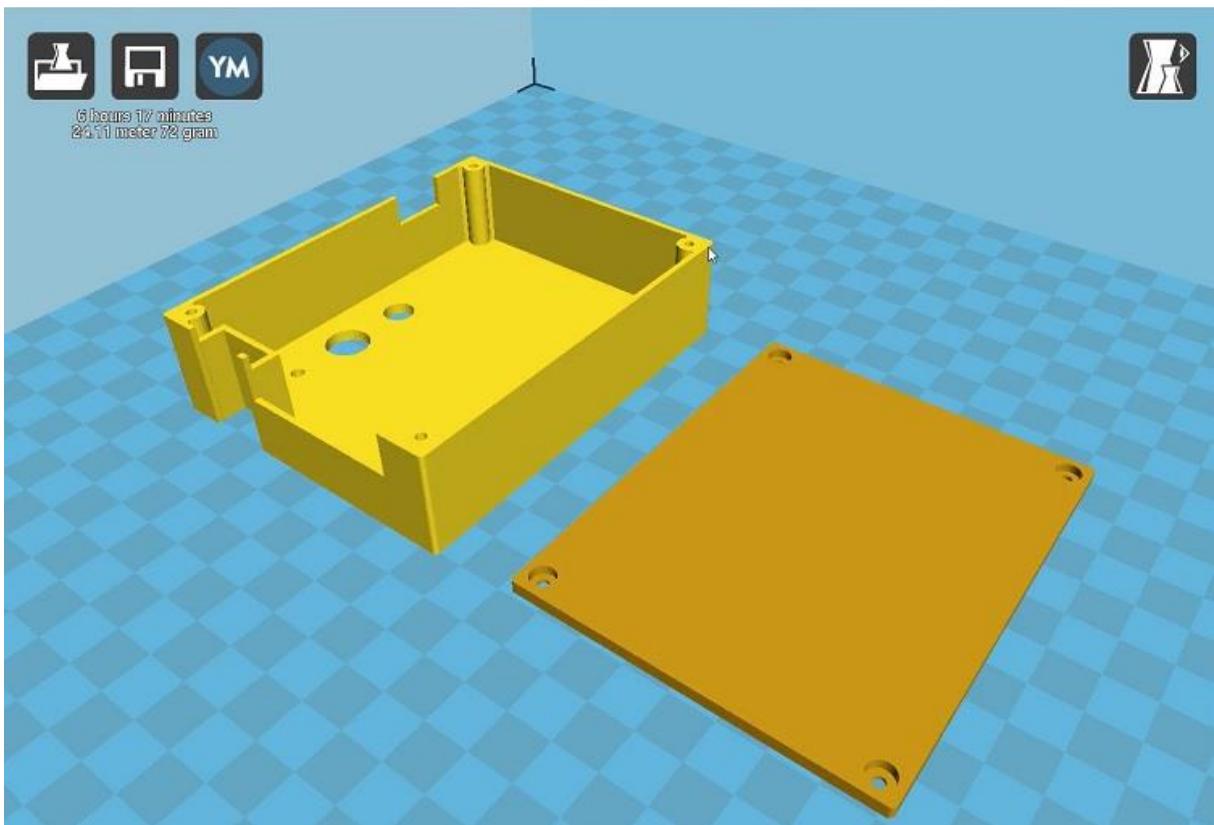
Finally, there are 4 holes to fix the lid with the bottom using 4 screws.



If you have a 3D printer, you can build an enclosure like this. Here are the required STL files (and SketchUp file):

- [Enclosure – 3D printer files](#)

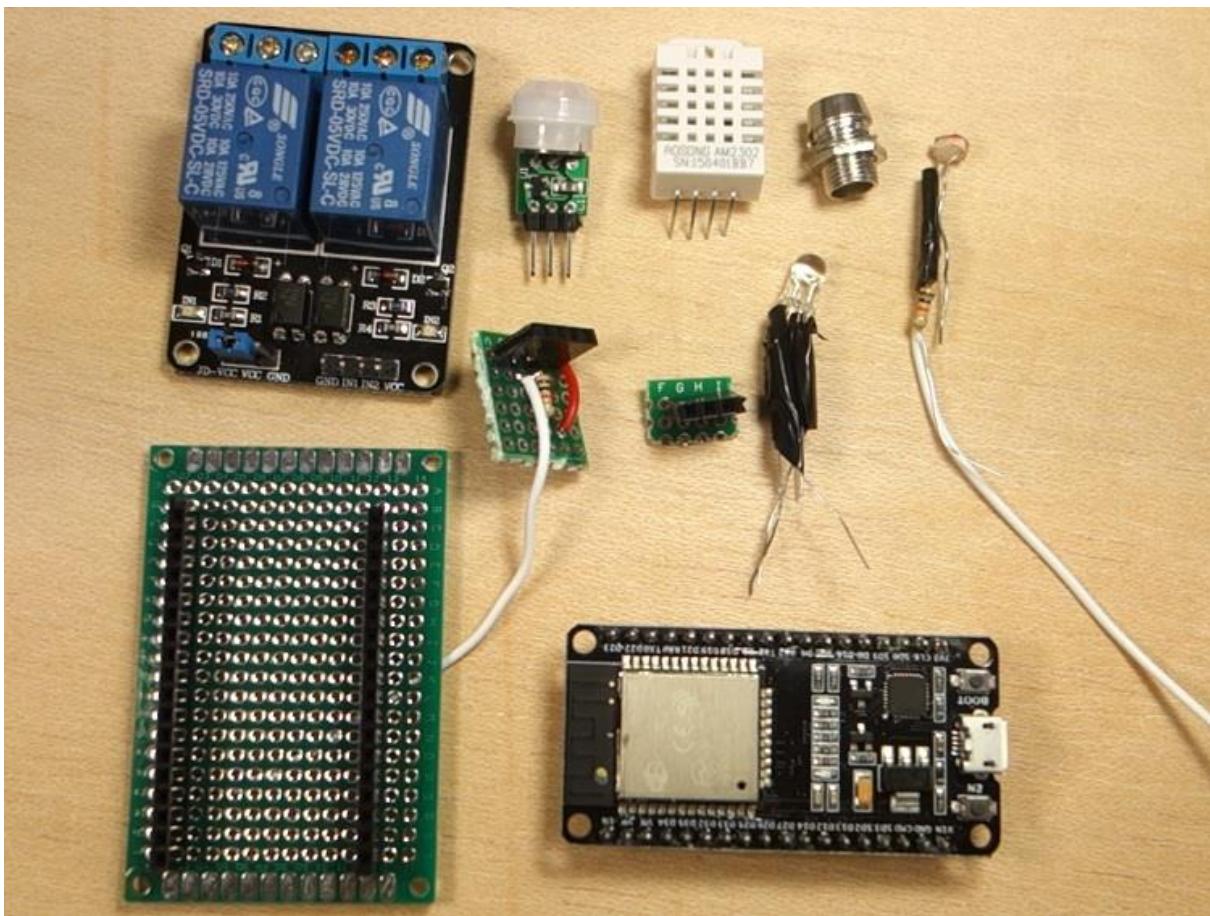
Note: we advise you to increase the size of the enclosure, as it will be easier to place all the components inside.



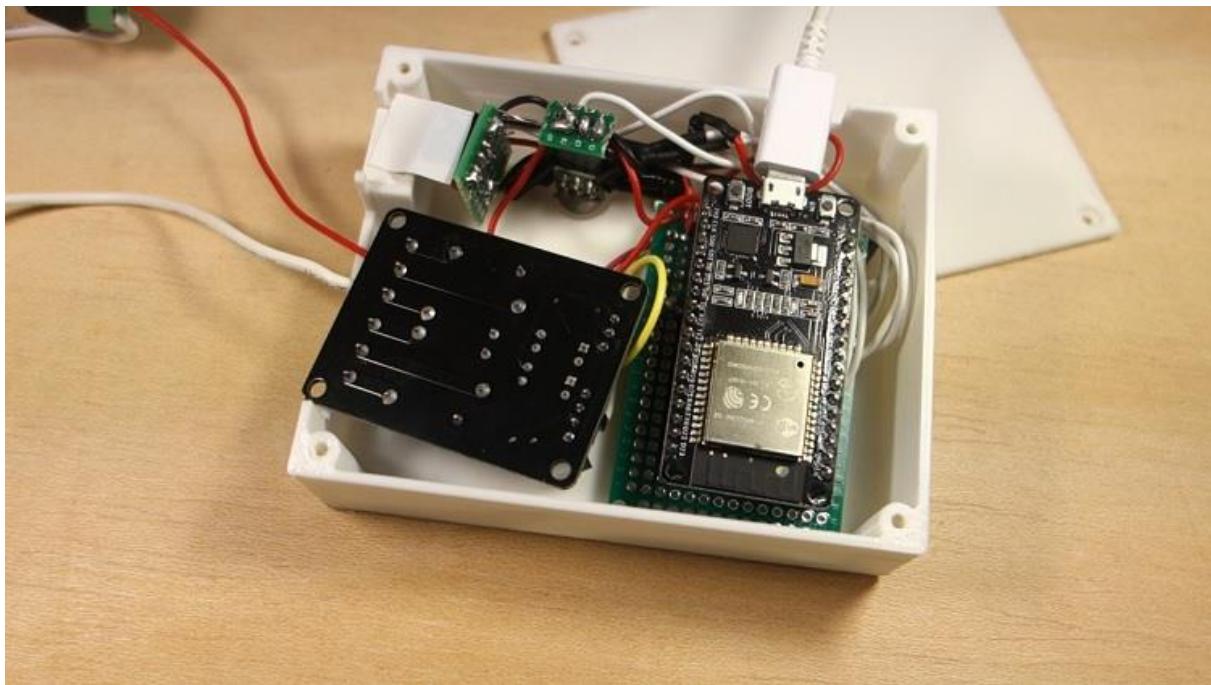
Alternatively, you can also use any other plastic enclosure and make some holes to place all the circuitry.



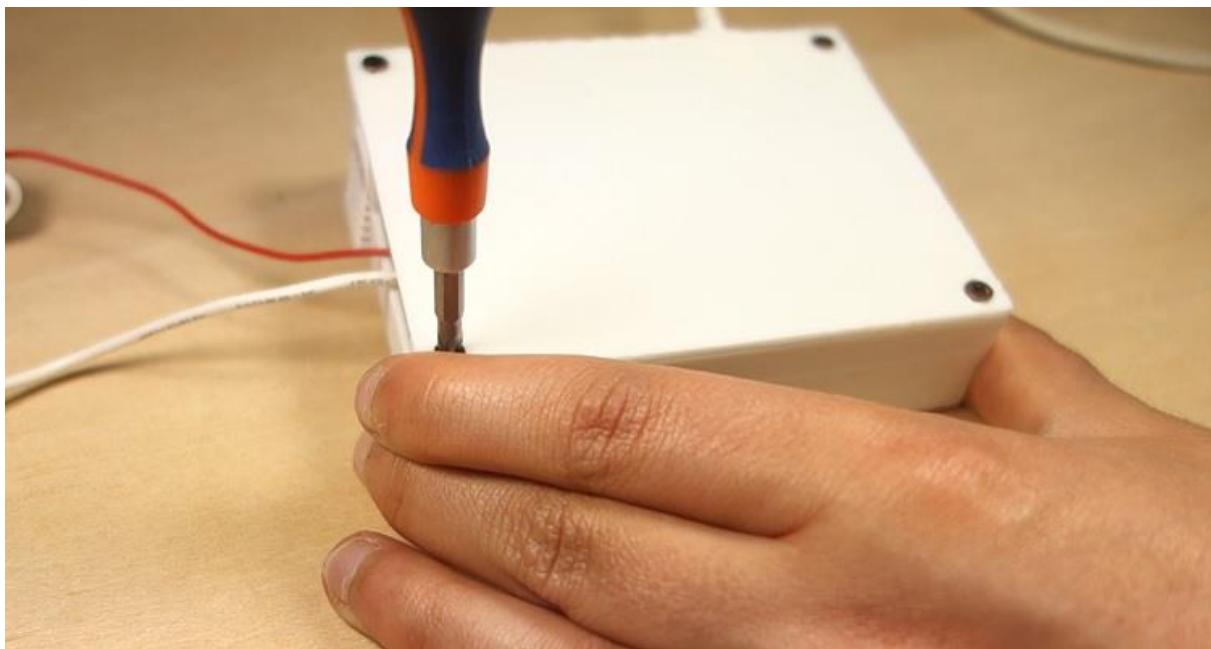
We've built a semi-permanent circuit using small pieces of prototyping board to place the ESP32, the PIR motion sensor, and the DHT. We've also soldered wires directly to the cheaper components like the LDR and the RGB LED.



The following figure shows how the circuit looks like inside the enclosure. If you don't like to solder, you can use the circuit on a breadboard and place it inside a different enclosure.

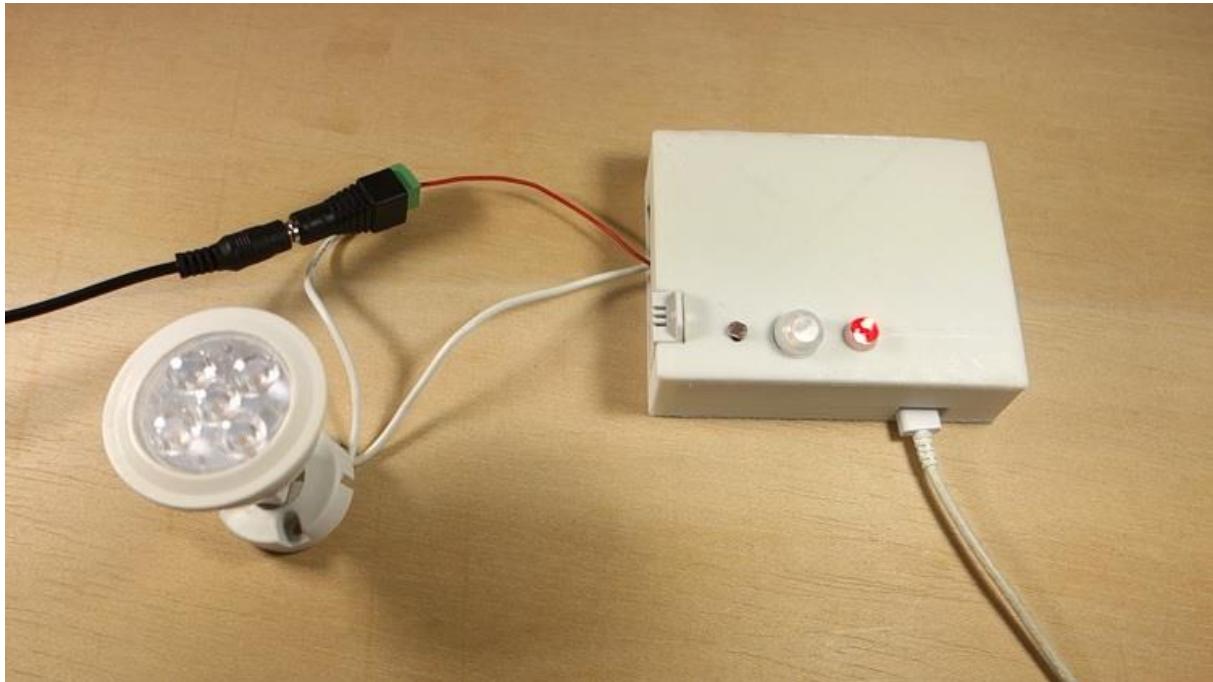


After powering the ESP32 using a USB cable, and connecting the relay to the lamp, we can close the enclosure using four screws.

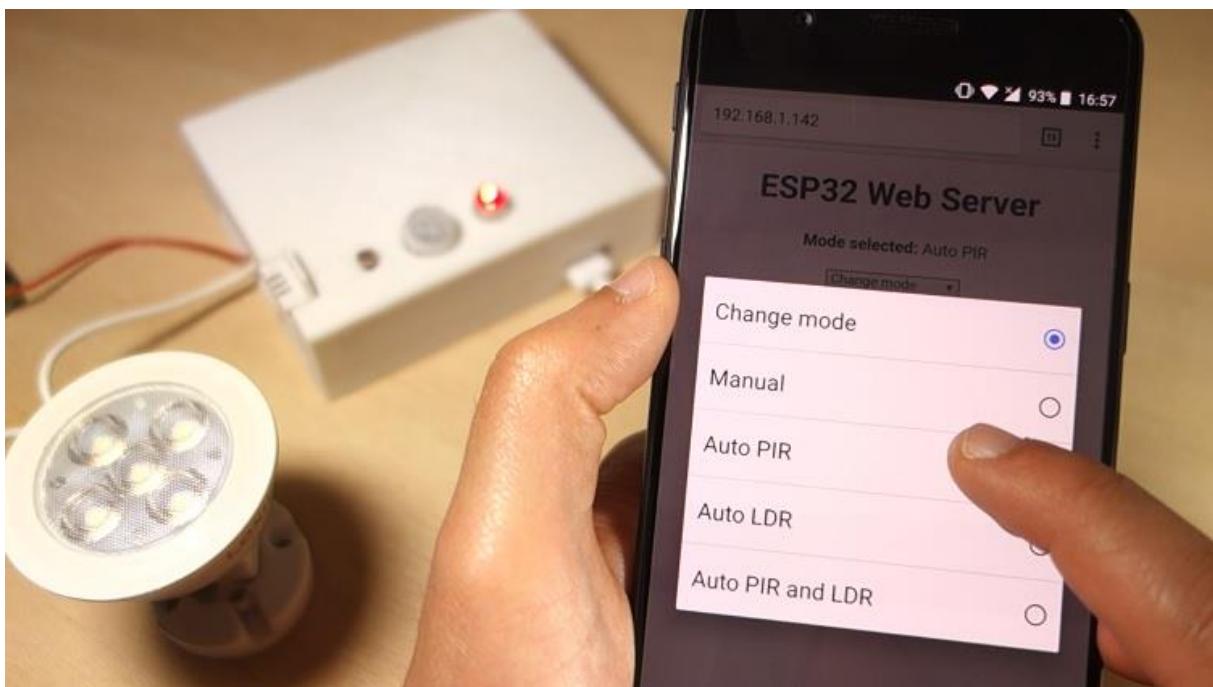


Demonstration

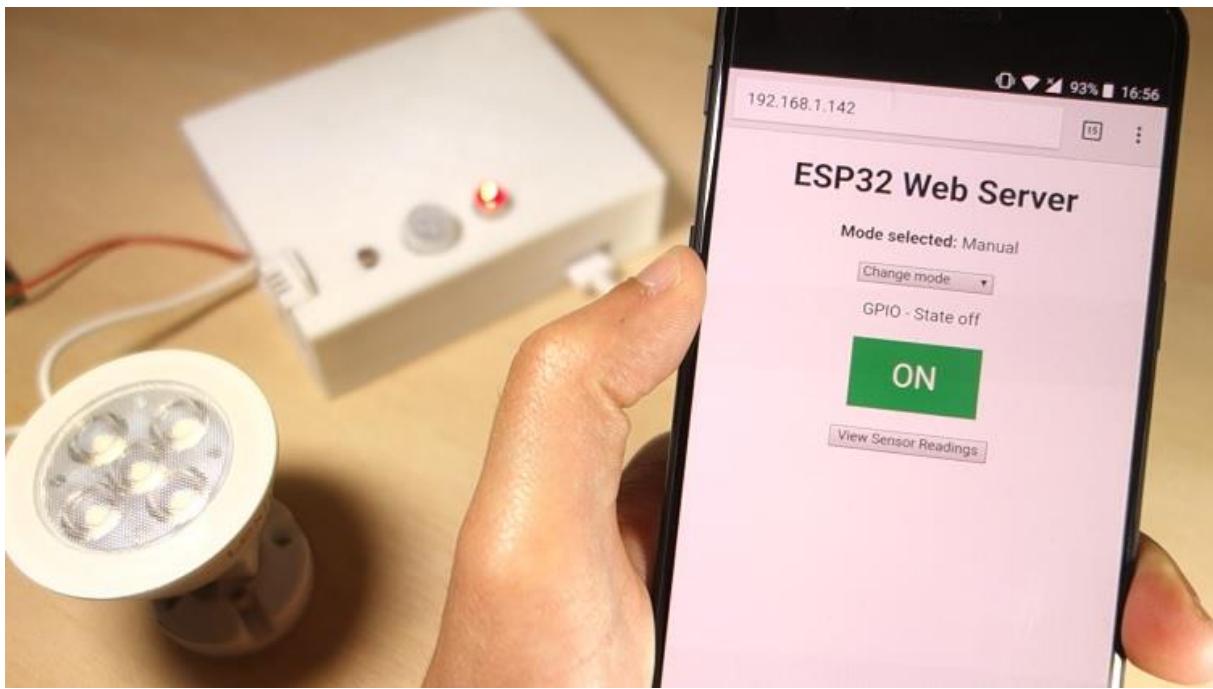
The following figure shows what the Wi-Fi Multisensor looks like.



Now, you can access your web server to control the output in different modes and configure the modes with your settings.



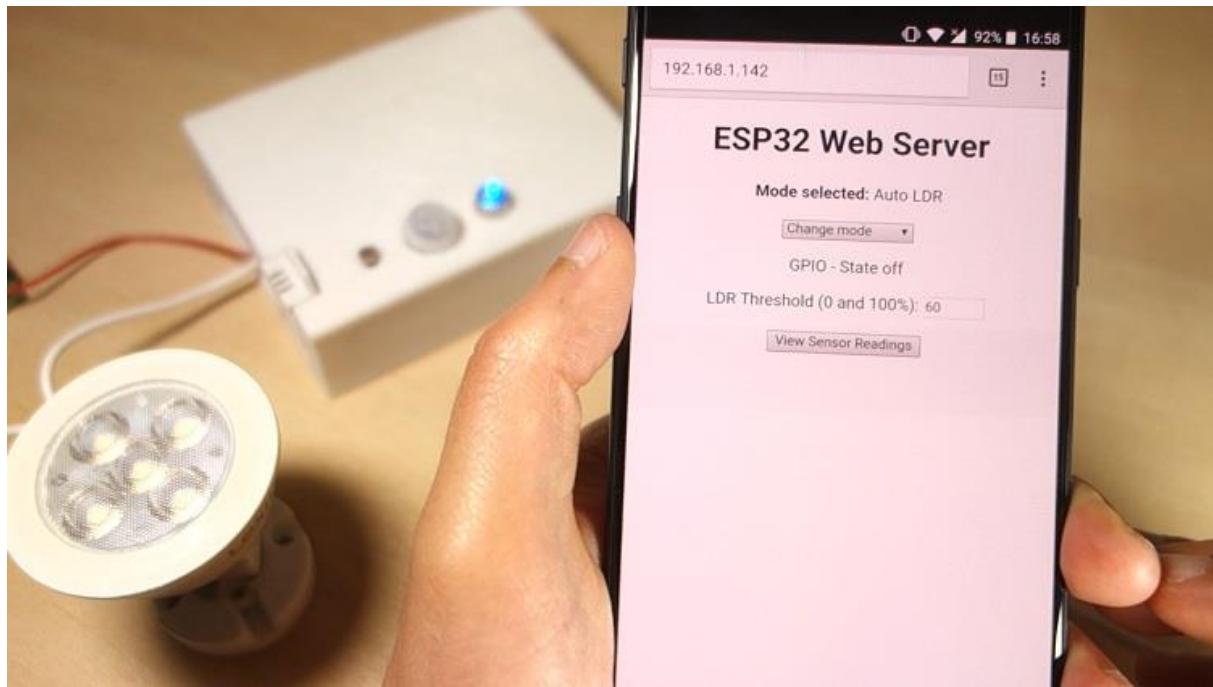
On manual mode, press the buttons to turn the lamp on and off.



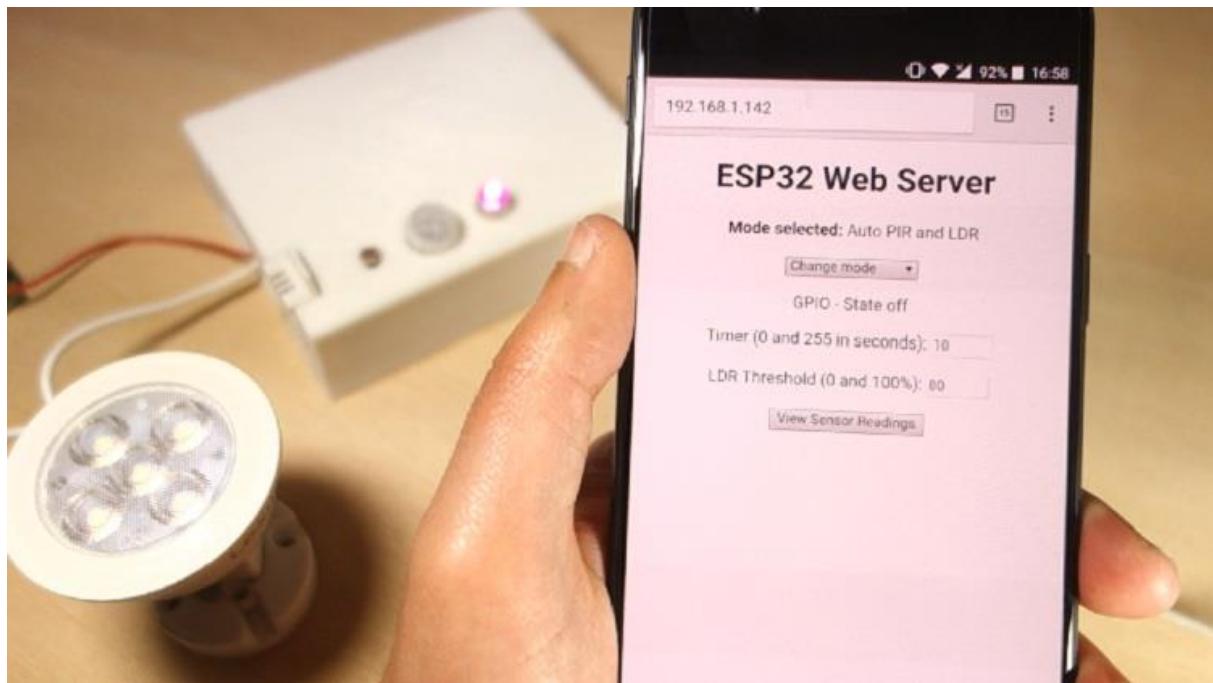
Select the Auto PIR mode. In this mode, the lamp turns on for the number of seconds you set, when motion is detected.



On the LDR mode, you can set the threshold value that will make the lamp light up. When I cover the LDR, the luminosity goes below the threshold, and the lamp lights up.

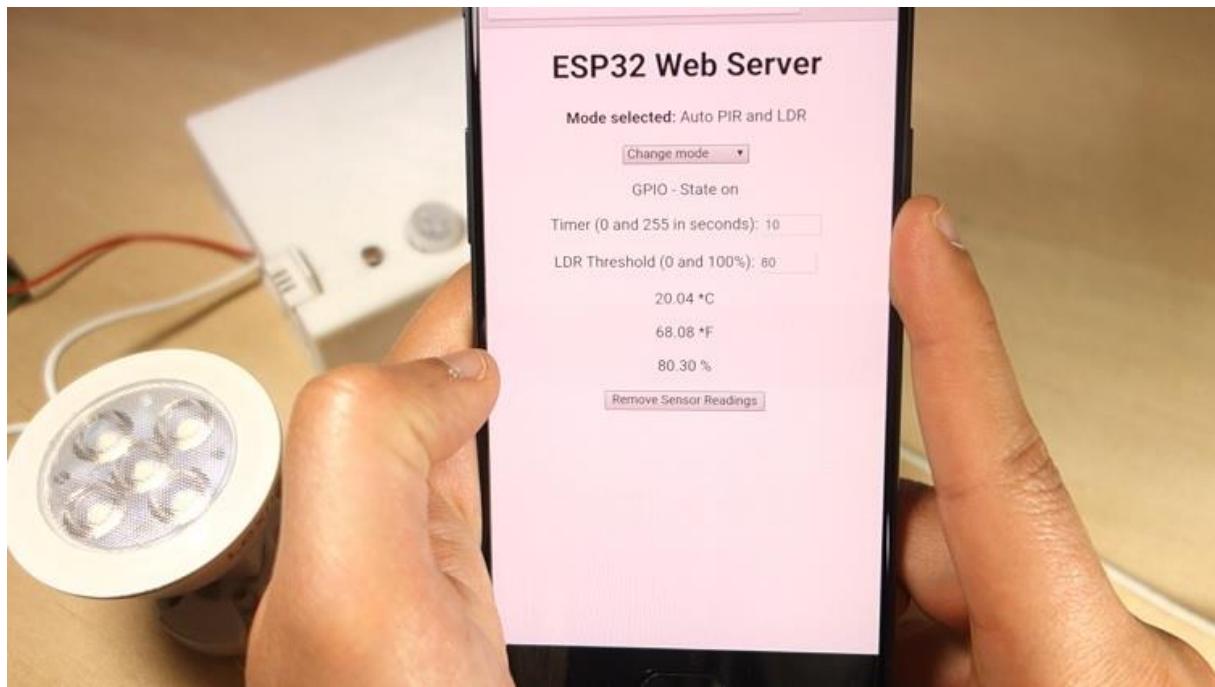


In Auto PIR and LDR mode, I can set the timer and the LDR threshold.



If motion is detected but the light intensity is above the threshold, nothing happens. But if I cover the LDR, which means there's no light, and motion is detected, the lamp turns on for the number of seconds I've defined in the settings.

You can also request the latest sensor readings every time you want.



So, that's it for the ESP32 Wi-Fi Multisensor project. Go to the next Unit to learn how the code works.

Unit 2 - ESP32 Wi-Fi Multisensor: How the Code Works?

In this Unit, we'll take a look at how the Wi-Fi Multisensor code works.

How the Code Works

This is the code used in the ESP32 Wi-Fi Multisensor project.

- [Click here to download the code.](#)

Including Libraries

You start by including the necessary libraries.

```
#include <WiFi.h>
#include <Preferences.h>
#include "DHT.h"
#include <Adafruit_Sensor.h>
```

Setting your Network Credentials

You need to add your network credentials in these next two variables.

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

DHT Sensor

In the next part of the code, you select the DHT sensor type you're using. In this case, we're using the DHT22.

```
// Uncomment one of the lines below for whatever DHT sensor type you're using!
//#define DHTTYPE DHT11    // DHT 11
//#define DHTTYPE DHT21    // DHT 21 (AM2301)
#define DHTTYPE DHT22    // DHT 22 (AM2302), AM2321
```

Then, you define the pin the DHT is connected to, and initialize an instance for the DHT sensor.

```
// DHT Sensor
const int DHTPin = 27;
// Initialize DHT sensor.
DHT dht(DHTPin, DHTTYPE);
```

The following variables are auxiliary variables to store the temperature in Celsius/Fahrenheit and the humidity.

```
static char celsiusTemp[7];
static char fahrenheitTemp[7];
static char humidityTemp[7];
```

Preferences

Next, create a preferences object so that later we can store variables permanently on the flash memory. We'll save: the last output state, the selected mode, the timer value, and the LDR threshold value.

```
Preferences preferences;
```

Defining GPIOs

In this section, we define the GPIOs for the output, RGB LED, PIR motion sensor, and the LDR.

```
const int output = 2;
const int redRGB = 14;
const int greenRGB = 12;
const int blueRGB = 13;
const int motionSensor = 25;
const int ldr = 33;
```

We also create a String variable to hold the `outputState` to be displayed on the web server.

```
String outputState = "off";
```

Timers

Next, we create auxiliary variables for the timers:

```
long now = millis();
long lastMeasure = 0;
boolean startTimer = false;
```

Selected Mode and Settings

Here, we initialize variables to store the selected mode and settings:

```
int selectedMode = 0;
```

```
int timer = 0;
int ldrThreshold = 0;
int armMotion = 0;
int armLdr = 0;
String modes[4] = { "Manual", "Auto PIR", "Auto LDR", "Auto PIR and LDR" };
```

Setting Variables for the Web Server

The following snippet of code is related to the web server. If you've followed previous Units, you should be familiar with it.

```
String valueString = "0";
int pos1 = 0;
int pos2 = 0;
// Variable to store the HTTP request
String header;
// Set web server port number to 80
WiFiServer server(80);
```

setup()

In the `setup()`, start by initializing the DHT sensor.

```
dht.begin();
```

Initialize the Serial Monitor at a baud rate of 115200 for debugging purposes.

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

Interrupt

Set the PIR motion sensor as an `INPUT_PULLUP`, and define it as an interrupt in `RISING` mode.

```
pinMode(motionSensor, INPUT_PULLUP);
attachInterrupt(digitalPinToInterrupt(motionSensor), detectsMovement, RISING);
```

Initialize Preferences

We initialize a namespace in `preferences` called `settings` where we'll save our data permanently.

```
// Initialize Preferences
preferences.begin("settings", false);
```

RGB LED

Here, you set the output (the relay) and RGB LED pins as outputs:

```
pinMode(output, OUTPUT);
```

```
pinMode(redRGB, OUTPUT);
pinMode(greenRGB, OUTPUT);
pinMode(blueRGB, OUTPUT);
```

Read from preferences

Read the flash memory and set the output to the last state saved. The output state is saved with the key `outputState`. If that key doesn't exist yet, it will return `0`.

```
if(preferences.getInt("outputState", 0) == 0) {
```

We check if the state saved is `1` or `0` to update the `outputState` variable with “**on**” or “**off**”. This variable is used to display the state of the output in the web server.

```
if(preferences.getInt("outputState", 0) == 0) {
    outputState = "off";
    digitalWrite(output, HIGH);
}
else {
    outputState = "on";
    digitalWrite(output, LOW);
}
```

We also update all variables that hold settings with the values saved in the flash memory, like the selected mode, timer, and LDR threshold.

```
selectedMode = preferences.getInt("selectedMode", 0);
timer = preferences.getInt("timer", 0);
ldrThreshold = preferences.getInt("ldrThreshold", 0);
```

Then, we call the `configureMode()` function to assign the right values for each mode.

```
configureMode();
```

configureMode()

Let's take a look at how this function works.

If the selected mode is Manual, the motion is not activated (`armMotion`), neither the LDR (`armLdr`). We also set the RGB LED to color red.

```
// Mode: Manual
if(selectedMode == 0) {
    armMotion = 0;
    armLdr = 0;
    // RGB LED color: red
    digitalWrite(redRGB, LOW);
    digitalWrite(greenRGB, HIGH);
    digitalWrite(blueRGB, HIGH);
```

```
}
```

A similar process is done to configure the other modes. You change the arm variables to activate or deactivate a sensor and set the RGB LED to a different color to indicate the selected mode. Now, let's go back to the `setup()`.

Wi-Fi connection

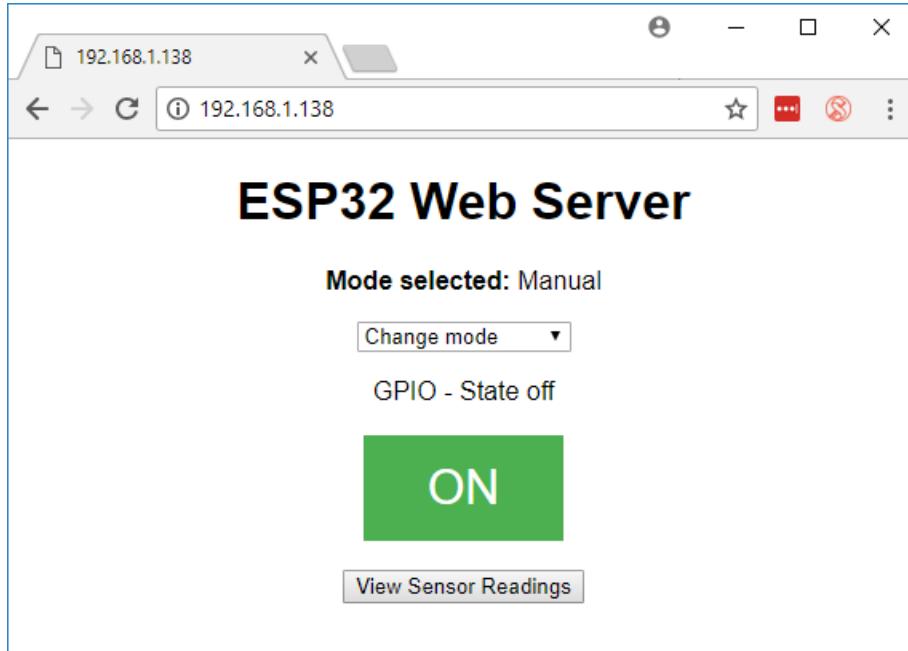
Here, we connect to the Wi-Fi network and print the ESP32 IP address in the Serial Monitor.

```
// 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 display the web server and make things happen accordingly to the selected mode and settings. We've covered web servers in great detail in previous units. So, we'll just take a look at the parts that are relevant for this project. This part of the code is easier to understand if we explain what's happening with a live demonstration.

When you access the web server, you'll see a similar web page as shown in the following figure.



At the top, you can select one of these four different modes.

- Manual
- Auto PIR
- Auto LDR
- Auto PIR and LDR

Manual mode

For example, if you choose Manual mode, the following part of the code is being executed.

```
if(header.indexOf("GET /?mode=") >= 0) {  
    pos1 = header.indexOf('=');  
    pos2 = header.indexOf('&');  
    valueString = header.substring(pos1+1, pos2);  
    selectedMode = valueString.toInt();  
    preferences.putInt("selectedMode", selectedMode);  
    configureMode();  
}
```

It saves the selected mode in the `selectedMode` variable and stores it in the flash memory with the key “`selectedMode`”.

```
preferences.putInt("selectedMode", selectedMode);
```

The web page look changes accordingly to the selected mode. In this case, since we've selected the manual mode that corresponds to `0`, the following if statement is true and the web page will display two buttons to control the output.

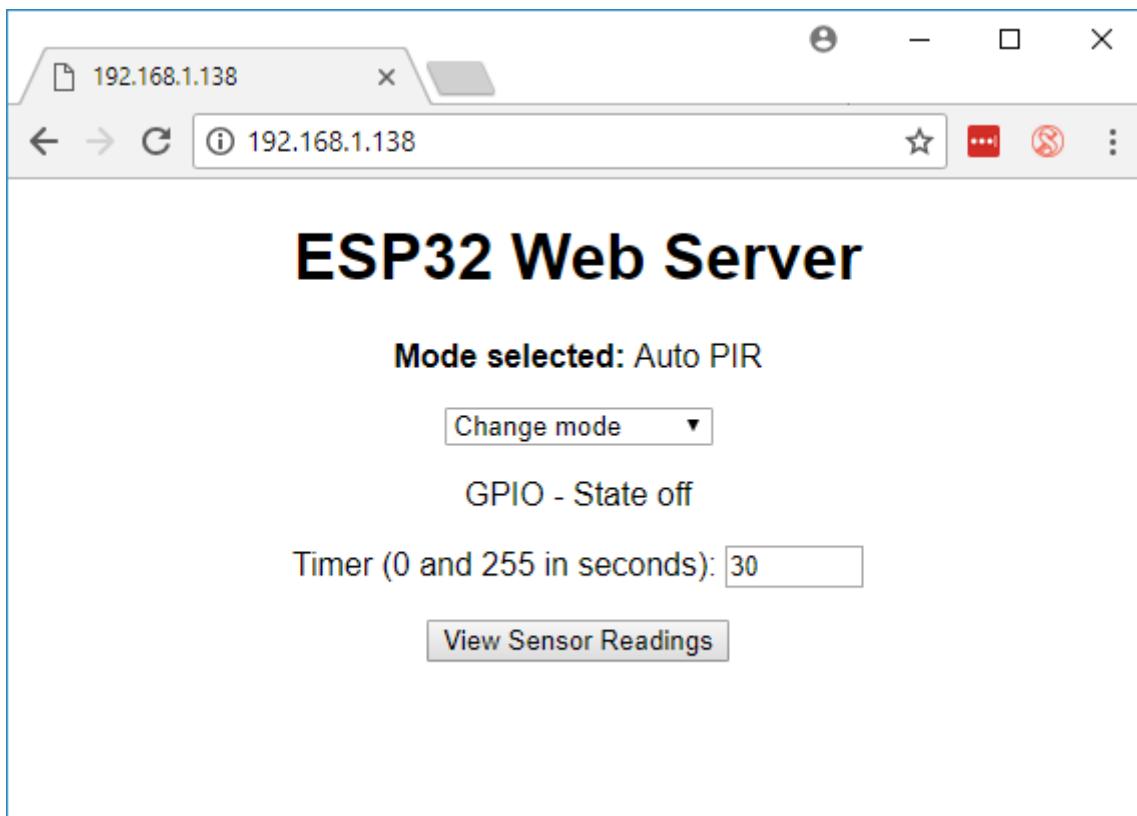
```
if(selectedMode == 0) {
    if(outputState == "off") {
        client.println("<p><button class=\"button\" onclick=\"outputOn()\">>ON</button></p>");
    }
    else {
        client.println("<p><button class=\"button button2\" onclick=\"outputOff()\">>OFF</button></p>");
    }
}
```

When you click the on and off buttons, in the background, the following code runs, and one of these two `else if` statements turn the output on or off.

```
// Change the output state - turn GPIOs on and off
else if(header.indexOf("GET /?state=on") >= 0) {
    outputOn();
}
else if(header.indexOf("GET /?state=off") >= 0) {
    outputOff();
}
```

Auto PIR mode

Now, in the drop-down menu select the Auto PIR mode.



There's a new input field that shows up in the web page. This field allows you to type an int number from 0 to 255 to specify the number of seconds the output should remain on after motion is detected.

When you change the number, it calls the following part of the code and changes the `timer` variable.

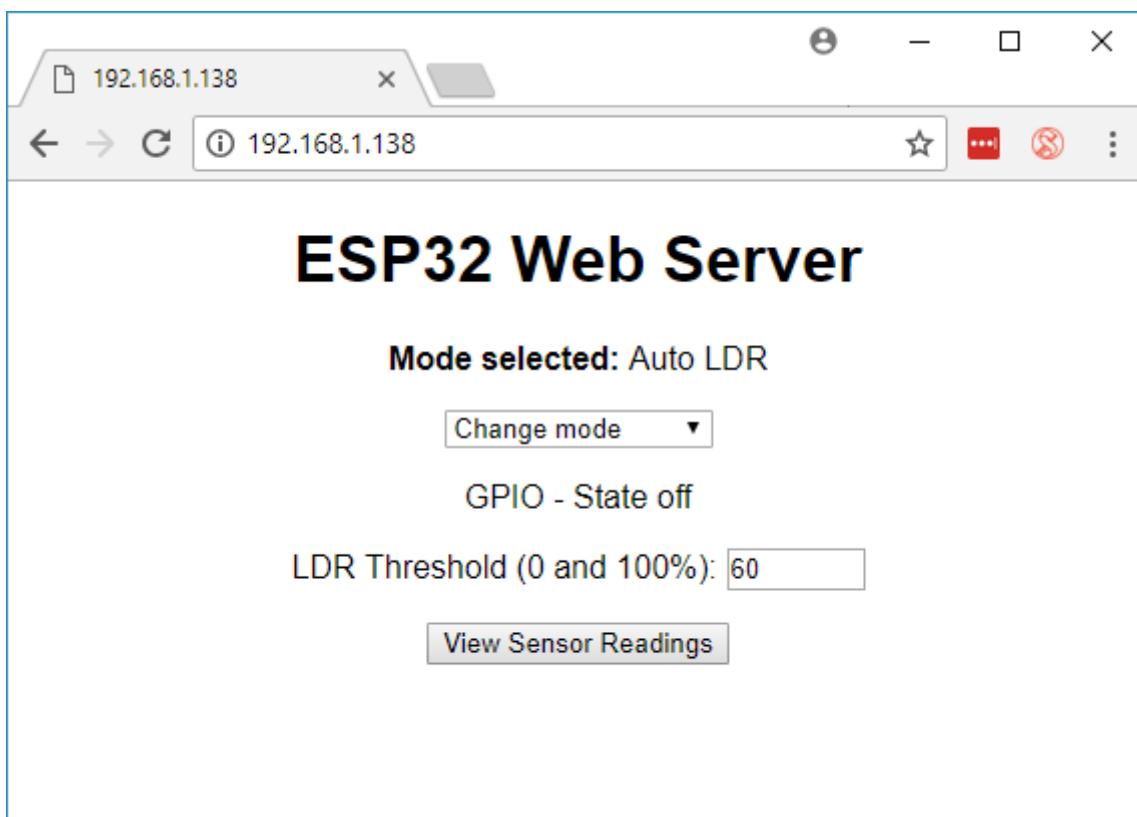
```
// Set timer value
else if(header.indexOf("GET /?timer=") >= 0) {
    pos1 = header.indexOf('=');
    pos2 = header.indexOf('&');
    valueString = header.substring(pos1+1, pos2);
    timer = valueString.toInt();
    preferences.putInt("timer", timer);
    Serial.println(valueString);
}
```

In this mode (mode 1), it only displays the input field for the timer.

```
else if(selectedMode == 1) {
    client.println("<p>Timer (0 and 255 in seconds):
        <input type=\"number\" name=\"txt\" value=\"" +
        String(preferences.getInt("timer", 0)) + "\"
        onchange=\"setTimer(this.value)\" min=\"0\" max=\"255\"></p>");
}
```

Auto LDR mode

Select Auto LDR mode and a new input field appears.



This sets the LDR threshold value and you can write down a number between 0 and 100 to indicate the % of luminosity. When you change this field, it calls the following part of the code to update the LDR threshold value:

```

// Set LDR Threshold value
else if(header.indexOf("GET /?ldrthreshold=") >= 0) {
    pos1 = header.indexOf('=');
    pos2 = header.indexOf('&');
    valueString = header.substring(pos1+1, pos2);
    ldrThreshold = valueString.toInt();
    preferences.putInt("ldrThreshold", ldrThreshold);
    Serial.println(valueString);
}

```

This is mode 2, and it will display the ldr threshold input field.

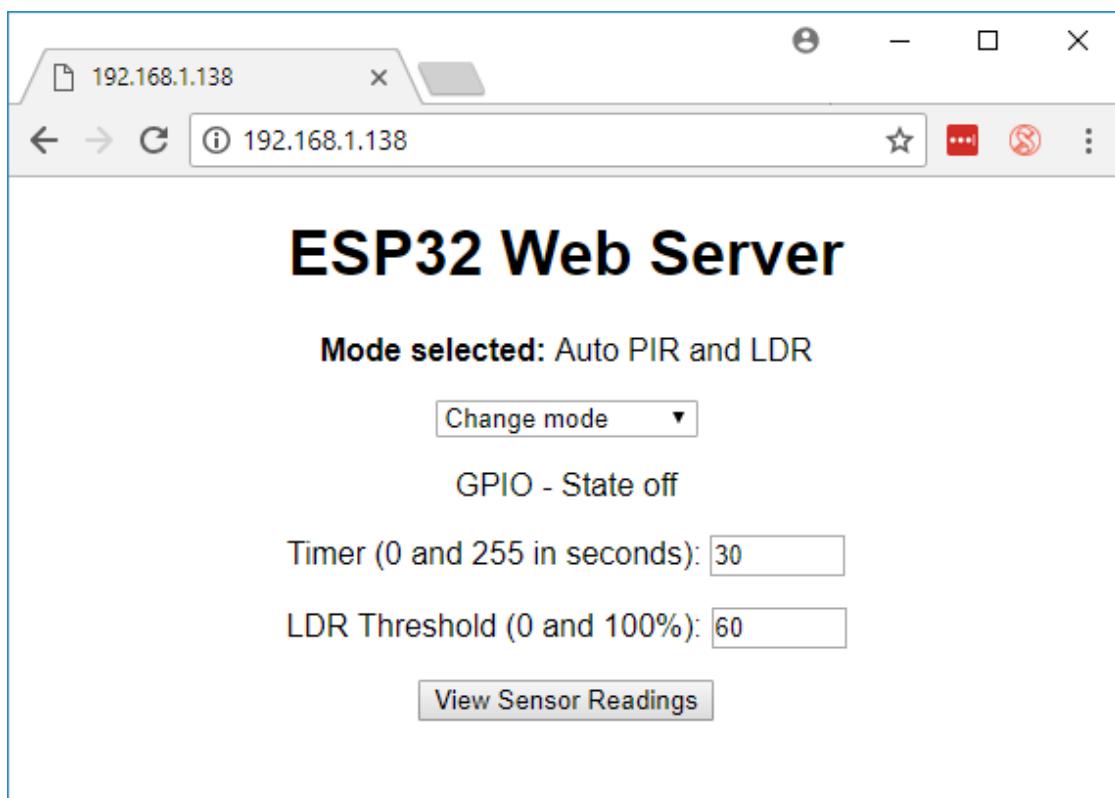
```

else if(selectedMode == 2) {
    client.println("<p>LDR Threshold (0 and 100%):");
    client.println("  <input type=\"number\" name=\"txt\" value=\"" +
        String(preferences.getInt("ldrThreshold", 0)) +
        "\" onchange=\"setThreshold(this.value)\" min=\"0\" max=\"100\"></p>");
}

```

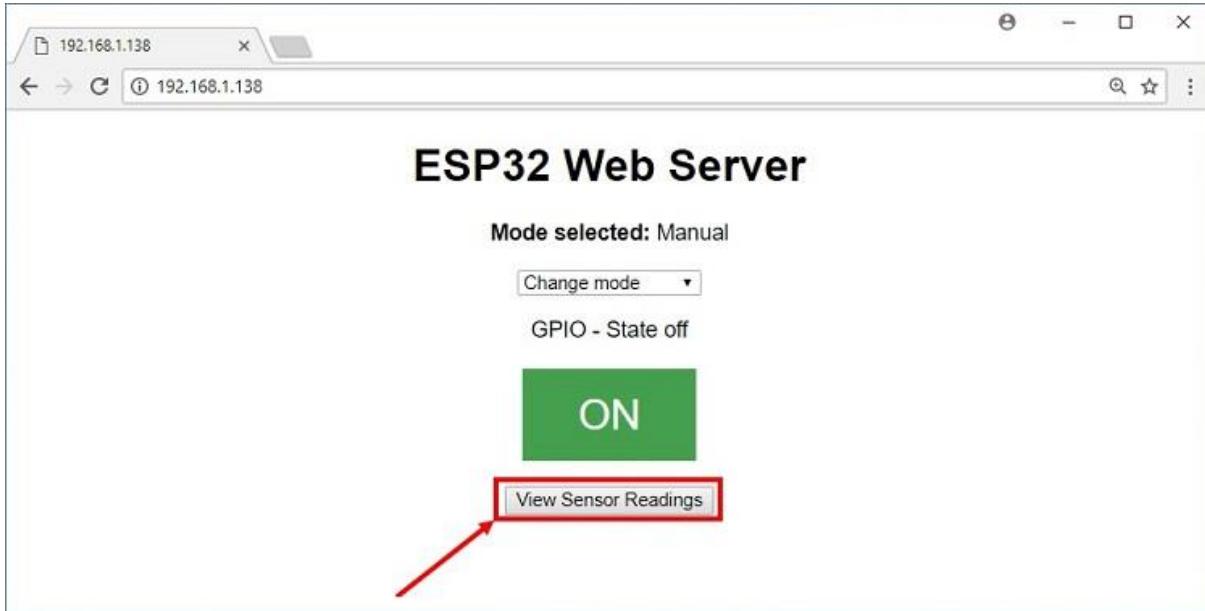
Auto PIR and LDR mode

Selecting the Auto PIR and LDR mode activates both the PIR and LDR. It also loads a new web page with two input fields.



Both input fields work the same way, as we've described earlier.

Sensor readings



Lastly, there's a button to request and display sensor readings.

```
// Get and display DHT sensor readings
if(header.indexOf("GET /?sensor") >= 0) {
    float h = dht.readHumidity();
    // Read temperature as Celsius (the default)
    float t = dht.readTemperature();
    // Read temperature as Fahrenheit (isFahrenheit = true)
    float f = dht.readTemperature(true);
    // Check if any reads failed and exit early (to try again).
    if (isnan(h) || isnan(t) || isnan(f)) {
        Serial.println("Failed to read from DHT sensor!");
        strcpy(celsiusTemp, "Failed");
        strcpy(fahrenheitTemp, "Failed");
        strcpy(humidityTemp, "Failed");
    }
    else {
        dtostrf(t, 6, 2, celsiusTemp);
        dtostrf(f, 6, 2, fahrenheitTemp);
        dtostrf(h, 6, 2, humidityTemp);
    }
    client.println("<p>");
    client.println(celsiusTemp);
    client.println("<*C</p><p>");
    client.println(fahrenheitTemp);
    client.println("<*F</p></div><p>");
    client.println(humidityTemp);
    client.println("%</p></div>");
}
```

There's also a button you can press to remove those readings.

```
client.println("<p><a href=\"/\\"><button>Remove Sensor Readings</button></a></p>");
```

That's how you configure the settings of your multisensor. Then, according to the mode and settings selected, another part of the `loop()` is running to check whether the output should be on or off.

Controlling the Output State

For example, when motion is detected, it calls the `detectsMovement()` function that starts a timer.

```
void detectsMovement() {
    if(armMotion || (armMotion && armLdr)) {
        Serial.println("MOTION DETECTED!!!");
        startTimer = true;
        lastMeasure = millis();
    }
}
```

Then, depending on the elapsed time, it turns the output on or off.

```
// Mode selected (1): Auto PIR
if(startTimer && armMotion && !armLdr) {
    if(outputState == "off") {
        outputOn();
    }
    else if((now - lastMeasure > (timer * 1000))) {
        outputOff();
        startTimer = false;
    }
}
```

There's also the following section of the code to turn the output on or off according to the luminosity of the threshold value.

```
// Mode selected (2): Auto LDR
// Read current LDR value and turn the output accordingly
if(armLdr && !armMotion) {
    int ldrValue = map(analogRead(ldr), 0, 4095, 0, 100);
    //Serial.println(ldrValue);
    if(ldrValue > ldrThreshold && outputState == "on") {
        outputOff();
    }
    else if(ldrValue < ldrThreshold && outputState == "off") {
        outputOn();
    }
    delay(100);
}
```

Finally, the following snippet of code runs when the auto PIR and LDR mode is selected and motion is detected.

```
// Mode selected (3): Auto PIR and LDR
```

```
if(startTimer && armMotion && armLdr) {  
    int ldrValue = map(analogRead(ldr), 0, 4095, 0, 100);  
    //Serial.println(ldrValue);  
    if(ldrValue > ldrThreshold) {  
        outputOff();  
        startTimer = false;  
    }  
    else if(ldrValue < ldrThreshold && outputState == "off") {  
        outputOn();  
    }  
    else if(now - lastMeasure > (timer * 1000)) {  
        outputOff();  
        startTimer = false;  
    }  
}
```

That's pretty much how the code works. We made an effort to write many comments in the code to make it easier to understand.

PROJECT 2

**Remote Controlled Wi-Fi
Car Robot**

Unit 1 - Remote Controlled Wi-Fi Car Robot

(Part 1/2)

In this project, we'll show you how to build an ESP32 Wi-Fi remote-controlled car robot step by step.



This project is quite long, so it is divided in two parts.

- **Part 1:** First, we'll take a look at how to control DC motors with the ESP32 and assemble the circuit;
- **Part 2:** Then, we'll program the ESP32 with a Web Server to control the Robot.

This project applies several concepts addressed throughout the course. For a better understanding of this project, we recommend taking a look at the following units if you haven't already:

- [**ESP32 Pulse-Width Modulation \(PWM\): Unit 2.5**](#)
- [**ESP32 Web Server – Control Outputs: Unit 6.2**](#)

Parts required:

Here's a list of the parts required to build the ESP32 Wi-Fi remote-controlled car robot:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [Smart Robot Chassis Kit](#) (or your own DIY robot chassis + 2x [DC motors](#))
- [L298N motor driver](#)
- [1x Power bank – portable charger](#)
- [4x 1.5 AA batteries](#)
- [2x 100nF ceramic capacitors](#)
- [1x SPDT Slide Switch](#)
- [Jumper wires](#)
- [Breadboard](#) or [stripboard](#)
- [Velcro tape](#)

Project Overview

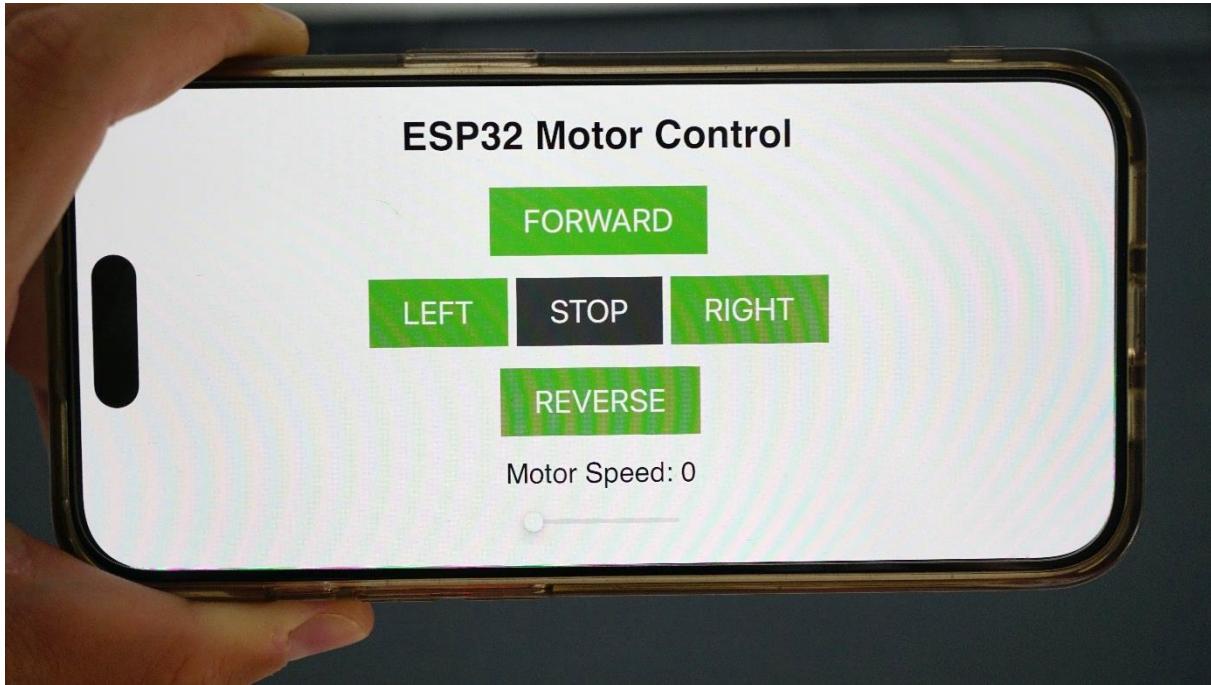
Before starting the project, we'll highlight the most important features and components used to make the robot work.

Wi-Fi

The robot will be controlled via Wi-Fi using your ESP32. We'll create a web-based interface to control the robot that can be accessed on any device inside your local network.

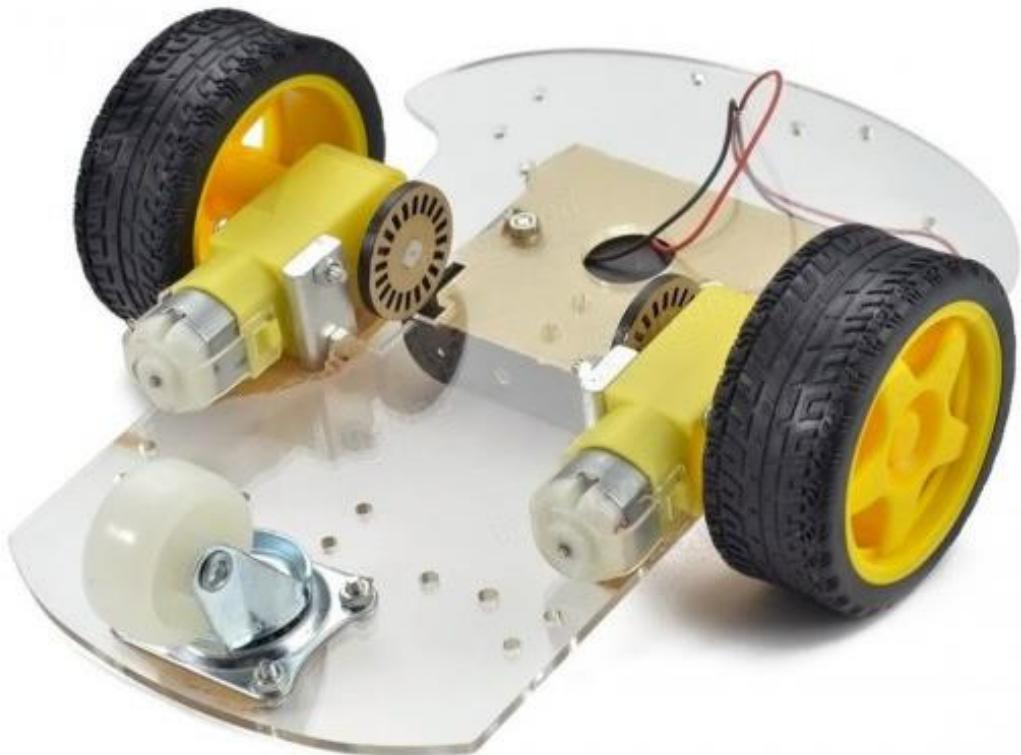
Robot Controls

The web server has five control options: forward, reverse, right, left, and stop. There's also a slider to control the speed.



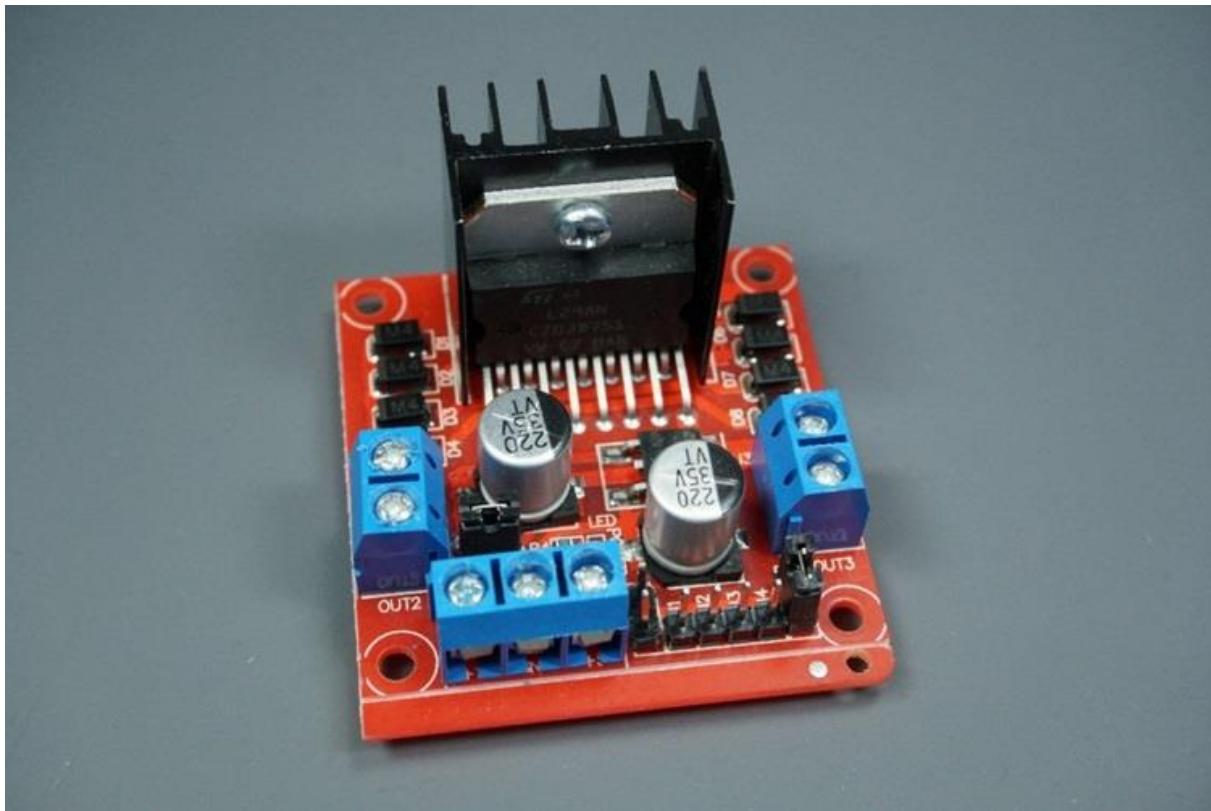
Smart Robot Chassis Kit

We're going to use the robot chassis kit shown in the figure below. That is the [Smart Robot Chassis Kit](#). You can find it in most online stores. The kit costs around \$10, and it's easy to assemble. You can use any other chassis kit as long as it comes with two DC motors.



L298N Motor Driver

There are many ways to control DC motors. We'll use the L298N motor driver that provides an easy way to control the speed and direction of 2 DC motors.

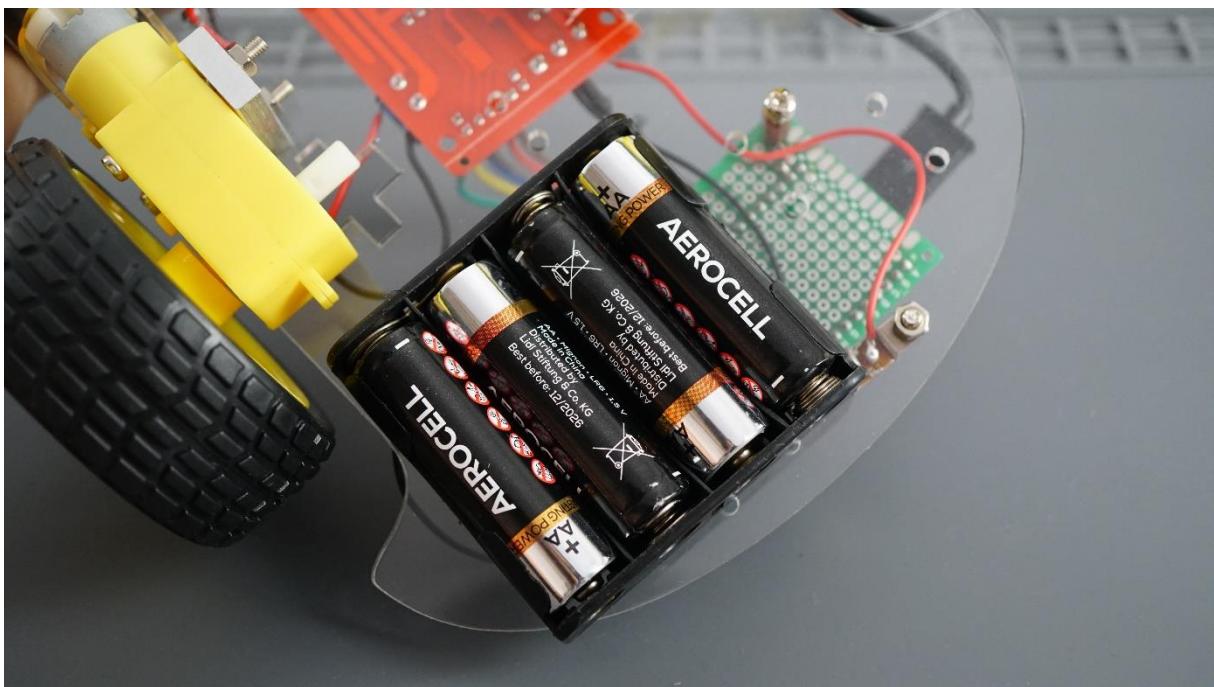


Power

The motors draw a lot of current, so you need to use an external power supply. This means you need two different power sources. One will power the DC motors, and the other will power the ESP32. We'll power the ESP32 using a powerbank (like the ones used to charge your smartphone).



The motors will be powered using 4 AA 1.5V batteries. You might consider using rechargeable batteries or any other suitable power supply.



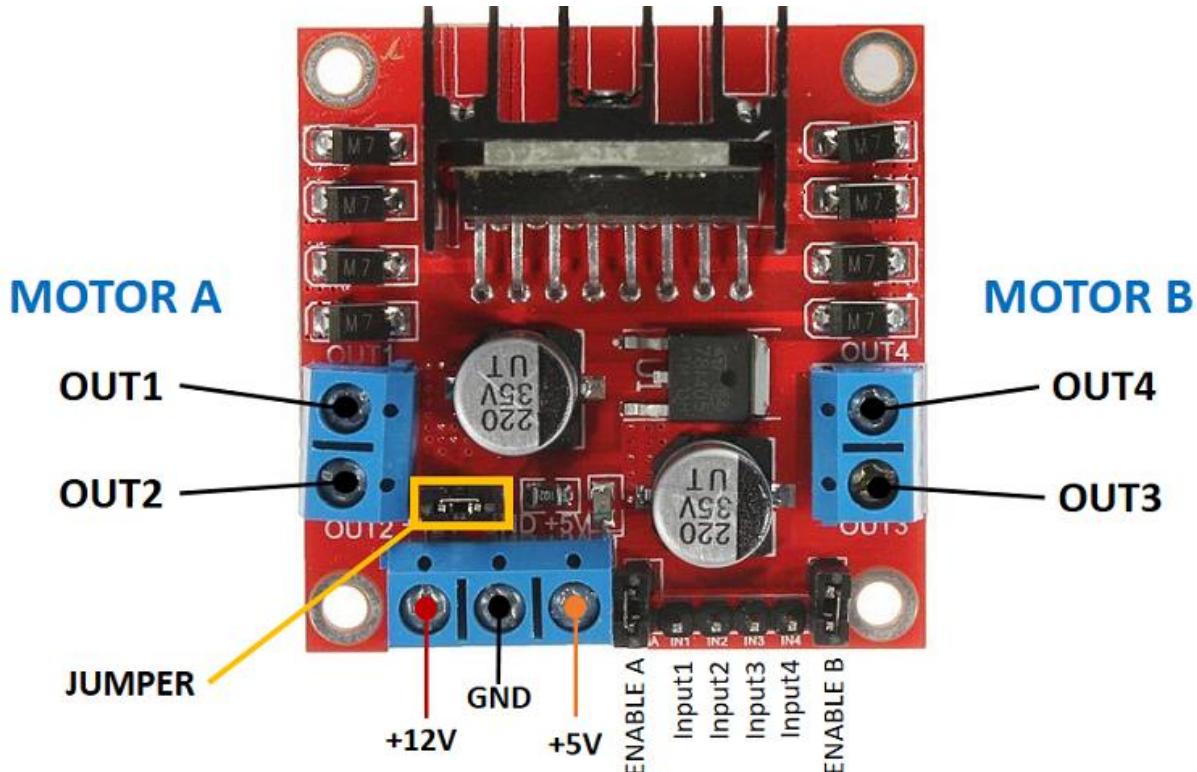
Introducing the L298N Motor Driver

As we've mentioned previously, there are many ways to control a DC motor. The method we'll use here is suitable for most hobbyist motors that require 6V or 12V to operate.

We're going to use the L298N motor driver that can handle up to 3A at 35V, which is suitable for what we want to do. Additionally, it allows us to drive two DC motors simultaneously, perfect for building a robot.

L298N Motor Driver Pinout

Let's take a look at the L298N motor driver pinout and see how it works.



The motor driver has a two-terminal block on each side for each motor. OUT1 and OUT2 at the left and OUT3 and OUT4 at the right.

- **OUT1:** DC motor A + terminal
- **OUT2:** DC motor A - terminal
- **OUT3:** DC motor B + terminal
- **OUT4:** DC motor B - terminal

At the bottom, you have a three-terminal block with **+12V**, **GND**, and **+5V**. The **+12V** terminal block is used to power up the motors. The **+5V** terminal is used to power up the L298N chip. However, if the jumper is in place, the chip is powered using the motor's power supply. So, in this case, you don't supply 5V through the **+5V** terminal.

Important: despite the +12V terminal name, you can supply any voltage between 5V and 35V (but 6V to 12V is the recommended range).

Note: If you supply more than 12V, you need to remove the jumper and supply 5V to the **+5V** terminal.

With the setup we'll use here (with the jumper in place), you can supply any voltage between 6V and 12V. In our project, we'll use 4 AA 1.5V batteries that combined output approximately 6V.

In summary:

- **+12V:** The +12V terminal is where you should connect the motor's power supply.
- **GND:** power supply GND
- **+5V:** provide 5V if jumper is removed. Acts as a 5V output if jumper is in place
- **Jumper:** jumper in place – uses the motors power supply to power up the chip. Jumper removed: you need to provide 5V to the +5V terminal. If you supply more than 12V, you should remove the jumper.

At the bottom right, you have four input pins, and two enable terminals. The input pins are used to control the direction of your DC motors, and the enable pins are used to control the speed of each motor.

- **IN1** – Input 1 for Motor A
- **IN2** – Input 2 for Motor A
- **IN3** – Input 1 for Motor B
- **IN4** – Input 2 for Motor B
- **EN1** – Enable pin for Motor A
- **EN2** – Enable pin for Motor B

There are jumper caps on the enable pins by default. You need to remove those jumper caps to control the speed of your motors.

Control DC motors with the L298N

Now that you're familiar with the L298N Motor Driver, let's see how to use it to control your DC motors.

The Enable Pins

The enable pins are like an ON and OFF switch for your motors. For example:

If you send a HIGH signal to the enable 1 pin, motor A is ready to be controlled and at the maximum speed. If you send a LOW signal to the enable 1 pin, motor A turns off.

If you send a PWM signal, you can control the speed of the motor. The motor speed is proportional to the duty cycle. However, note that for small duty cycles, the motors might not spin, and make a continuous buzz sound.

SIGNAL SENT TO THE ENABLE PIN	MOTOR STATE
HIGH	Motor enabled
LOW	Motor not enabled
PWM	Motor enabled: speed proportional to duty cycle

The Input Pins

The input pins control the direction the motors are spinning. Input 1 and input 2 control motor A, and input 3 and 4 control motor B.

If you apply LOW to input1 and HIGH to input 2, the motor will spin forward. If you apply power the other way around: HIGH to input 1 and LOW to input 2, the motor will rotate backwards. Motor B is controlled using the same method.

So, if you want your robot to move forward, both motors should be rotating forward. To make it go backwards, both should be rotating backwards.

To turn the robot in one direction, you need to spin the opposite motor faster. For example, to make the robot turn right, we'll enable the motor at the left, and disable the motor at the right.

DIRECTION	INPUT 1	INPUT 2	INPUT 3	INPUT 4
Forward	0	1	0	1
Backward	1	0	1	0
Right	0	1	0	0
Left	0	0	0	1
Stop	0	0	0	0

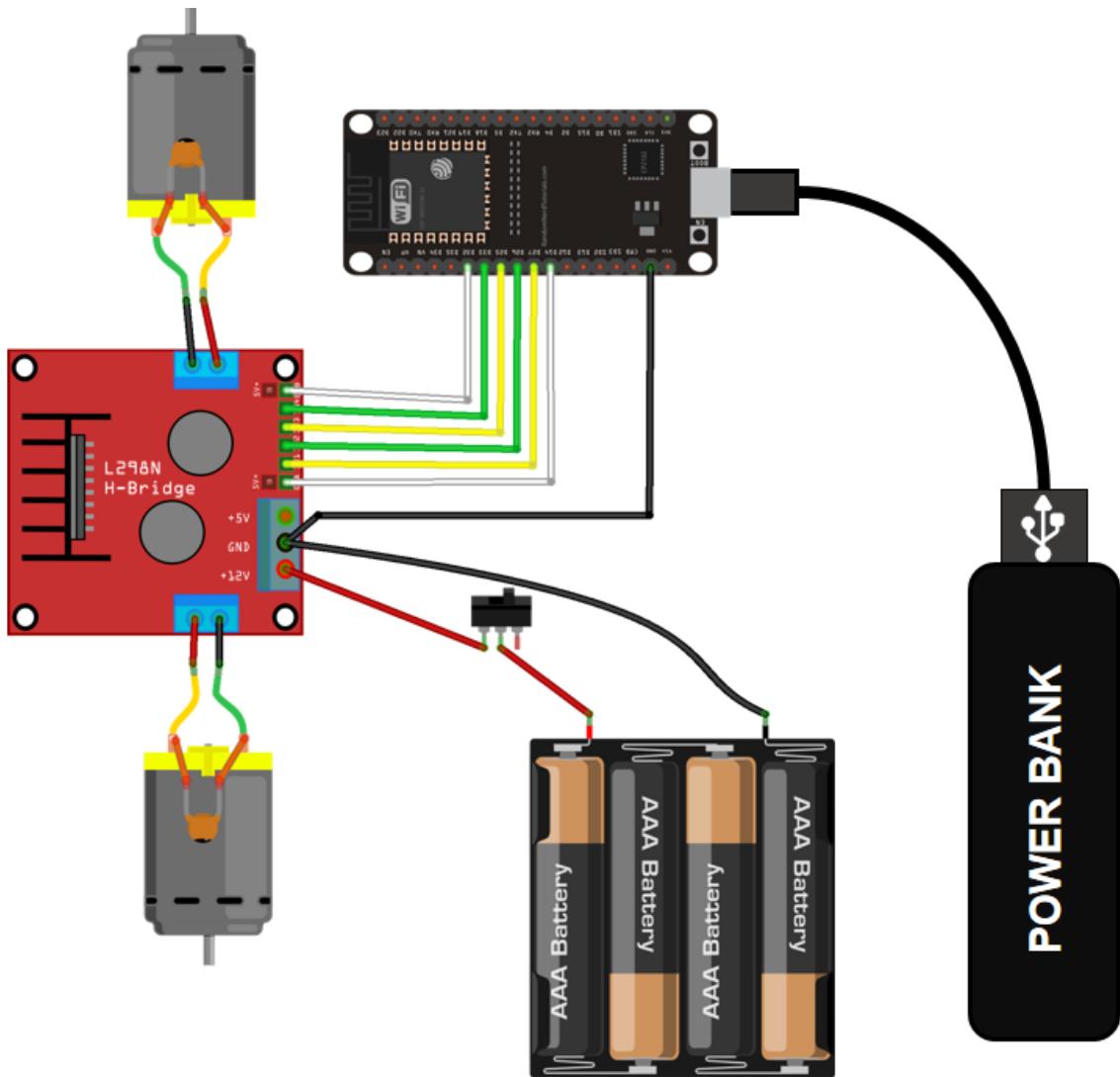
Now that you know how to control DC motors with the L298N motor driver, let's assemble the robot and build the circuit.

Assembling the Chassis

Assembling the Smart Robot Chassis Kit is very straightforward. If you need some guidance, take a look at Unit 3 of this Project.

Wiring the Circuit

After assembling the robot chassis, you can wire the circuit by following the next schematic diagram.



Start by connecting the ESP32 to the motor driver as shown in the schematic diagram. You can either use a mini breadboard or a stripboard to place your ESP32 and build the circuit. The following table shows the connections between the ESP32 and the L298N Motor Driver.

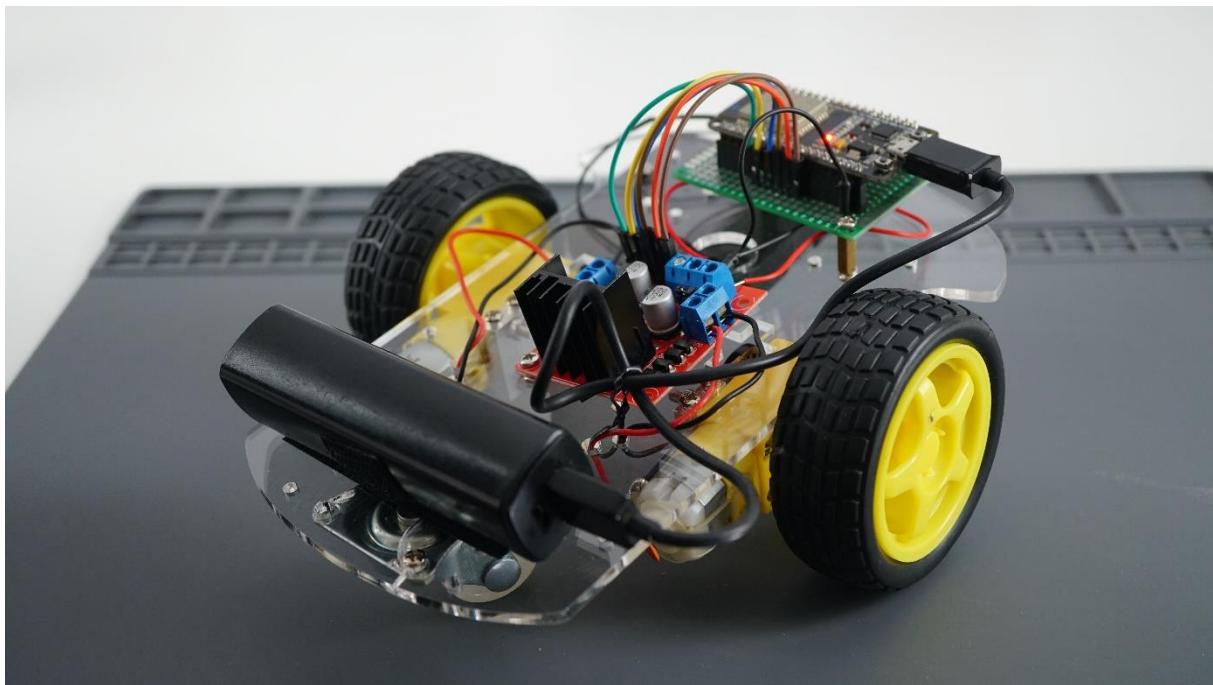
L298N MOTOR DRIVER	ESP32
IN1	GPIO 27
IN2	GPIO 26
ENA (Enable pin for Motor A)	GPIO 14
IN3	GPIO 33
IN4	GPIO 25
ENB (Enable pin for Motor B)	GPIO 32

After that, wire each motor to their terminal blocks. We recommend soldering a 0.1 uF ceramic capacitor to each motor's positive and negative terminals, as shown in the diagram, to help smooth out any voltage spikes.

Additionally, you can solder a slider switch to the red wire that comes from the battery pack. This way, you can turn the power to the motors and motor driver on and off. Finally, power the motors by connecting a 4 AA battery pack to the motor driver power blocks. Since you want your robot to be portable, the ESP32 will be powered using a powerbank. You can attach the powerbank to the robot chassis using velcro, for example.

Don't connect the powerbank yet, because first you need to upload some code to the ESP32.

Your robot should look similar to the following figure:



Continues...

Go to the next Unit to program the ESP32 with a web server to control the robot.

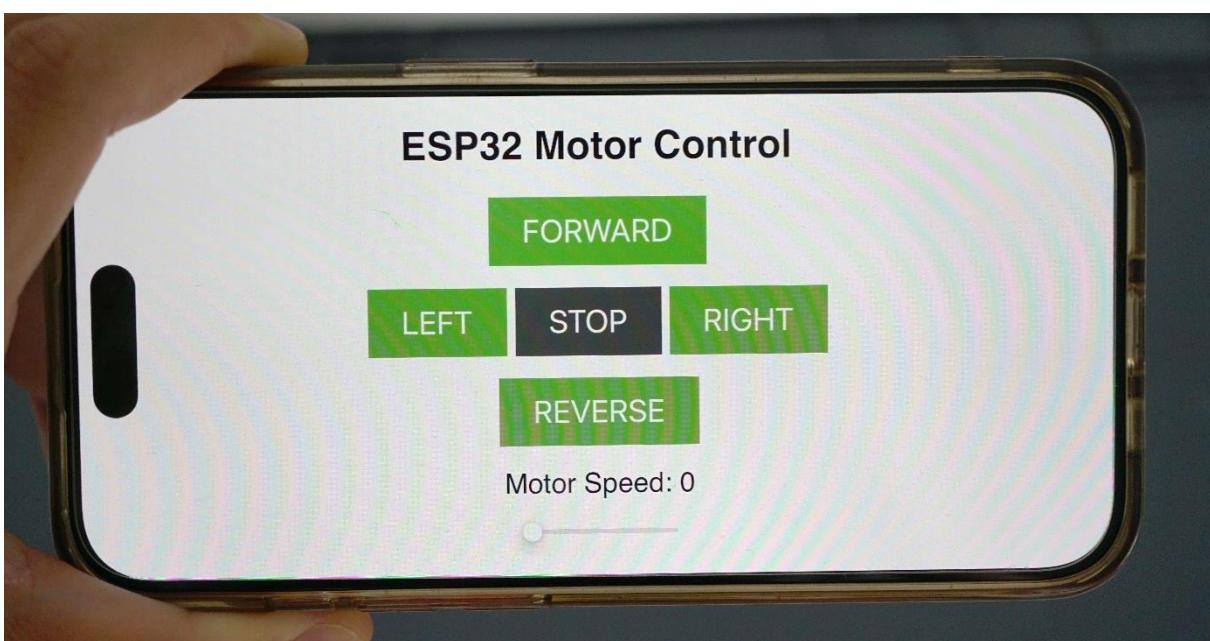
Unit 2 - Remote Controlled Wi-Fi Car Robot (Part 2/2)



This is Part 2 of the ESP32 Wi-Fi Car Robot project. In the previous Unit, you've learned how to control DC motors using the L298N motor driver. In this part, we're going to build the web server with the ESP32 to control the Robot.

Project Overview

Let's take a look at the web server.



As you can see, you have five controls to move the robot forward, reverse, right, left, and stop. You also have a slider to control the speed. You can select 0, 25, 50, 75, or 100% to control the motor speed.

Code

Copy the following code to your Arduino IDE. Don't upload it yet. First, we'll take a quick look at how it works first.

- [Click here to download the code.](#)

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

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

// Create an instance of the WebServer on port 80
WebServer server(80);

// Motor 1
int motor1Pin1 = 27;
int motor1Pin2 = 26;
int enable1Pin = 14;

// Motor 2
int motor2Pin1 = 33;
int motor2Pin2 = 25;
int enable2Pin = 32;

// Setting PWM properties
const int freq = 30000;
const int resolution = 8;
int dutyCycle = 0;

String valueString = String(0);

void handleRoot() {
    const char html[] PROGMEM = R"rawliteral(
    <!DOCTYPE HTML><html>
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="icon" href="data:,>
        <style>
            html { font-family: Helvetica; display: inline-block; margin: 0px auto;
                    text-align: center; }
            .button { -webkit-user-select: none; -moz-user-select: none;
                      -ms-user-select: none; user-select: none;
                      background-color: #4CAF50; border: none; color: white;
                      padding: 12px 28px; text-decoration: none;
                      font-size: 26px; margin: 1px; cursor: pointer; }
            .button2 {background-color: #555555;}
        </style>
    </head>
    <body>
        <h1>Control Robot</h1>
        <div>
            <input type="button" value="Forward" class="button" onclick="moveForward()"/>
            <input type="button" value="Reverse" class="button" onclick="moveReverse()"/>
            <input type="button" value="Left" class="button" onclick="moveLeft()"/>
            <input type="button" value="Right" class="button" onclick="moveRight()"/>
            <input type="button" value="Stop" class="button" onclick="moveStop()"/>
        </div>
        <div>
            <input type="range" min="0" max="100" value="50" oninput="updateSliderValue()"/>
            <span>Motor Speed:</span>
            <span>50</span>
        </div>
    </body>
)<raw>
```

```

<script>
    function moveForward() { fetch('/forward'); }
    function moveLeft() { fetch('/left'); }
    function stopRobot() { fetch('/stop'); }
    function moveRight() { fetch('/right'); }
    function moveReverse() { fetch('/reverse'); }

    function updateMotorSpeed(pos) {
        document.getElementById('motorSpeed').innerHTML = pos;
        fetch(`/speed?value=${pos}`);
    }
</script>
</head>
<body>
    <h1>ESP32 Motor Control</h1>
    <p><button class="button" onclick="moveForward()">FORWARD</button></p>
    <div style="clear: both;">
        <p>
            <button class="button" onclick="moveLeft()">LEFT</button>
            <button class="button button2" onclick="stopRobot()">STOP</button>
            <button class="button" onclick="moveRight()">RIGHT</button>
        </p>
    </div>
    <p><button class="button" onclick="moveReverse()">REVERSE</button></p>
    <p>Motor Speed: <span id="motorSpeed">0</span></p>
    <input type="range" min="0" max="100" step="25"
        id="motorSlider" oninput="updateMotorSpeed(this.value)" value="0"/>
</body>
</html>)rawliteral";
server.send(200, "text/html", html);
}

void handleForward() {
    Serial.println("Forward");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, HIGH);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, HIGH);
    server.send(200);
}

void handleLeft() {
    Serial.println("Left");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, HIGH);
    server.send(200);
}

void handleStop() {
    Serial.println("Stop");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, LOW);
    server.send(200);
}

void handleRight() {
    Serial.println("Right");
}

```

```

        digitalWrite(motor1Pin1, LOW);
        digitalWrite(motor1Pin2, HIGH);
        digitalWrite(motor2Pin1, LOW);
        digitalWrite(motor2Pin2, LOW);
        server.send(200);
    }

void handleReverse() {
    Serial.println("Reverse");
    digitalWrite(motor1Pin1, HIGH);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, HIGH);
    digitalWrite(motor2Pin2, LOW);
    server.send(200);
}

void handleSpeed() {
    if (server.hasArg("value")) {
        valueString = server.arg("value");
        int value = valueString.toInt();
        if (value == 0) {
            ledcWrite(enable1Pin, 0);
            ledcWrite(enable2Pin, 0);
            digitalWrite(motor1Pin1, LOW);
            digitalWrite(motor1Pin2, LOW);
            digitalWrite(motor2Pin1, LOW);
            digitalWrite(motor2Pin2, LOW);
        } else {
            dutyCycle = map(value, 25, 100, 200, 255);
            ledcWrite(enable1Pin, dutyCycle);
            ledcWrite(enable2Pin, dutyCycle);
            Serial.println("Motor speed set to " + String(value));
        }
    }
    server.send(200);
}

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

    // Set the Motor pins as outputs
    pinMode(motor1Pin1, OUTPUT);
    pinMode(motor1Pin2, OUTPUT);
    pinMode(motor2Pin1, OUTPUT);
    pinMode(motor2Pin2, OUTPUT);

    // Configure PWM Pins
    ledcAttach(enable1Pin, freq, resolution);
    ledcAttach(enable2Pin, freq, resolution);

    // Initialize PWM with 0 duty cycle
    ledcWrite(enable1Pin, 0);
    ledcWrite(enable2Pin, 0);

    // Connect to Wi-Fi
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
}

```

```

    }
    Serial.println("");
    Serial.println("WiFi connected.");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());

    // Define routes
    server.on("/", handleRoot);
    server.on("/forward", handleForward);
    server.on("/left", handleLeft);
    server.on("/stop", handleStop);
    server.on("/right", handleRight);
    server.on("/reverse", handleReverse);
    server.on("/speed", handleSpeed);
    // Start the server
    server.begin();
}

void loop() {
    server.handleClient();
}

```

How Does the Code Work?

We've covered how to build a web server with the ESP32 with great detail in previous modules. So, we'll just take a look at the code sections that are relevant for this project.

Setting your Network Credentials

Start by typing your network credentials in the following variables so that the ESP32 can connect to your local network.

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

Creating Variables for the Motor Driver Pins

Next, create variables for the motor input pins, and the motor enable pins. These pins are connected to the motor driver, which controls the direction and speed of the motors.

```
// Motor 1
int motor1Pin1 = 27;
int motor1Pin2 = 26;
int enable1Pin = 14;
// Motor 2
int motor2Pin1 = 33;
int motor2Pin2 = 25;
int enable2Pin = 32;
```

Setting PWM Properties

To control the speed of the motors, you'll send PWM signals to the enable pins.

Define the PWM settings as follows:

```
// Setting PWM properties
const int freq = 30000;
const int resolution = 8;
int dutyCycle = 0;
```

setup()

In the `setup()`, you set the motor pins as outputs:

```
// Set the Motor pins as outputs
pinMode(motor1Pin1, OUTPUT);
pinMode(motor1Pin2, OUTPUT);
pinMode(motor2Pin1, OUTPUT);
pinMode(motor2Pin2, OUTPUT);
```

Configure the enable pins as PWM outputs with the PWM properties defined earlier.

```
// Configure PWM Pins
ledcAttach(enable1Pin, freq, resolution);
ledcAttach(enable2Pin, freq, resolution);
```

Finally, you generate the PWM signal with a defined duty cycle using the `ledcWrite()` function. When the code first starts, we want the motors to be off by default, so we're sending 0% duty cycle.

```
// Initialize PWM with 0 duty cycle
ledcWrite(enable1Pin, 0);
ledcWrite(enable2Pin, 0);
```

The following code in the `setup()` connects your ESP32 to your local network and prints the 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();
```

Controlling the Robot

Each motor action, such as moving forward, turning left, stopping, etc., is handled by specific routes. On the web server, when you click on the buttons, it will make different requests on different routes.

```
// Define routes
server.on("/", handleRoot);
server.on("/forward", handleForward);
server.on("/left", handleLeft);
server.on("/stop", handleStop);
server.on("/right", handleRight);
server.on("/reverse", handleReverse);
server.on("/speed", handleSpeed);
```

The `handleForward`, `handleLeft`, `handleStop`, `handleRight` and `handleReverse` functions will turn the appropriate GPIOs on and off to achieve the desired result.

```
void handleForward() {
    Serial.println("Forward");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, HIGH);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, HIGH);
    server.send(200);
}

void handleLeft() {
    Serial.println("Left");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, HIGH);
    server.send(200);
}

void handleStop() {
    Serial.println("Stop");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, LOW);
    server.send(200);
}

void handleRight() {
    Serial.println("Right");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, HIGH);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, LOW);
    server.send(200);
}
```

```

}

void handleReverse() {
    Serial.println("Reverse");
    digitalWrite(motor1Pin1, HIGH);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, HIGH);
    digitalWrite(motor2Pin2, LOW);
    server.send(200);
}

```

The combination of HIGH and LOW signals required to move the robot in a specific direction was explained in the previous Unit: **Remote Controlled Wi-Fi Car Robot – Part 1/2.**

Controlling the Speed

To control the motor speed using a slider, the slider's value is sent to the server, which adjusts the PWM duty cycle accordingly.

This is the part of the JavaScript code that sends a request to the server with the current slider value when you move the slider to a new position:

```

function updateMotorSpeed(pos) {
    document.getElementById('motorSpeed').innerHTML = pos;
    fetch(`/speed?value=${pos}`);
}

```

It will make a request with the following format:

/speed?value=SELECTED_SPEED_VALUE

On the ESP32, on the `handleSpeed` function, we start by getting the value of the slider and saving it on the `value` variable as follows.

```

void handleSpeed() {
    if (server.hasArg("value")) {
        valueString = server.arg("value");
        int value = valueString.toInt();
    }
}

```

If the slider value is zero, the motors are stopped. So, we set the duty cycle to 0 and all motor pins to LOW.

```

if (value == 0) {
    ledcWrite(enable1Pin, 0);
    ledcWrite(enable2Pin, 0);
    digitalWrite(motor1Pin1, LOW);
}

```

```
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, LOW);
}
```

Otherwise, we set the speed of the motors by adjusting the duty cycle taking into account the slider value.

```
} else {
    dutyCycle = map(value, 25, 100, 200, 255);
    ledcWrite(enable1Pin, dutyCycle);
    ledcWrite(enable2Pin, dutyCycle);
    Serial.println("Motor speed set to " + String(value));
}
```

We calculate the duty cycle based on the slider value using the `map()` function ([learn more about the Arduino map\(\) function](#)). In this case, we set the duty cycle to start at 200 because lower values won't make the robot move, and the motors will make a weird buzz sound.

Displaying the web page

The following part of the code displays a web page with 5 buttons to control the motor and a slider to set the motor speed. This time, instead of redirecting to different URL routes to make a new request to the server, we use the JavaScript `fetch()` function to make requests.

```
void handleRoot() {
    const char PROGMEM html[] = R"rawliteral(
        <!DOCTYPE HTML><html>
            <head>
                <meta name="viewport" content="width=device-width, initial-scale=1">
                <link rel="icon" href="data:,>
                <style>
                    html { font-family: Helvetica; display: inline-block; margin: 0px auto;
                           text-align: center; }
                    .button { -webkit-user-select: none; -moz-user-select: none;
                              -ms-user-select: none; user-select: none;
                              background-color: #4CAF50; border: none; color: white;
                              padding: 12px 28px; text-decoration: none;
                              font-size: 26px; margin: 1px; cursor: pointer; }
                    .button2 {background-color: #555555; }
                </style>
                <script>
                    function moveForward() { fetch('/forward'); }
                    function moveLeft() { fetch('/left'); }
                    function stopRobot() { fetch('/stop'); }
                    function moveRight() { fetch('/right'); }
                    function moveReverse() { fetch('/reverse'); }
    )rawliteral";
```

```

        function updateMotorSpeed(pos) {
            document.getElementById('motorSpeed').innerHTML = pos;
            fetch(`/speed?value=${pos}`);
        }
    </script>
</head>
<body>
    <h1>ESP32 Motor Control</h1>
    <p><button class="button" onclick="moveForward()">FORWARD</button></p>
    <div style="clear: both;">
        <p>
            <button class="button" onclick="moveLeft()">LEFT</button>
            <button class="button button2" onclick="stopRobot()">STOP</button>
            <button class="button" onclick="moveRight()">RIGHT</button>
        </p>
    </div>
    <p><button class="button" onclick="moveReverse()">REVERSE</button></p>
    <p>Motor Speed: <span id="motorSpeed">0</span></p>
    <input type="range" min="0" max="100" step="25"
        id="motorSlider" oninput="updateMotorSpeed(this.value)" value="0"/>
</body>
</html>)rawliteral";
server.send(200, "text/html", html);
}

```

The `html` variable contains all the text to build the HTML page. The `PROGMEM` keyword means that the variable will be saved in flash. The `R"rawliteral"` allows us to spread the text on different lines, instead of having to concatenate the HTML text on different lines. This is useful to improve the readability of our `html` variable.

The `R` means “Treat everything between these delimiters as a raw string”. That is, everything between `rawliteral`(at the beginning of the string) and `rawliteral` (at the end of the string). You can use any other word instead of `rawliteral` as long as that word is not used inside your string.

For example:

```
const char html[] PROGMEM = R"rawliteral( YOUR LONG STRING GOES HERE) rawliteral";
```

Testing the Web Server

Now, you can test the web server. Make sure you've modified the code to include your network credentials. Don't forget to check if you have the right board and COM port selected. Then, click the upload button. When the upload is finished, open the Serial Monitor at a baud rate of 115200.

Press the ESP32 enable button, and you'll get the ESP32 IP address.

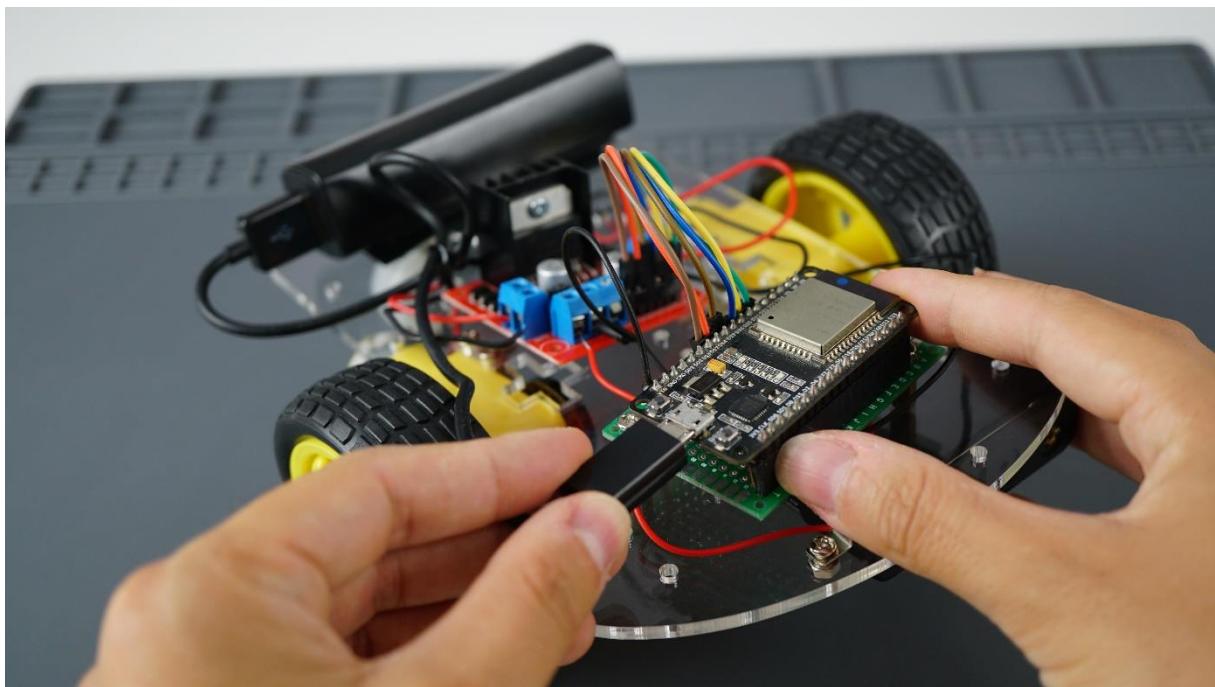
Serial Monitor X

Message (Enter to send message to 'DOIT ESP32 DEVKIT V1') New Line 115200 baud

```
rst:0x1 (POWERON_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:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Connecting to ML.. 100%
...
WiFi connected.
IP address:
192.168.1.140
```

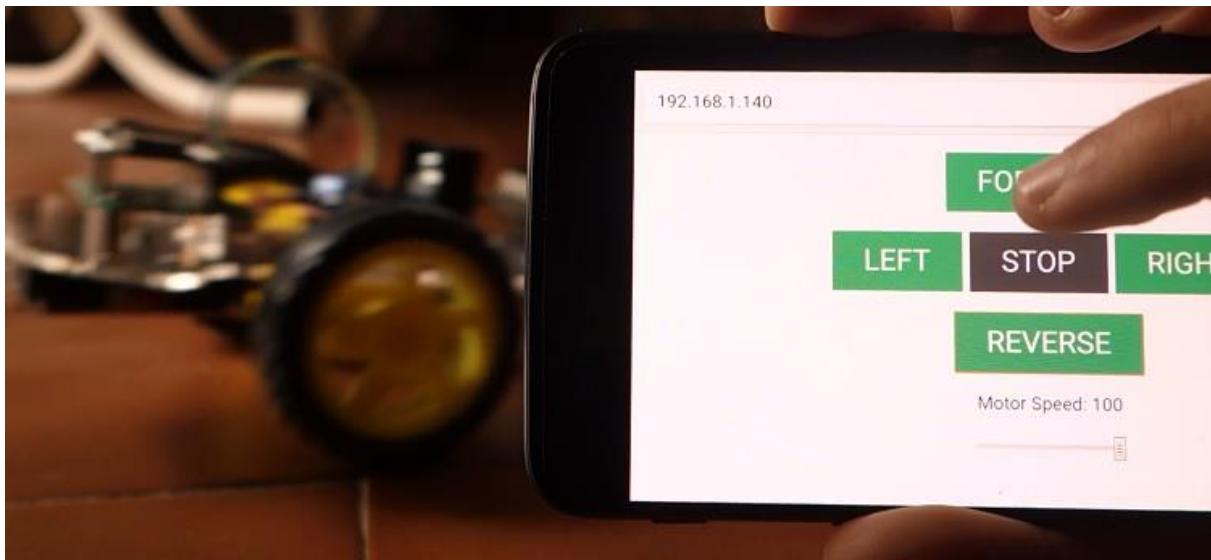
Ln 158, Col 68 DOIT ESP32 DEVKIT V1 on COM3

Disconnect the ESP32 from your computer and power it with the powerbank.



Make sure you also have the 4 AA batteries in place, and the slider switch turned on. Open your browser, and paste the ESP32 IP address to access the web server. You can use any device with a browser inside your local network to control the robot.

Now, you can control your robot.

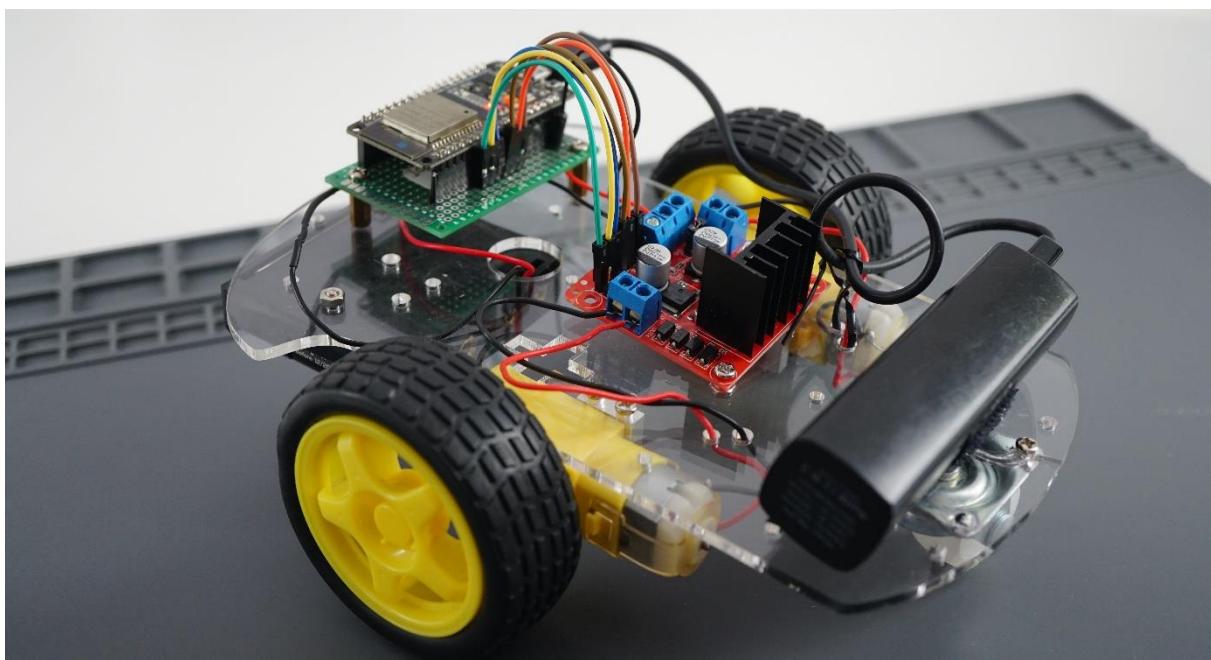


Important note: If the motors are spinning in the wrong direction, you can simply switch the motor wires. For example, switch the wire that goes to OUT1 with OUT2. Or OUT3 with OUT4. That should solve your problem.

Demonstration

Congratulations! The ESP32 Wi-Fi Remote Controlled Car Robot is complete! You can make your robot go forward, backward, right, and left. You can stop it by tapping the STOP button. A cool feature of this robot is that you can also adjust the speed with the slider.

The robot works pretty well and responds instantaneously to the commands.



Wrapping Up

Now, we encourage you to modify the robot with some upgrades. For example:

- Add an RGB LED that changes color depending on the direction the robot is moving;
- Add an ultrasonic sensor that makes the robot stop when it senses an obstacle;

This is a great project to put into practice many of the concepts learned throughout the course.

We hope you had fun building the robot.

Unit 3 - Assembling the Smart Robot Car Chassis Kit

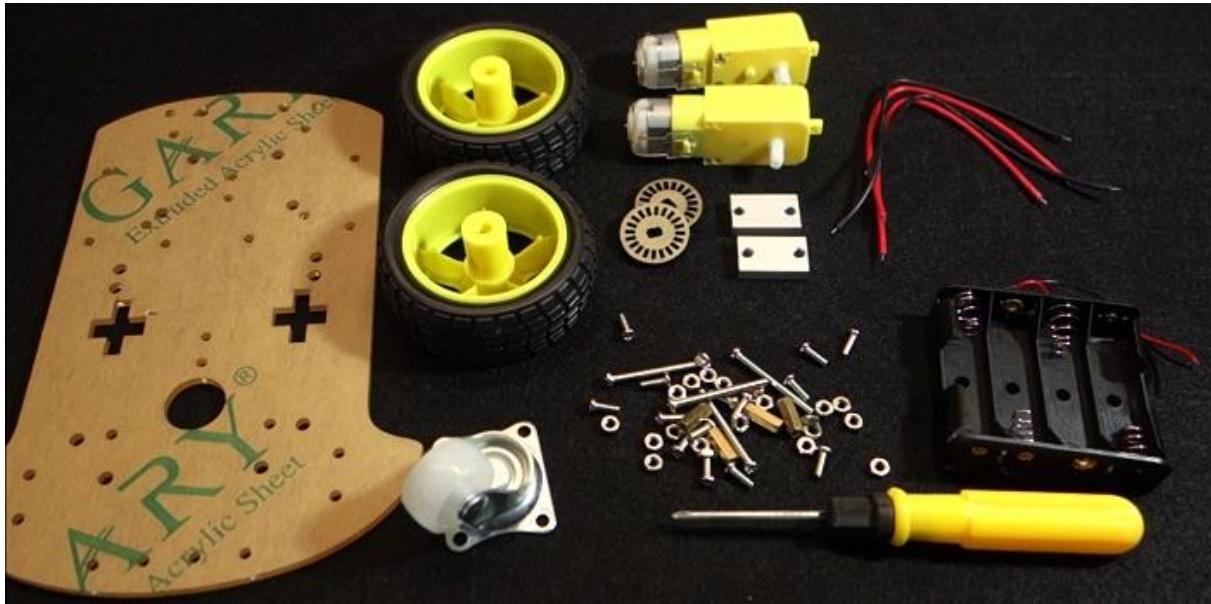
In this Unit, you're going to learn how to build a smart robot car chassis kit that is commonly used to build a smart car robot with the ESP32, ESP8266, Arduino boards, etc.

How to Assemble the Kit

- 1) This is the package that you get with the [Smart Robot Car Chassis Kit](#).



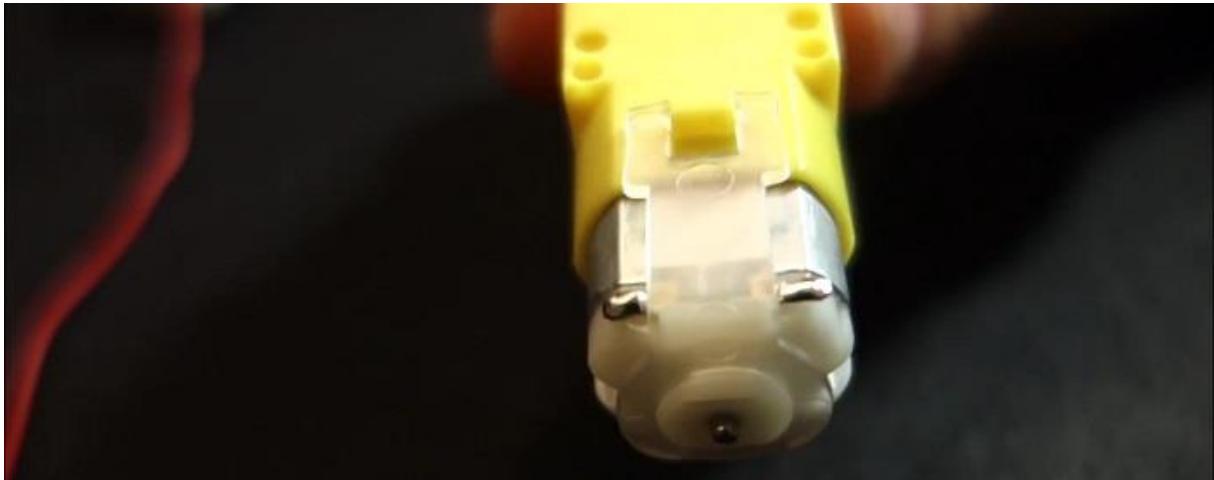
- 2) Open the package – it comes with one screwdriver, two DC motors, two wheels, and one acrylic car chassis. There is also a small plastic bag that comes with a battery holder, a small wheel for the front, some bolts and screws, four wires, and other required components to assemble the robot.



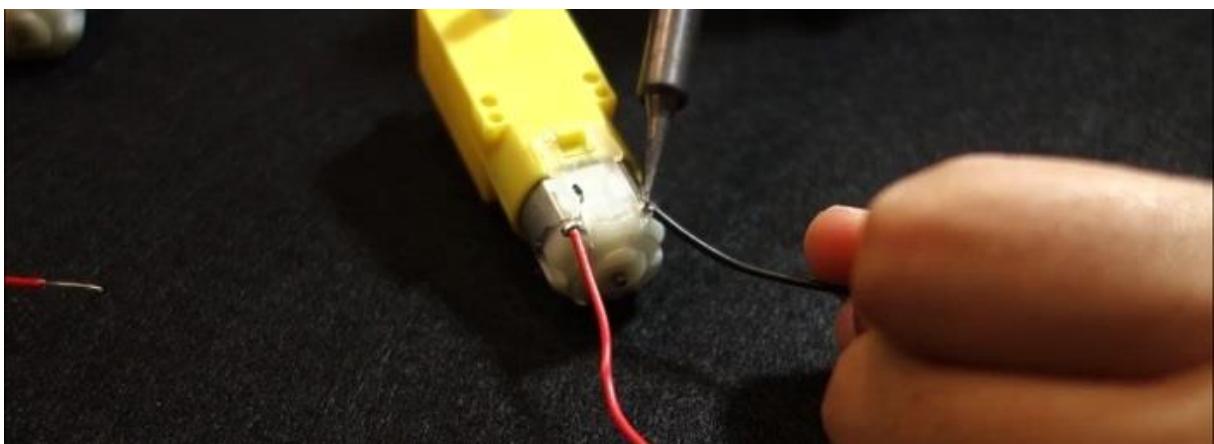
- 3) Although this kit comes with the needed wires, since they are a bit short and not very flexible, we've decided to replace them with other wires, because we can easily adjust their length. You'll need one red wire and one black wire for each motor. You can use a wire cutter to cut the wires to your desired length.



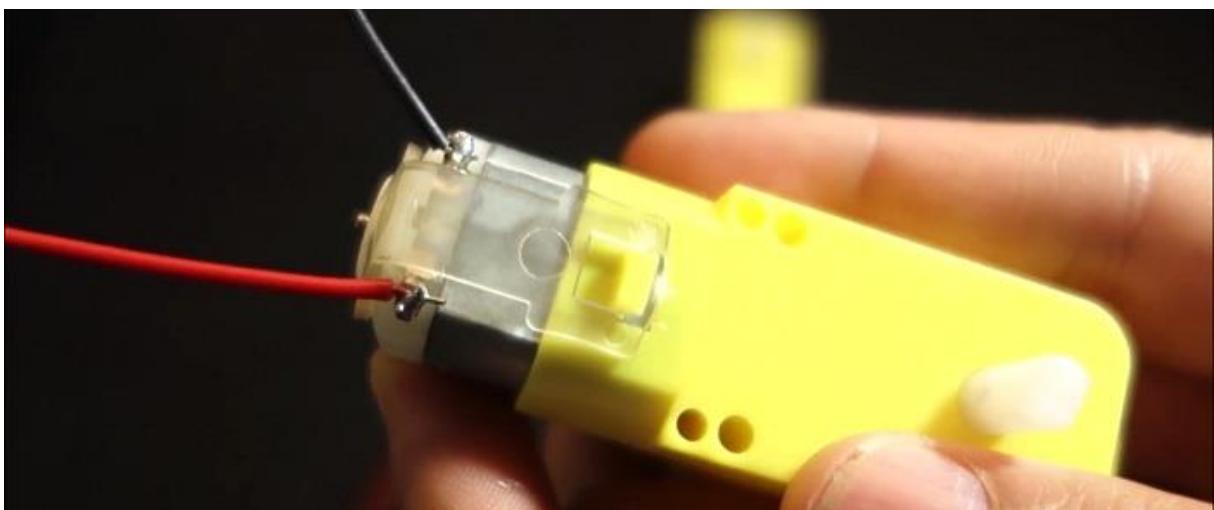
- 4) After preparing all four wires, you need to solder them to the DC motors. Tin the DC motor pins.



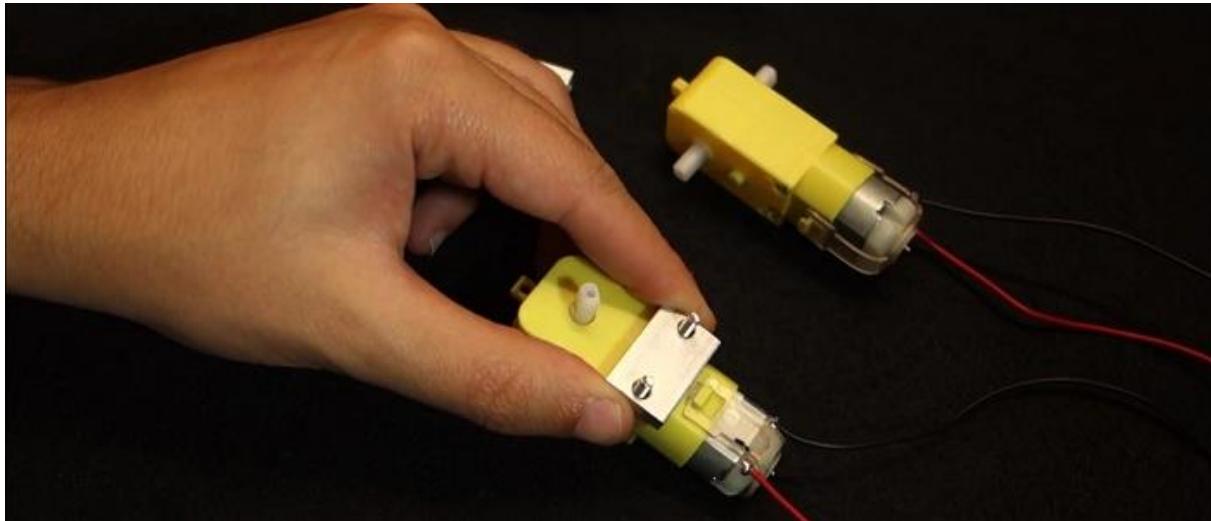
- 5) Then, grab the wire and solder it to the DC motor pin. Repeat that process to all the other wires.



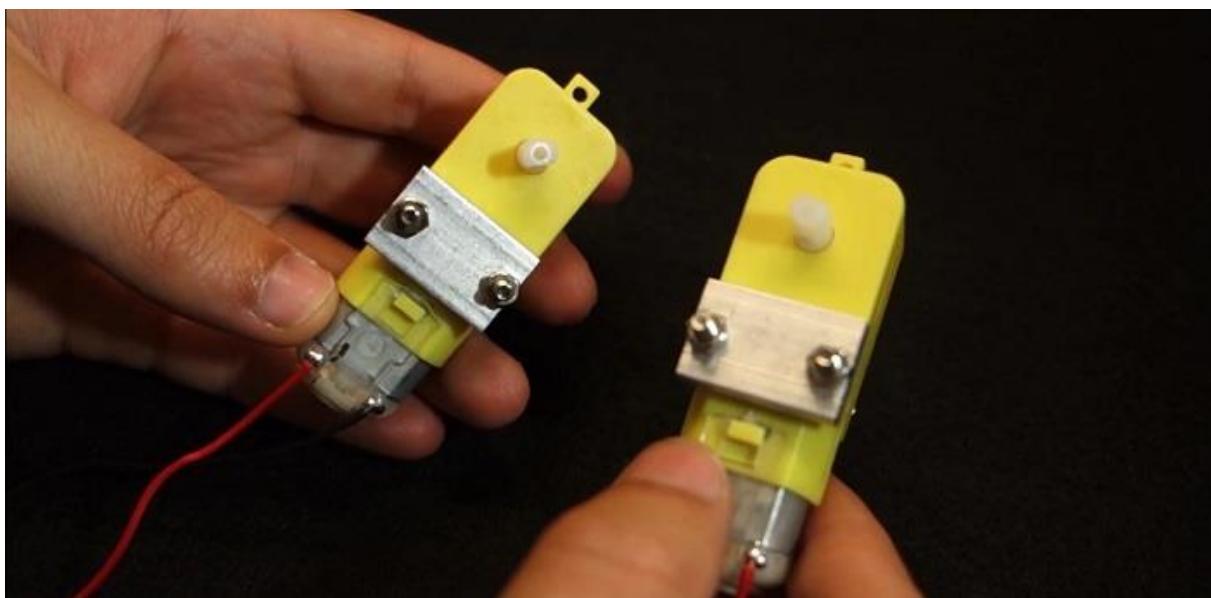
This is how the motors look like after soldering the wires.



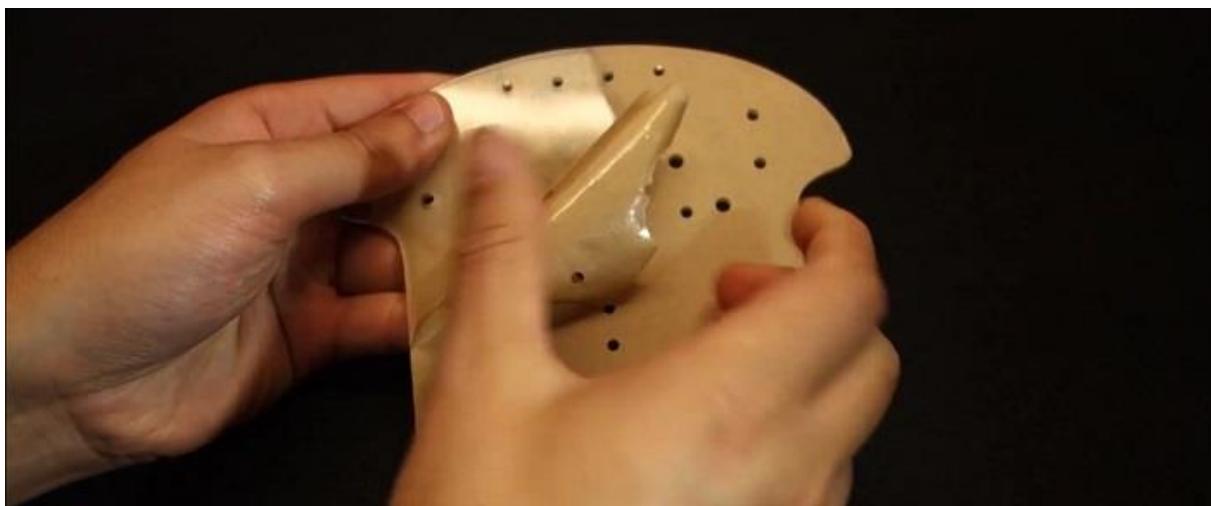
- 6) Now, you need the screwdriver, bolts and screws, and those metal pieces. Start by attaching the metal pieces to the DC motors.



7) Here's how they look like after this step:



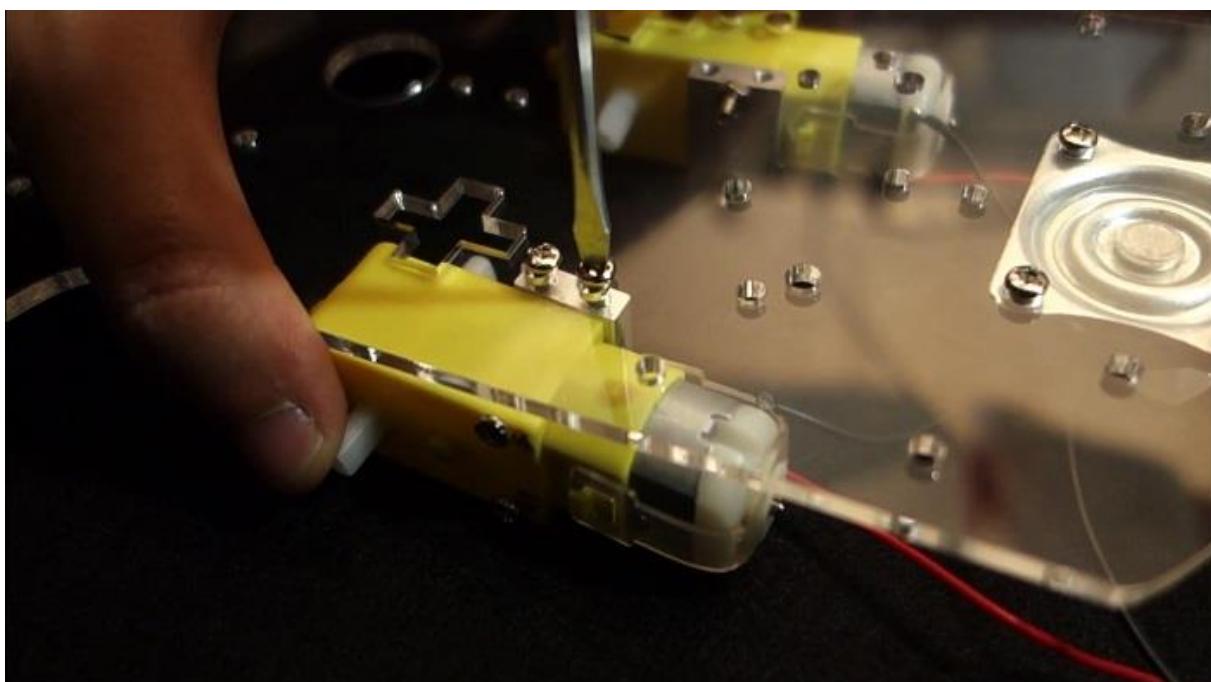
8) Remove the protective adhesive from the acrylic chassis on both sides.



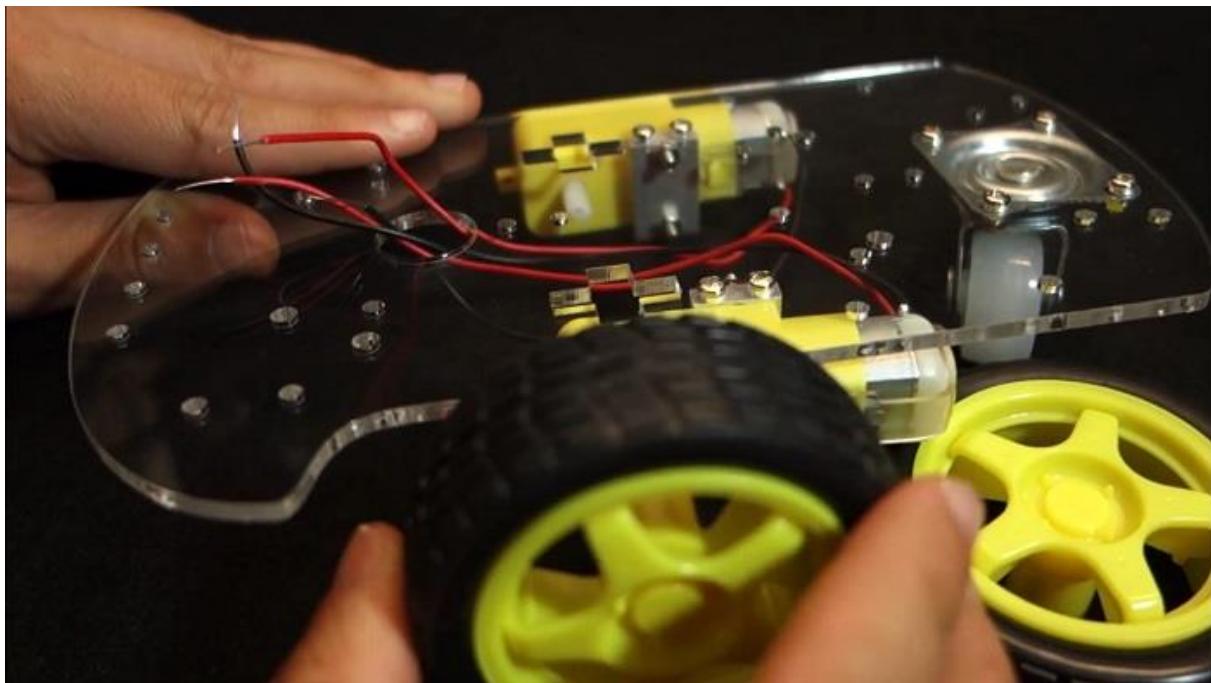
9) Grab the small wheel and attach it to the front part of the chassis.



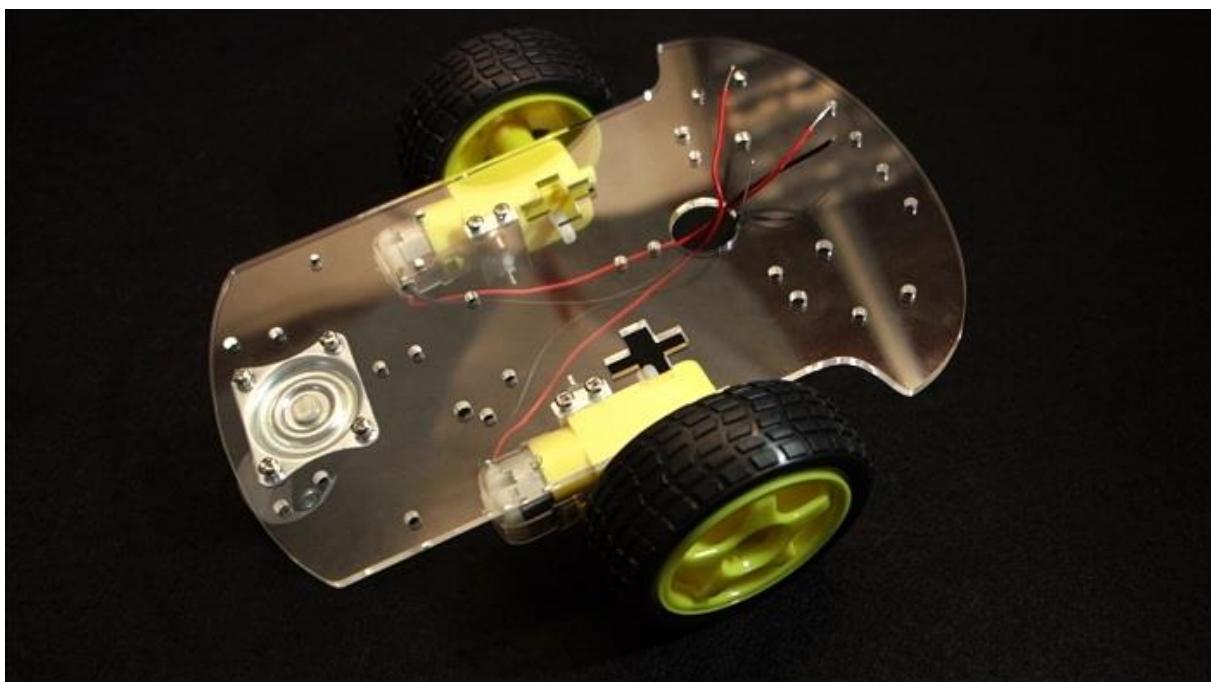
10) Finally, it is time to attach the DC motors to the chassis.



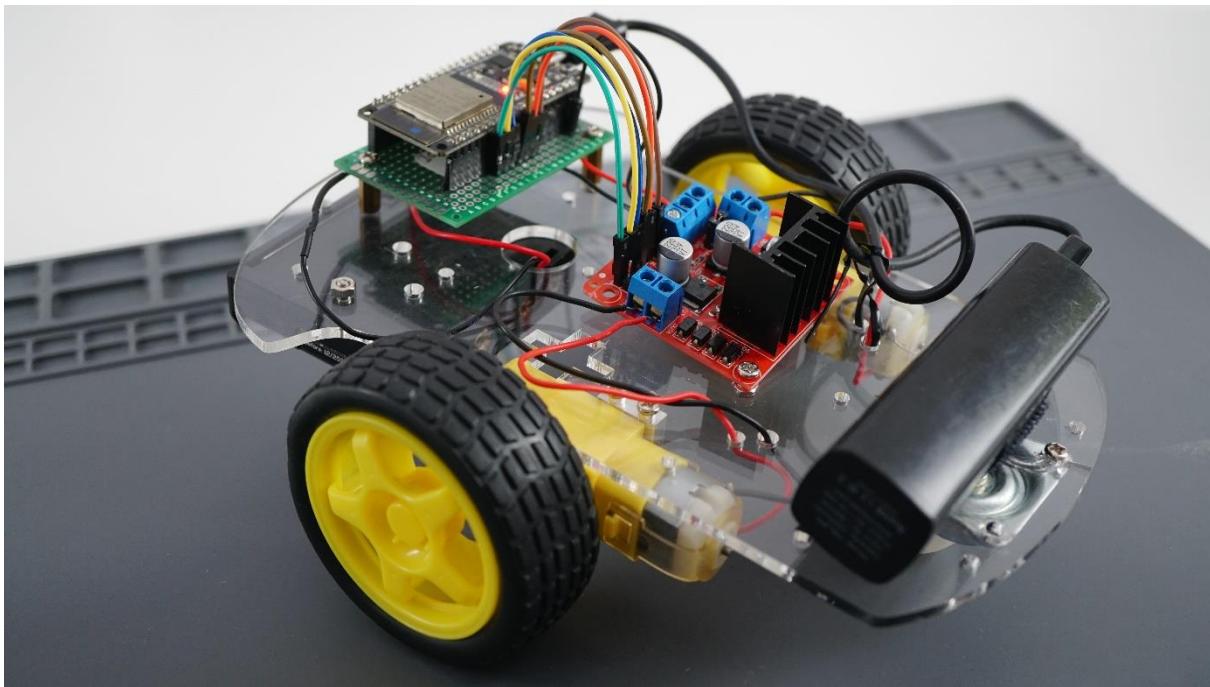
11) Lastly, connect the wheels to the DC motors.



Now, your robot car chassis is ready. Here's how it looks like after assembling:



Unit 4 - Access Point (AP) For Wi-Fi Car Robot



If you've followed the previous Units, you have a Wi-Fi car robot that requires a wireless connection to your router:

- Remote Controlled Wi-Fi Car Robot – Part 1/2
- Remote Controlled Wi-Fi Car Robot – Part 2/2
- Assembling the Smart Robot Car Chassis Kit

This means that you can just control it when it is within the range of your router. The solution to this problem is setting your ESP32 as an Access Point. This way, you don't need to be connected to a router to control your robot. In this extra Unit, you'll learn how to do that.

Having the same setup and circuit from those previous Units, you just need to upload the sketch provided below to set the ESP32 as an Access Point.

We already covered how to set the ESP32 as an access point in [Unit 5.1](#).

Uploading the Sketch

To set the ESP32 as an Access Point, upload the next sketch to your board:

- [Click here to download the code.](#)

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

// Replace with your network credentials
const char* ssid = "ESP32-Robot";
const char* password = "123456789";

// Create an instance of the WebServer on port 80
WebServer server(80);

// Motor 1
int motor1Pin1 = 27;
int motor1Pin2 = 26;
int enable1Pin = 14;

// Motor 2
int motor2Pin1 = 33;
int motor2Pin2 = 25;
int enable2Pin = 32;

// Setting PWM properties
const int freq = 30000;
const int resolution = 8;
int dutyCycle = 0;

String valueString = String(0);

void handleRoot() {
    const char html[] PROGMEM = R"rawliteral(
<!DOCTYPE HTML><html>
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" href="data:,>
    <style>
        html { font-family: Helvetica; display: inline-block; margin: 0px auto;
                text-align: center; }
        .button { -webkit-user-select: none; -moz-user-select: none;
                  -ms-user-select: none; user-select: none;
                  background-color: #4CAF50; border: none; color: white;
                  padding: 12px 28px; text-decoration: none;
                  font-size: 26px; margin: 1px; cursor: pointer; }
        .button2 {background-color: #555555;}
    </style>
    <script>
        function moveForward() { fetch('/forward'); }
        function moveLeft() { fetch('/left'); }
        function stopRobot() { fetch('/stop'); }
        function moveRight() { fetch('/right'); }
        function moveReverse() { fetch('/reverse'); }

        function updateMotorSpeed(pos) {
            document.getElementById('motorSpeed').innerHTML = pos;
            fetch(`/speed?value=${pos}`);
        }
    </script>
</head>
<body>
    <h1>ESP32 Motor Control</h1>
```

```

<p><button class="button" onclick="moveForward()">FORWARD</button></p>
<div style="clear: both;">
  <p>
    <button class="button" onclick="moveLeft()">LEFT</button>
    <button class="button button2" onclick="stopRobot()">STOP</button>
    <button class="button" onclick="moveRight()">RIGHT</button>
  </p>
</div>
<p><button class="button" onclick="moveReverse()">REVERSE</button></p>
<p>Motor Speed: <span id="motorSpeed">0</span></p>
<input type="range" min="0" max="100" step="25"
  id="motorSlider" oninput="updateMotorSpeed(this.value)" value="0"/>
</body>
</html>)rawliteral";
server.send(200, "text/html", html);
}

void handleForward() {
  Serial.println("Forward");
  digitalWrite(motor1Pin1, LOW);
  digitalWrite(motor1Pin2, HIGH);
  digitalWrite(motor2Pin1, LOW);
  digitalWrite(motor2Pin2, HIGH);
  server.send(200);
}

void handleLeft() {
  Serial.println("Left");
  digitalWrite(motor1Pin1, LOW);
  digitalWrite(motor1Pin2, LOW);
  digitalWrite(motor2Pin1, LOW);
  digitalWrite(motor2Pin2, HIGH);
  server.send(200);
}

void handleStop() {
  Serial.println("Stop");
  digitalWrite(motor1Pin1, LOW);
  digitalWrite(motor1Pin2, LOW);
  digitalWrite(motor2Pin1, LOW);
  digitalWrite(motor2Pin2, LOW);
  server.send(200);
}

void handleRight() {
  Serial.println("Right");
  digitalWrite(motor1Pin1, LOW);
  digitalWrite(motor1Pin2, HIGH);
  digitalWrite(motor2Pin1, LOW);
  digitalWrite(motor2Pin2, LOW);
  server.send(200);
}

void handleReverse() {
  Serial.println("Reverse");
  digitalWrite(motor1Pin1, HIGH);
  digitalWrite(motor1Pin2, LOW);
  digitalWrite(motor2Pin1, HIGH);
  digitalWrite(motor2Pin2, LOW);
  server.send(200);
}

```

```

void handleSpeed() {
    if (server.hasArg("value")) {
        valueString = server.arg("value");
        int value = valueString.toInt();
        if (value == 0) {
            ledcWrite(enable1Pin, 0);
            ledcWrite(enable2Pin, 0);
            digitalWrite(motor1Pin1, LOW);
            digitalWrite(motor1Pin2, LOW);
            digitalWrite(motor2Pin1, LOW);
            digitalWrite(motor2Pin2, LOW);
        } else {
            dutyCycle = map(value, 25, 100, 200, 255);
            ledcWrite(enable1Pin, dutyCycle);
            ledcWrite(enable2Pin, dutyCycle);
            Serial.println("Motor speed set to " + String(value));
        }
    }
    server.send(200);
}

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

    // Set the Motor pins as outputs
    pinMode(motor1Pin1, OUTPUT);
    pinMode(motor1Pin2, OUTPUT);
    pinMode(motor2Pin1, OUTPUT);
    pinMode(motor2Pin2, OUTPUT);

    // Configure PWM Pins
    ledcAttach(enable1Pin, freq, resolution);
    ledcAttach(enable2Pin, freq, resolution);
    // Initialize PWM with 0 duty cycle
    ledcWrite(enable1Pin, 0);
    ledcWrite(enable2Pin, 0);

    // Setting the ESP as an access point
    Serial.print("Setting AP (Access Point)...");
    WiFi.mode(WIFI_AP);
    WiFi.softAP(ssid, password);
    IPAddress IP = WiFi.softAPIP();
    Serial.print("AP IP address: ");
    Serial.println(IP);

    // Define routes
    server.on("/", handleRoot);
    server.on("/forward", handleForward);
    server.on("/left", handleLeft);
    server.on("/stop", handleStop);
    server.on("/right", handleRight);
    server.on("/reverse", handleReverse);
    server.on("/speed", handleSpeed);

    // Start the server
    server.begin();
}

void loop() {
    server.handleClient();
}

```

SSID and Password

The default SSID name for the ESP32 is **ESP32-Robot**, and the password is **123456789**, but you can modify those two variables:

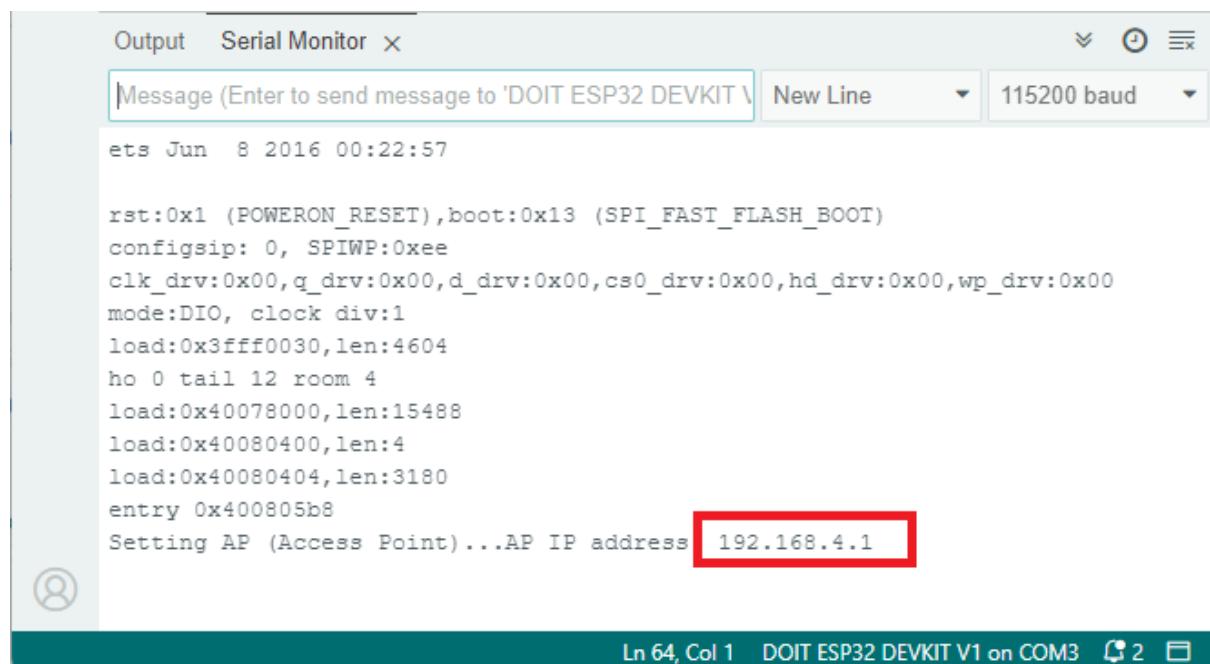
```
const char* ssid      = "ESP32-Robot";
const char* password = "123456789";
```

There's also this new section in the `setup()` function to set the ESP32 as an Access Point:

```
// Setting the ESP as an access point
Serial.print("Setting AP (Access Point)...");
WiFi.mode(WIFI_AP);
WiFi.softAP(ssid, password);
IPAddress IP = WiFi.softAPIP();
Serial.print("AP IP address: ");
Serial.println(IP);
```

The rest of the code and the web page features are similar to the sketch explained in previous Robot Project Units.

After uploading the code, reboot your ESP32 with the Serial Monitor open to print the IP address:



The screenshot shows the Arduino Serial Monitor window. The title bar says "Output" and "Serial Monitor". The message area contains the following text:

```
ets Jun  8 2016 00:22:57

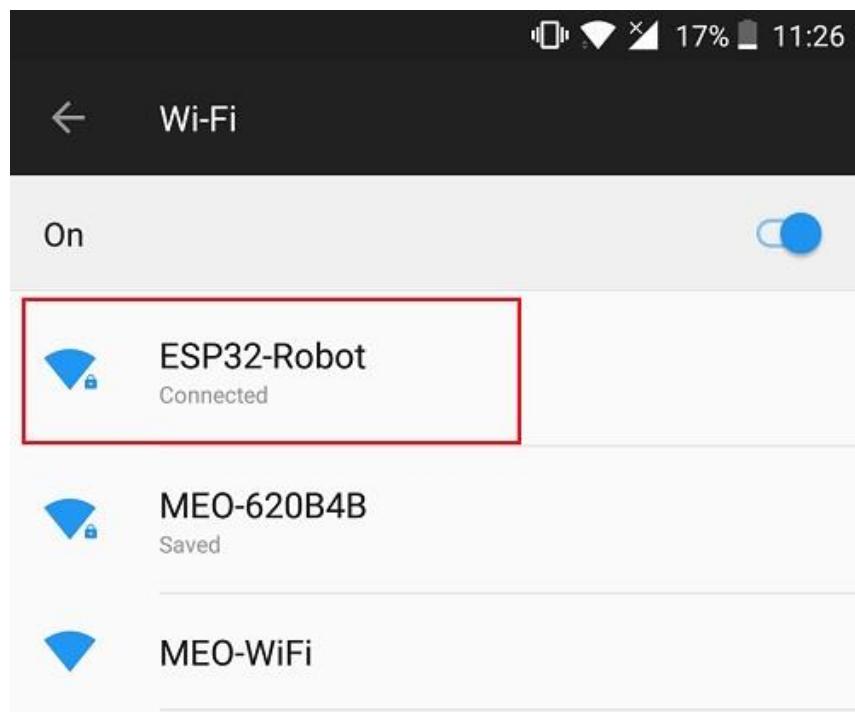
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:4604
ho 0 tail 12 room 4
load:0x40078000,len:15488
load:0x40080400,len:4
load:0x40080404,len:3180
entry 0x400805b8
Setting AP (Access Point)...AP IP address 192.168.4.1
```

The line "Setting AP (Access Point)...AP IP address 192.168.4.1" is highlighted with a red rectangle. The status bar at the bottom of the window shows "Ln 64, Col 1 DOIT ESP32 DEVKIT V1 on COM3" and some icons.

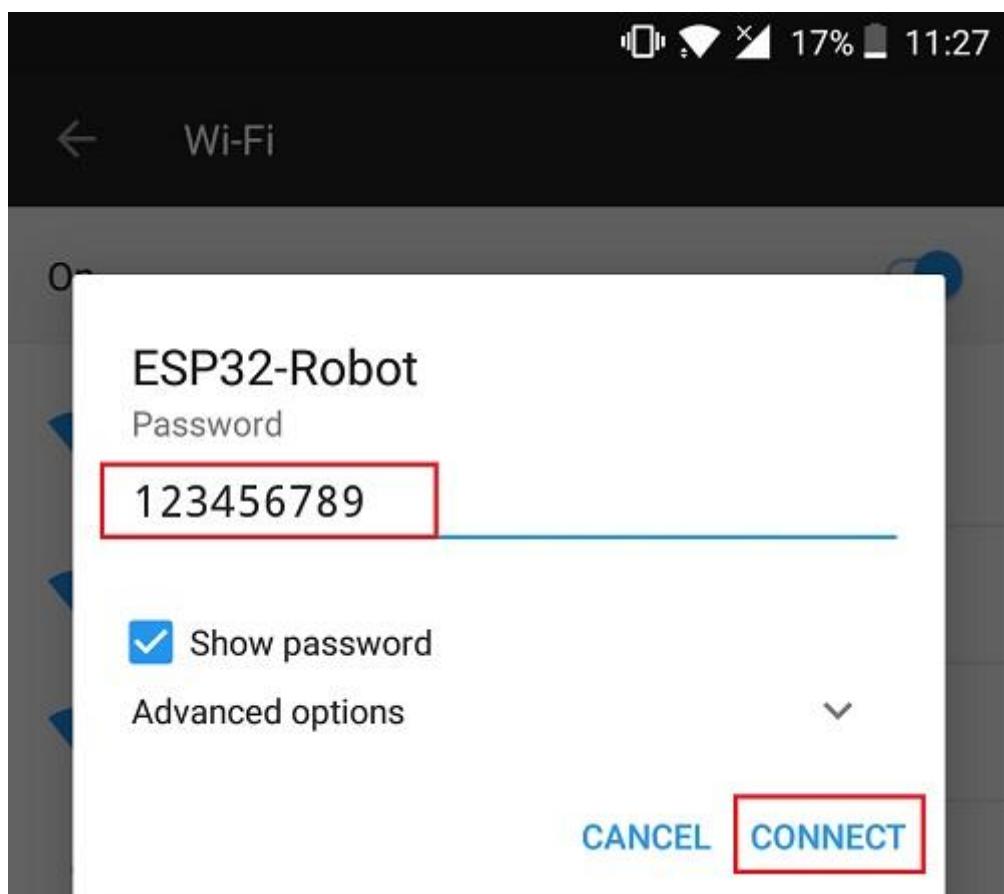
In my case it's **192.168.4.1**, save that IP, because you'll need it later.

Connect to the ESP32 AP

Having the ESP32 running the new sketch, in your smartphone open your Wi-Fi settings and tap the **ESP32-Robot** Access Point:



Enter the password **123456789** and tap the **CONNECT** button:



Open your web browser and type the IP address **192.168.4.1**. The web server should load:



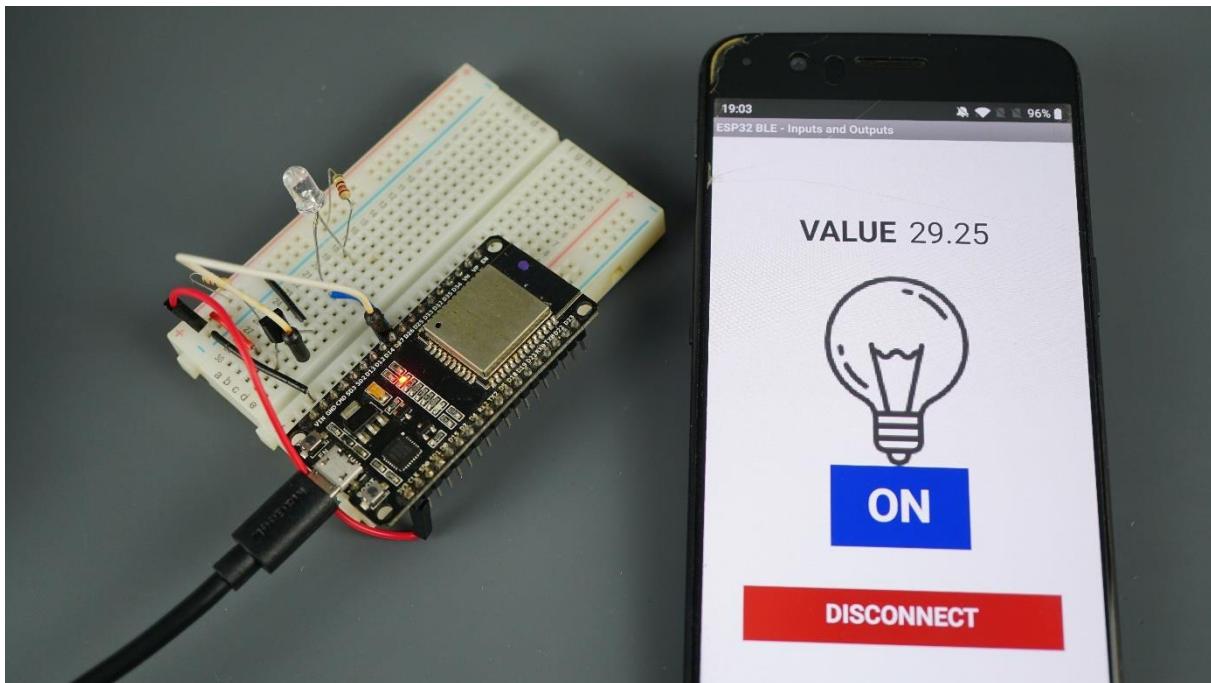
Move the slider to set the speed and then tap the buttons to move the robot. You should be able to control the robot exactly as shown in the previous Units.

PROJECT 3

ESP32 BLE Android Application

Unit 1 - ESP32 BLE Android Application: Control Outputs and Display Sensor Readings

In this project, you'll create an Android application to interact with the ESP32 using Bluetooth Low Energy (BLE). **This project is not compatible with iOS.**



There are three steps in this project:

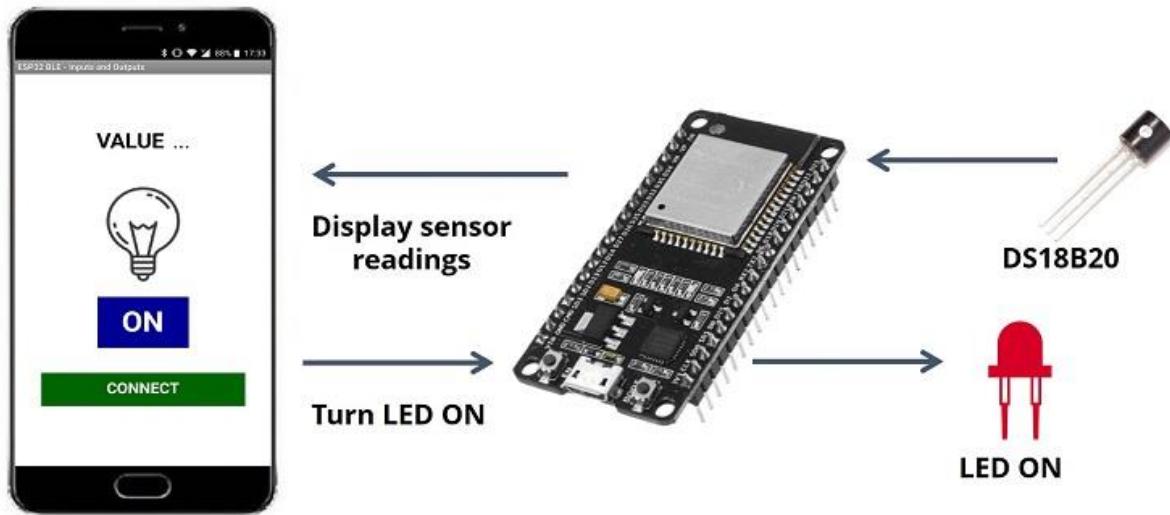
- 1) First, building the circuit;
- 2) Then, uploading the code to the ESP32;
- 3) And finally, installing the Android app provided.

Project Overview

Let's take a look at the project's main features:

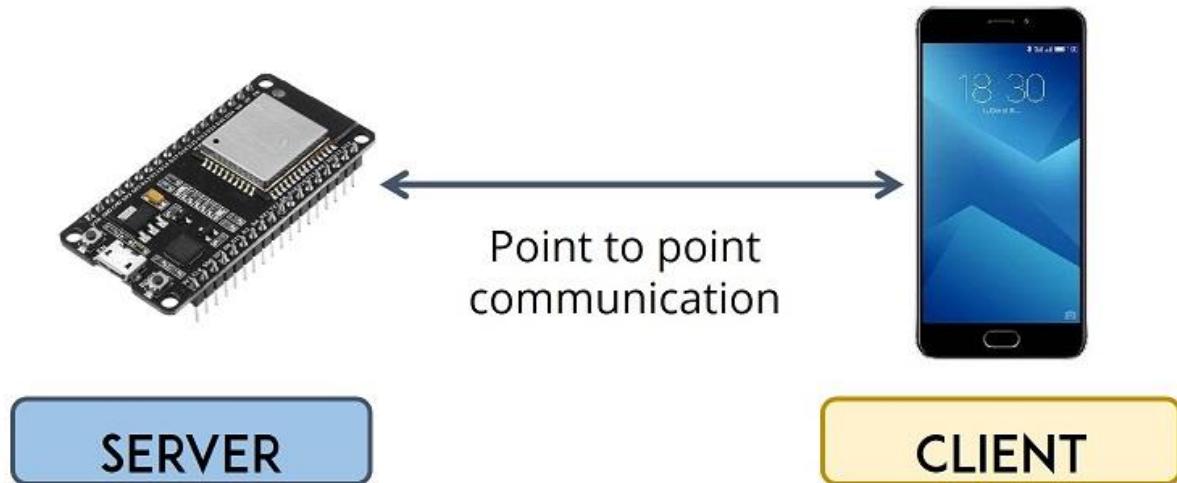
- The Android app controls an ESP32 output and displays sensor readings;
- In this example, we're controlling an LED and reading temperature from the DS18B20 sensor;
- The aim of this project is to create a simple app and explore how you can use BLE with your ESP32. After completing this project, you should be able

to replace the LED with another output and display readings from other sensors.



Here's how this project works:

- The ESP32 is the server, and your smartphone with the Android app is the client, they will be establishing a point-to-point communication;



- The ESP32, as the server, advertises its existence so that other devices can find it, and it contains the data the client can read;
- The client, your smartphone, scans nearby devices to find the ESP32 server;
- In the Android app, you just need to click the "CONNECT" button to search for new devices and select your ESP32.

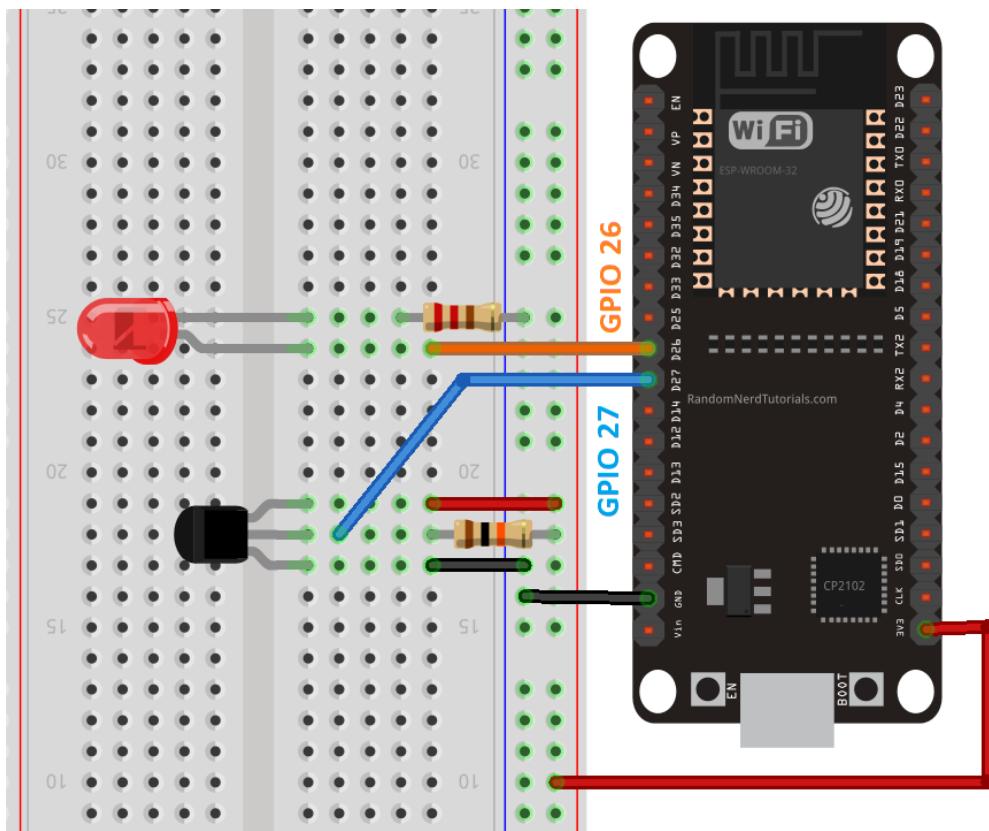
Note: the app for this project was built using MIT App Inventor 2. We won't cover how to build the Android app in this unit, but we provide an additional unit explaining how the app was created.

Wiring the Circuit

Here's a list of the parts needed for this project:

- ESP32 DOIT DEVKIT V1 board
 - Android smartphone with Bluetooth
 - DS18B20 temperature sensor
 - 10K Ohm resistor
 - 5mm LED
 - 220 Ohm resistor
 - Jumper wires

Connect the DS18B20 temperature sensor as shown in this schematic diagram, with the data pin connected to GPIO 27 and the LED to GPIO 26.



Installing the Required Libraries

You need to install the [One Wire library by Paul Stoffregen](#) and the [Dallas Temperature library](#) to interface with the DS18B20 sensor.

To install these libraries, go to **Sketch > Include Libraries > Manage Libraries** and search for the libraries' names. Then, install the libraries.

Uploading the code

After installing the required libraries, copy the following code to your Arduino IDE.

- [Click here to download the code.](#)

```
// Include necessary libraries
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <OneWire.h>
#include <DallasTemperature.h>

// DO NOT CHANGE THE NEXT UUIDS
// Otherwise, you also need to modify the Android application used in this project
#define SERVICE_UUID          "C6FBDD3C-7123-4C9E-86AB-005F1A7EDA01"
#define CHARACTERISTIC_UUID_RX "B88E098B-E464-4B54-B827-79EB2B150A9F"
#define CHARACTERISTIC_UUID_TX "D769FACF-A4DA-47BA-9253-65359EE480FB"

// Data wire is plugged into ESP32 GPIO
#define ONE_WIRE_BUS 27
// Setup a oneWire instance to communicate with any OneWire devices
OneWire oneWire(ONE_WIRE_BUS);
// Pass our oneWire reference to Dallas Temperature
DallasTemperature sensors(&oneWire);

BLECharacteristic *pCharacteristic;
bool deviceConnected = false;

// Temperature Sensor variable
float temperature = 0;
const int ledPin = 26;

// Setup callbacks onConnect and onDisconnect
class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
        deviceConnected = true;
    };
    void onDisconnect(BLEServer* pServer) {
        deviceConnected = false;
    }
};

// Setup callback when new value is received (from the Android application)
class MyCallbacks: public BLECharacteristicCallbacks {
```

```

void onWrite(BLECharacteristic *pCharacteristic) {
    String rxValue = pCharacteristic->getValue();
    if(rxValue.length() > 0) {
        Serial.print("Received value: ");
        for(int i = 0; i < rxValue.length(); i++) {
            Serial.print(rxValue[i]);
        }
        // Turn the LED ON or OFF according to the command received
        if(rxValue.indexOf("ON") >= 0) {
            Serial.println(" - LED ON");
            digitalWrite(ledPin, HIGH);
        }
        else if(rxValue.indexOf("OFF") >= 0) {
            Serial.println(" - LED OFF");
            digitalWrite(ledPin, LOW);
        }
    }
}

void setup() {
    Serial.begin(115200);
    pinMode(ledPin, OUTPUT);
    sensors.begin();

    // Create the BLE Device
    BLEDevice::init("ESP32_Board");

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

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

    // Create a BLE Characteristic
    pCharacteristic = pService->createCharacteristic(
        CHARACTERISTIC_UUID_TX,
        BLECharacteristic::PROPERTY_NOTIFY);

    pCharacteristic->addDescriptor(new BLE2902());

    BLECharacteristic *pCharacteristic = pService->createCharacteristic(
        CHARACTERISTIC_UUID_RX,
        BLECharacteristic::PROPERTY_WRITE);

    pCharacteristic->setCallbacks(new MyCallbacks());

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

    // Start advertising
    pServer->getAdvertising()->start();
    Serial.println("Waiting to connect...");
}

void loop() {
    // When the device is connected
    if(deviceConnected) {
        // Measure temperature
        sensors.requestTemperatures();
    }
}

```

```

// Temperature in Celsius
temperature = sensors.getTempCByIndex(0);
// Uncomment the next line to set temperature in Fahrenheit
// (and comment the previous temperature line)
//temperature = sensors.getTempFByIndex(0); // Temperature in Fahrenheit

// Convert the value to a char array
char txString[8];
dtostrf(temperature, 1, 2, txString);

// Set new characteristic value
pCharacteristic->setValue(txString);
// Send the value to the Android application
pCharacteristic->notify();
Serial.print("Sent value: ");
Serial.println(txString);
}
delay(5000);
}

```

How Does the Code Work?

Let's take a look at the code and see how the ESP32 BLE server works.

Including libraries

First, you need to include the necessary libraries to set the ESP32 as a BLE server and to use the DS18B20 temperature sensor.

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <OneWire.h>
#include <DallasTemperature.h>
```

Defining UUIDs

Then, define the UUIDs for the service, Receiver characteristic (RX) and Transmitter characteristic (TX).

```
#define SERVICE_UUID          "C6FBDD3C-7123-4C9E-86AB-005F1A7EDA01"
#define CHARACTERISTIC_UUID_RX "B88E098B-E464-4B54-B827-79EB2B150A9F"
#define CHARACTERISTIC_UUID_TX "D769FACF-A4DA-47BA-9253-65359EE480FB"
```

Important: do not change these UUIDs, otherwise you also need to modify the Android application used in this project, so they can establish a connection.

Temperature sensor and variables

Define a pin to read data from the temperature sensor. In this case, it's GPIO 27.

```
#define ONE_WIRE_BUS 27
```

Create an instance to communicate with a `oneWire` device.

```
OneWire oneWire(ONE_WIRE_BUS);
```

Then, pass the `oneWire` reference to a `sensors` object.

```
DallasTemperature sensors(&oneWire);
```

The following line creates a pointer to a `BLECharacteristic`.

```
BLECharacteristic *pCharacteristic;
```

Create a boolean variable to control if the device is connected or not.

```
bool deviceConnected = false;
```

Then, create an auxiliary variable to save the temperature sent to the client, in this case to the Android application.

```
float float temperature = 0;
```

Finally, set the `ledPin` to GPIO 26.

```
const int ledPin = 26;
```

setup()

Now, scroll down to the `setup()`. Initialize the serial port at a baud rate of 115200, set the `ledPin` as an `OUTPUT`, and begin the communication with the temperature sensor.

```
void setup() {
    Serial.begin(115200);
    pinMode(ledPin, OUTPUT);
    sensors.begin();
```

BLE device

Create a new BLE device with a name that allows you to identify your ESP32 board, for example `ESP32_Board`.

```
BLEDevice::init("ESP32_Board");
```

BLE server

Create a BLE server.

```
BLEServer *pServer = BLEDevice::createServer();
```

Set callback for the server.

```
pServer->setCallbacks(new MyServerCallbacks());
```

And for the characteristic.

```
pCharacteristic->setCallbacks(new MyCallbacks());
```

Basically, upon a successful connection with a client, it calls the `onConnect()` function and changes the `deviceConnected` boolean variable to `true`.

```
void onConnect(BLEServer* pServer) {
    deviceConnected = true;
};
```

When the client gets disconnected, it calls the `onDisconnect()` function that sets the `deviceConnected` boolean variable to `false`.

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

BLE service

Getting back to the `setup()`... Create a service with the UUID you've defined earlier.

```
BLEService *pService = pServer->createService(SERVICE_UUID);
```

BLE characteristics

And two characteristics for that service.

```
pCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_UUID_TX,
    BLECharacteristic::PROPERTY_NOTIFY);
```

```
BLECharacteristic *pCharacteristic = pService->createCharacteristic(  
    CHARACTERISTIC_UUID_RX,  
    BLECharacteristic::PROPERTY_WRITE);
```

The TX characteristic is responsible for sending values to the client. In this example, it will notify the client with new temperature values every 5 seconds.

The second characteristic is the RX characteristic, which is responsible for receiving new values from the client. In this project it receives the on and off commands to control the output. This characteristic has the WRITE property enabled and we've assigned the `onWrite()` callback function.

```
void onWrite(BLECharacteristic *pCharacteristic) {  
    String rxValue = pCharacteristic->getValue();  
    if(rxValue.length() > 0) {  
        Serial.print("Received value: ");  
        for(int i = 0; i < rxValue.length(); i++) {  
            Serial.print(rxValue[i]);  
        }  
        // Turn the LED ON or OFF according to the command received  
        if(rxValue.indexOf("ON") >= 0) {  
            Serial.println(" - LED ON");  
            digitalWrite(ledPin, HIGH);  
        }  
        else if(rxValue.indexOf("OFF") >= 0) {  
            Serial.println(" - LED OFF");  
            digitalWrite(ledPin, LOW);  
        }  
    }  
}
```

When the server receives a new value, it calls the `onWrite()` function and writes a new value in the server's characteristic. That value is stored in the `rxValue` variable. Then, according to the value received, whether it's an ON command or an OFF command, it turns the LED on or off.

```
// Turn the LED ON or OFF according to the command received  
if(rxValue.indexOf("ON") >= 0) {  
    Serial.println(" - LED ON");  
    digitalWrite(ledPin, HIGH);  
}  
else if(rxValue.indexOf("OFF") >= 0) {  
    Serial.println(" - LED OFF");  
    digitalWrite(ledPin, LOW);  
}
```

Start BLE

After creating the server and service, and assigning the characteristics, start the BLE service:

```
pService->start();
```

And start the advertising, so that a client can find the ESP32 server.

```
pServer->getAdvertising()->start();
```

loop()

In every `loop()`, we constantly check if the device is connected or not. If the device is connected, the following `if` statement is true. So, it will take a new sensor reading and save it in the `temperature` variable.

```
if(deviceConnected) {  
    // Measure temperature  
    sensors.requestTemperatures();  
  
    // Temperature in Celsius  
    temperature = sensors.getTempCByIndex(0);  
    // Uncomment the next line to set temperature in Fahrenheit  
    // (and comment the previous temperature line)  
    //temperature = sensors.getTempFByIndex(0); // Temperature in Fahrenheit  
  
    // Convert the value to a char array  
    char txString[8];  
    dtostrf(temperature, 1, 2, txString);  
  
    // Set new characteristic value  
    pCharacteristic->setValue(txString);  
    // Send the value to the Android application  
    pCharacteristic->notify();  
    Serial.print("Sent value: ");  
    Serial.println(txString);  
}
```

By default, it's sending the temperature in Celsius degrees, you can comment this next line:

```
temperature = sensors.getTempCByIndex(0);
```

And uncomment the next one to send the temperature in Fahrenheit degrees.

```
//temperature = sensors.getTempFByIndex(0); // Temperature in Fahrenheit
```

Finally, before sending the value to the connected client, you need to convert the float variable to a char array using the `dtostrf()` function.

```
dtostrf(temperature, 1, 2, txString);
```

Then, set the characteristics with the new value and notify the client.

```
// Set new characteristic value
pCharacteristic->setValue(txString);
// Send the value to the Android application
pCharacteristic->notify();
```

This process is repeated every 5 seconds.

```
delay(5000);
```

In summary, when a client is connected to the ESP32 server, it sends a new temperature reading every 5 seconds. When a client writes on the RX characteristic the ON or OFF commands, it controls the LED.

Uploading the Code

You can now upload the code to your ESP32. Make sure you have the right board and COM port selected.

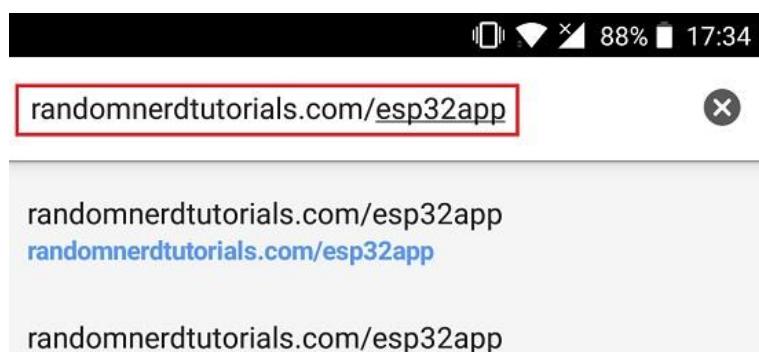
Preparing the Android App

Let's move on to the Android application. As we've mentioned previously, this app was built using [MIT App Inventor 2](#). You can download the .apk and .aia files:

- [Android application .apk file](#)
- [Edit the app with .aia file on MIT App Inventor](#)

To install the app on your smartphone, copy the following link to your smartphone to download the .apk file:

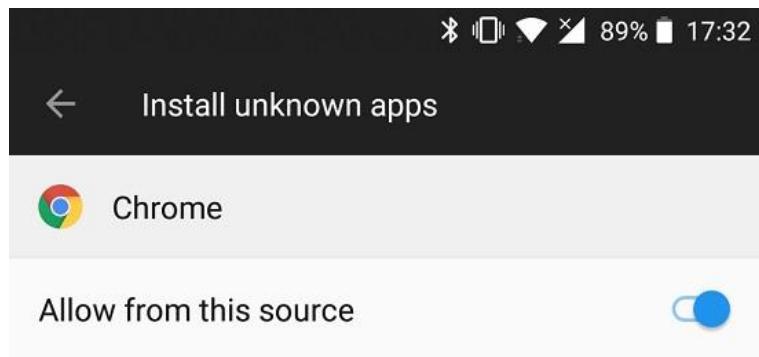
- <https://RandomNerdTutorials.com/esp32app>



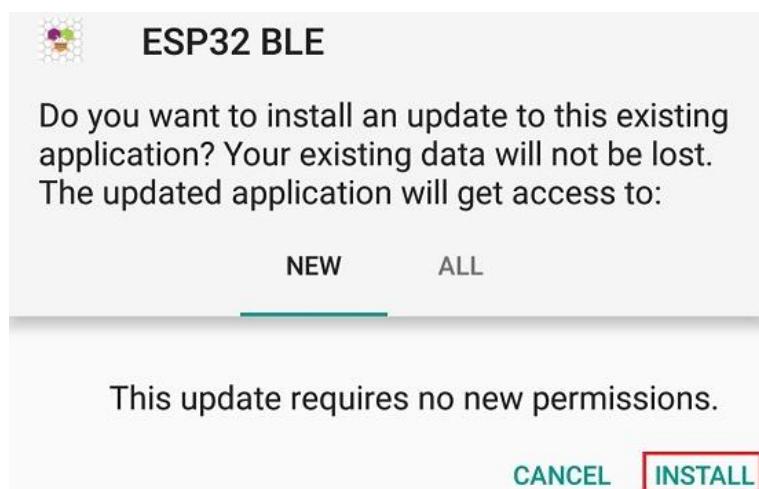
The file is secure, even though you might get a security notice when trying to download.

Alternatively, simply move the *.apk* file downloaded earlier to your smartphone.

In your smartphone, you need to open the downloaded file and allow applications from this source to be installed:



Follow the installation wizard to install the app.



It should take a few seconds to install.

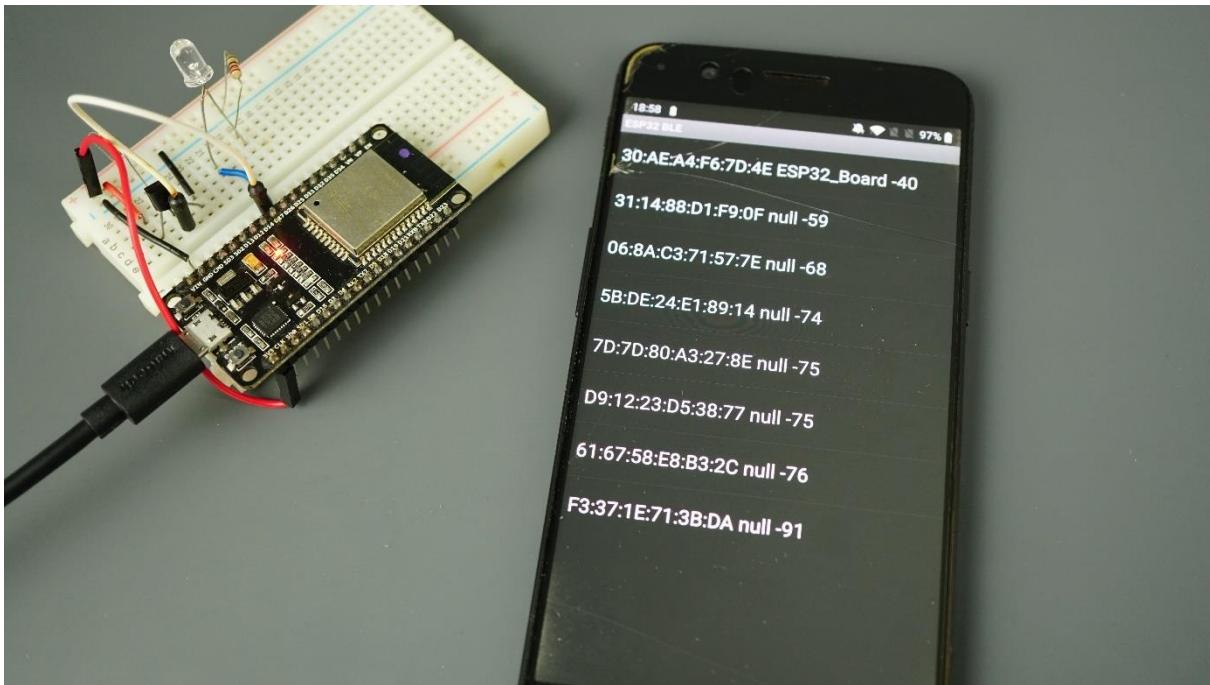
If everything went as expected, the app icon should be on your home screen (or you might have to search for the app "ESP32 BLE").

Demonstration

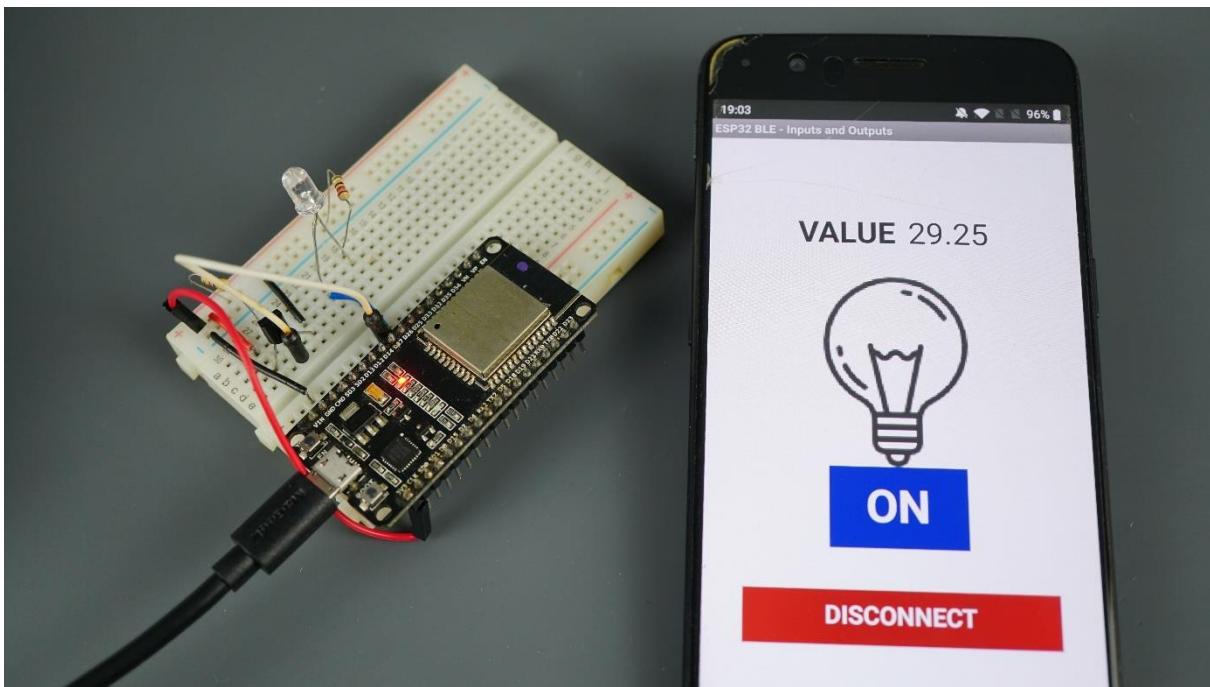
Let's test the project.

Enable Bluetooth on your smartphone. Open the Android app and tap the "CONNECT" button to search for nearby devices. Select the **ESP32_Board** device,

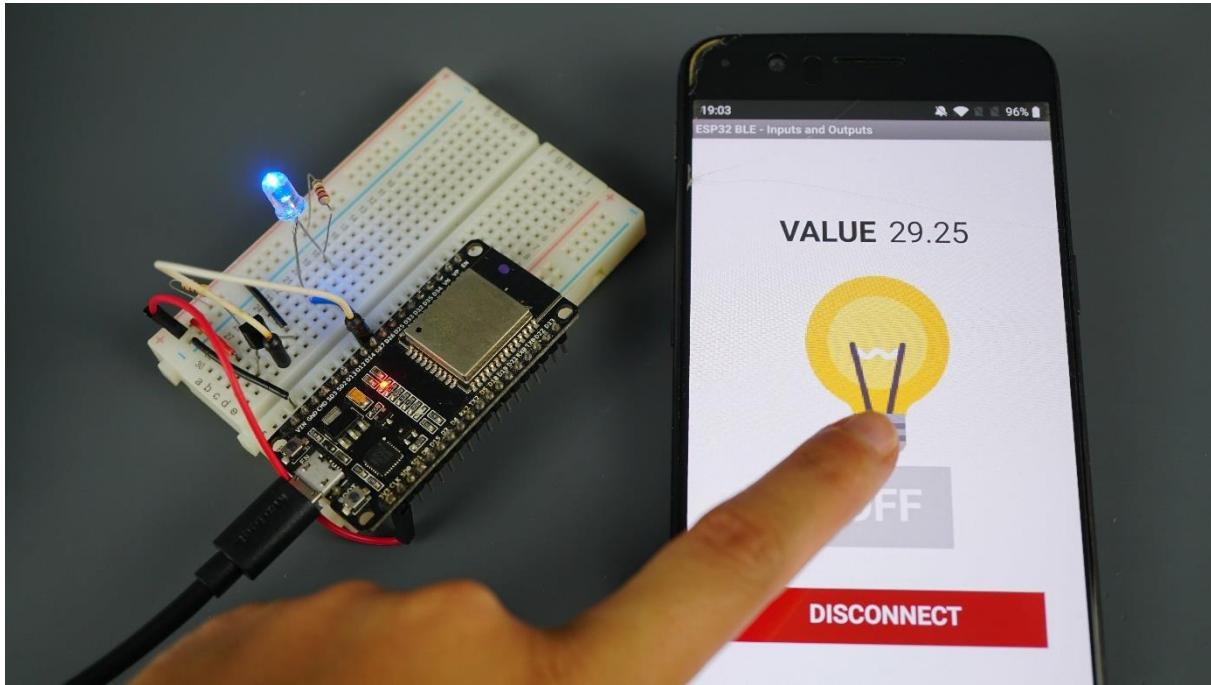
and it should connect within a few seconds. At this point, everything should be ready.



After establishing a successful connection, new temperature readings will be displayed and updated every 5 seconds.



Press the ON button to turn the LED on. There's an image that shows the current LED state. Press the OFF button to turn the LED off. This setup works perfectly and responds instantly to the ON and OFF commands.



Wrapping Up

That's it for this project. You can replace that LED with a relay to control your own electronics appliances. You can also replace the DS18B20 with another sensor that best suits your needs.

If you want to learn how the app works, read the next Unit.

Note: if you'd like to learn more about Android apps, we have a dedicated course on that topic: [Android Apps for Arduino with MIT App Inventor 2](#).

To learn more about this subject and get you familiar with the MIT App Inventor software, you can take a look at our [Getting Started Guide](#).

Unit 2 - Bluetooth Low Energy (BLE)

Android Application with MIT App Inventor 2: How the App Works?

This Unit shows how the Android Application for the ESP32 BLE project was created, so that you can modify it yourself.

Resources:

- To edit the Android app, you need to download the .aia file and import it to MIT App Inventor: [ESP32_BLE_Inputs_and_Outputs.aia](#)
- You can download the .apk file to install the Android app: [ESP32_BLE_Inputs_and_Outputs.apk](#)
- There's also a .zip folder with all the resources for this project: [Project folder for ESP32 BLE Inputs and Outputs](#)

Introducing MIT App Inventor 2

The application for the ESP32 BLE project was created using MIT App Inventor 2. You don't need to download or install anything on your computer because the software is cloud-based. So, you build the apps directly in your browser (Chrome, Mozilla, Safari, Internet Explorer, etc). You only need an internet connection to build the apps.



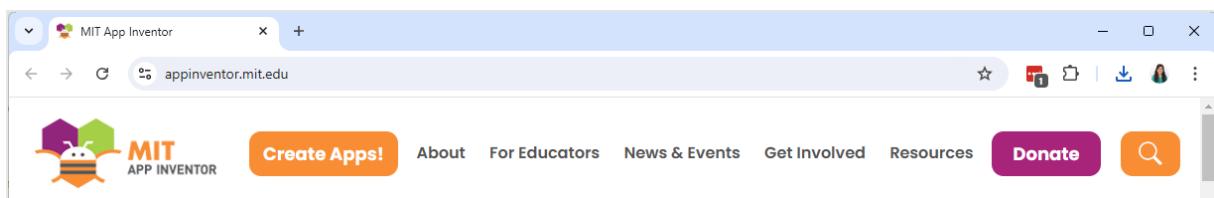
Why MIT App Inventor 2?

MIT App Inventor 2 is a simple and intuitive free service for creating Android applications. You don't have to be an expert in programming or design to build awesome and useful apps.

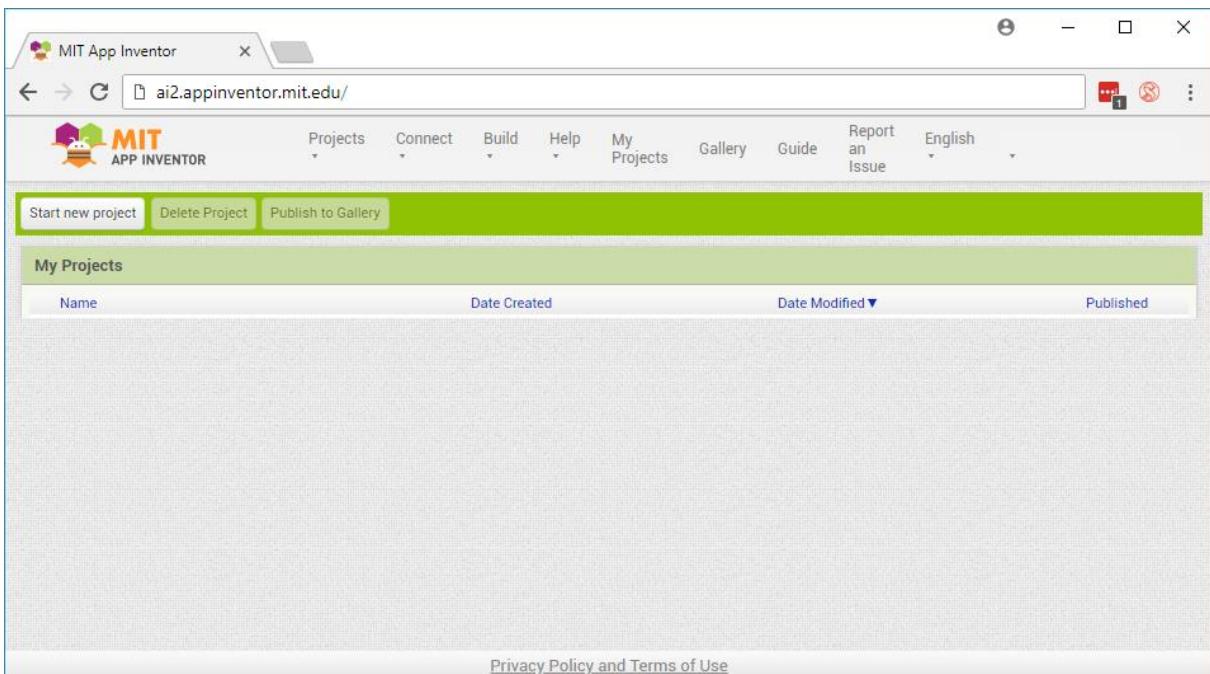
Creating the design is as easy as selecting and placing components on the smartphone screen. The coding is done with drag and drop puzzle blocks. Anyone can learn how to build their own apps with MIT App Inventor 2 with a few hours of practice.

Accessing MIT App Inventor 2

To access MIT App Inventor 2 go to <http://appinventor.mit.edu/explore> and press the orange "Create apps!" button.

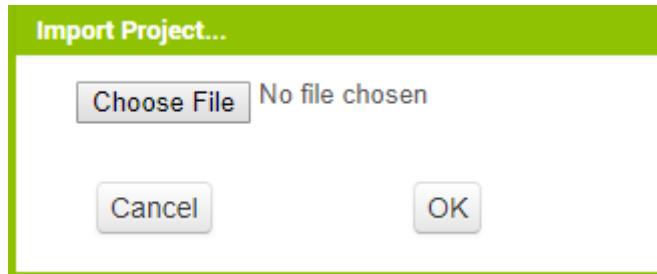


To access the app builder, you need a Google account. Follow the on-screen steps to login into MIT App Inventor 2. After that, you'll be presented with the following dashboard:



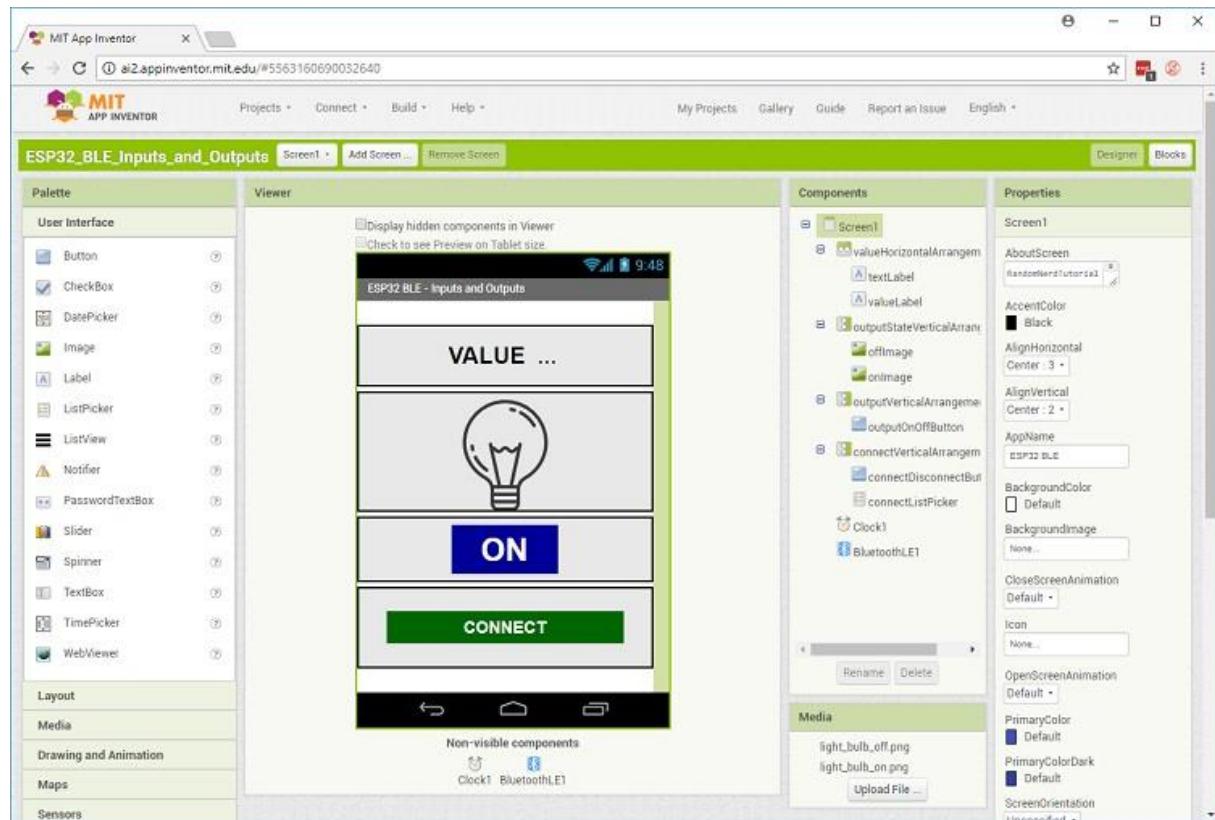
Importing the Android App File

In that dashboard, at the top menu, select **Projects** and go to **Import project (.aia)** from my computer. Click on “**Choose File**” and select the *.aia* file provided.



- [Click here to download the .aia file](#)

After successfully importing the *.aia* file, the next page should load.



This is the **Designer** tab. The Designer tab gives you the ability to add buttons, text, images, sliders, etc... It allows you to change the overall app look. In the Designer tab, you have several sections:

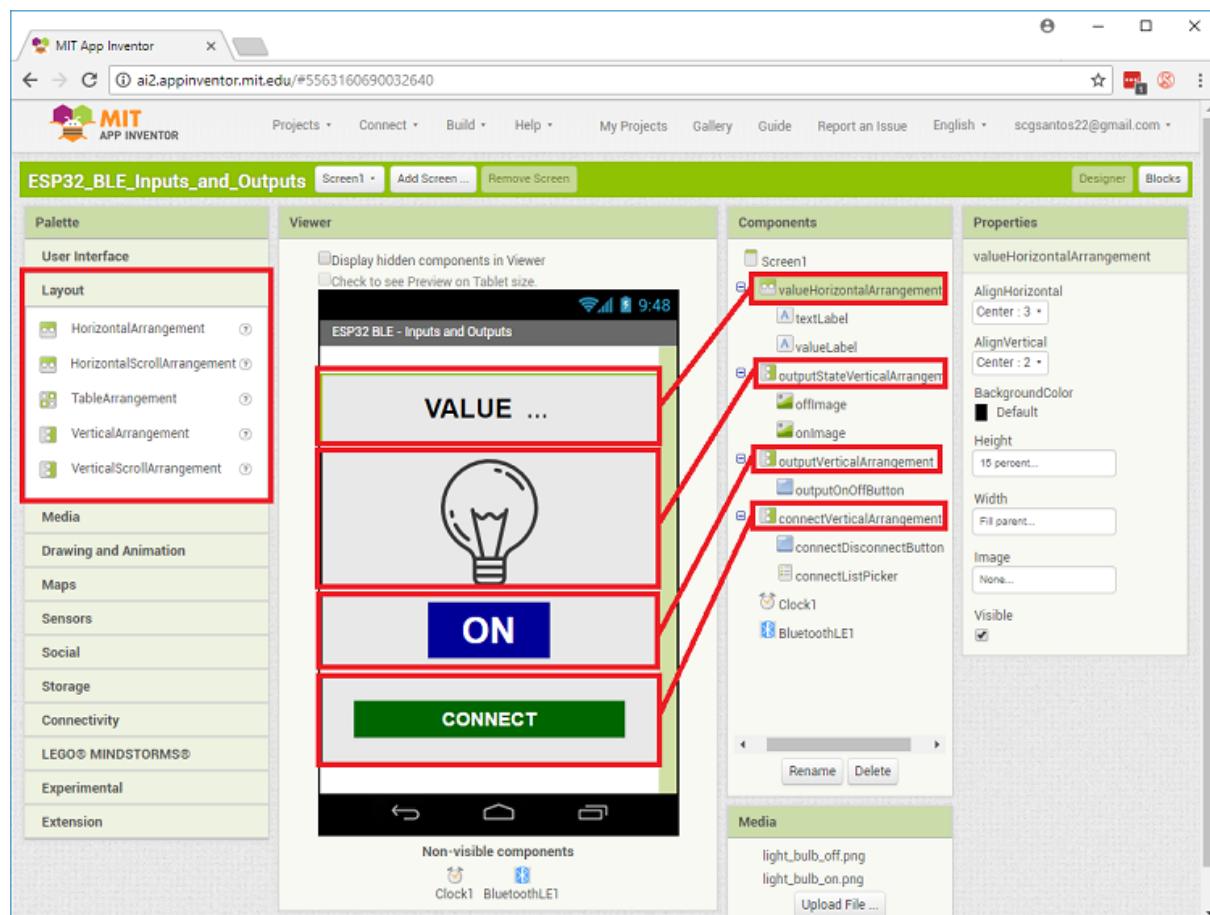
- **Palette:** contains the components to build the app design like buttons, sliders, images, labels, etc...
- **Viewer:** this is where you drag the components to build the app look.

- **Components:** shows the different components added to the app and how they are organized hierarchically.
- **Media:** this part shows the imported media like images or sounds you want to add to your application.
- **Properties:** this is where you select your components' properties like color, size and orientation.

Exploring the App Design

The components you add to the app design should be placed inside arrangements. You can add new arrangements to your app in the Layout under the Palette section.

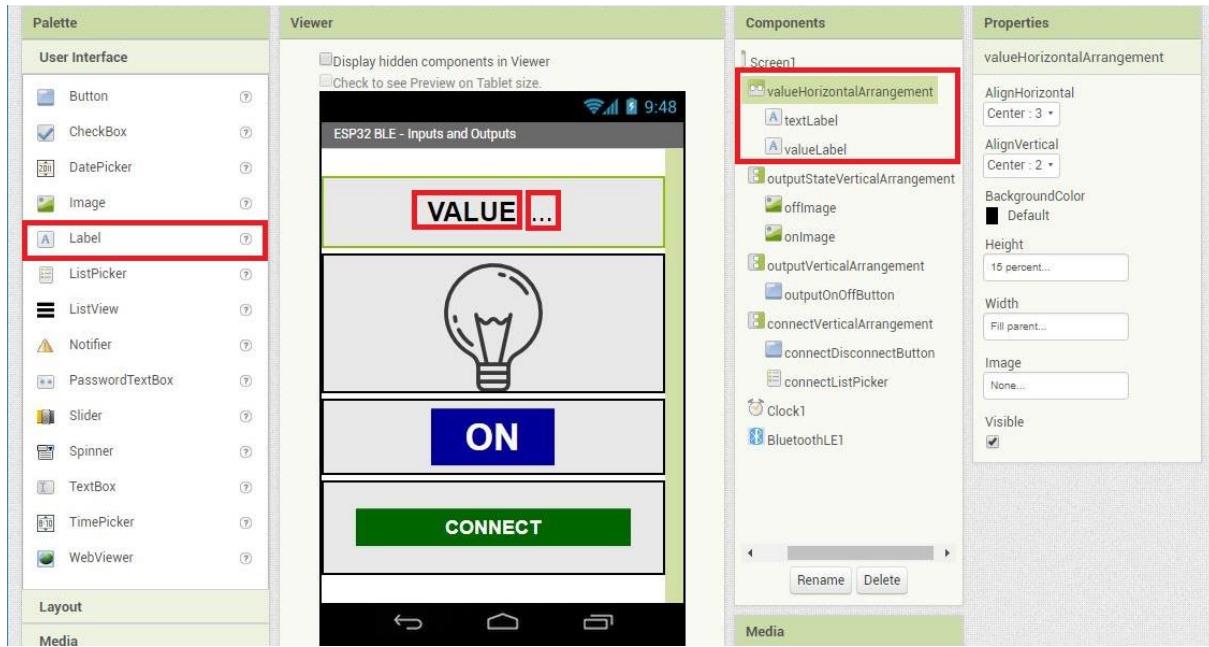
There are several arrangements to place the components. The image below shows the arrangements for each app component.



Now let's take a look at each component.

Text Labels

The first arrangement contains two text labels.



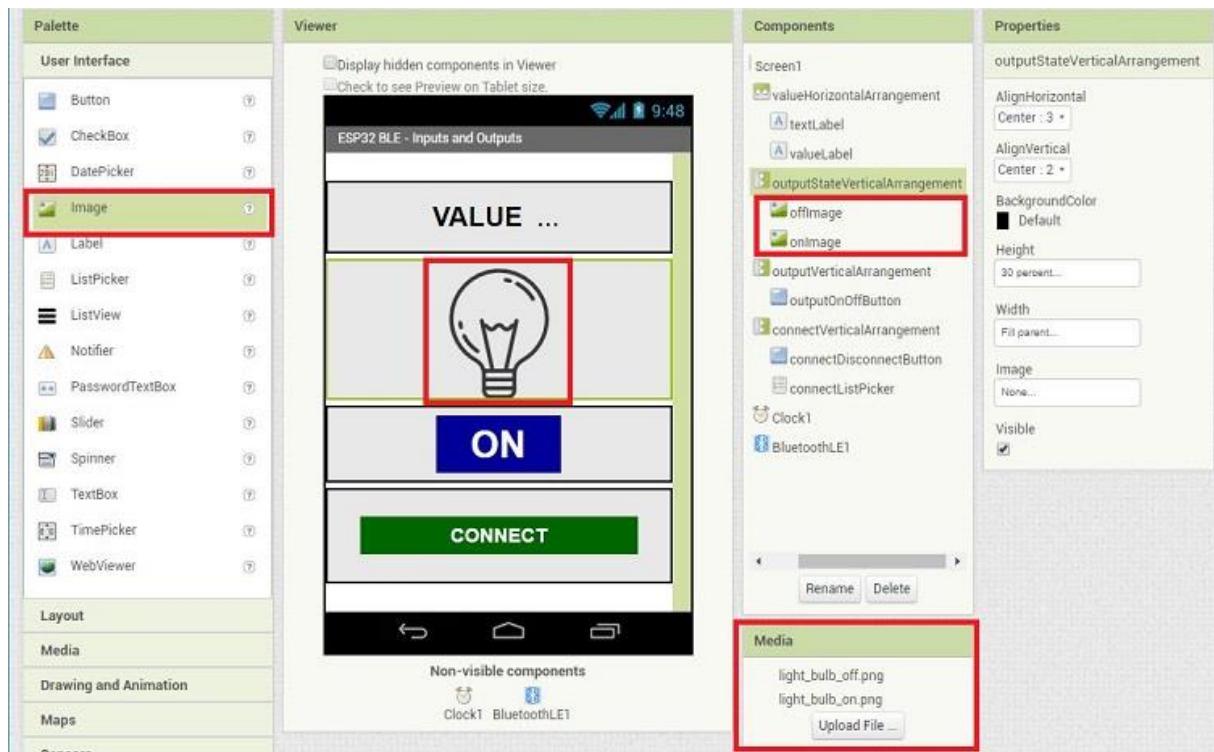
One text label contains the text “VALUE”. We’ve defined the text as “VALUE” because instead of temperature you may want to display any other sensor readings. Modify that text to match the measurement you’re displaying.

The other label will display the readings—at the moment it displays “...”. If you wanted to display more readings you would need to place more Label components into the arrangement.

To add more labels, you just need to drag the Label component under Palette > User Interface to the viewer as highlighted in the figure above. When you select the text label, at the right, under the Properties section, you can edit the text appearance.

Image

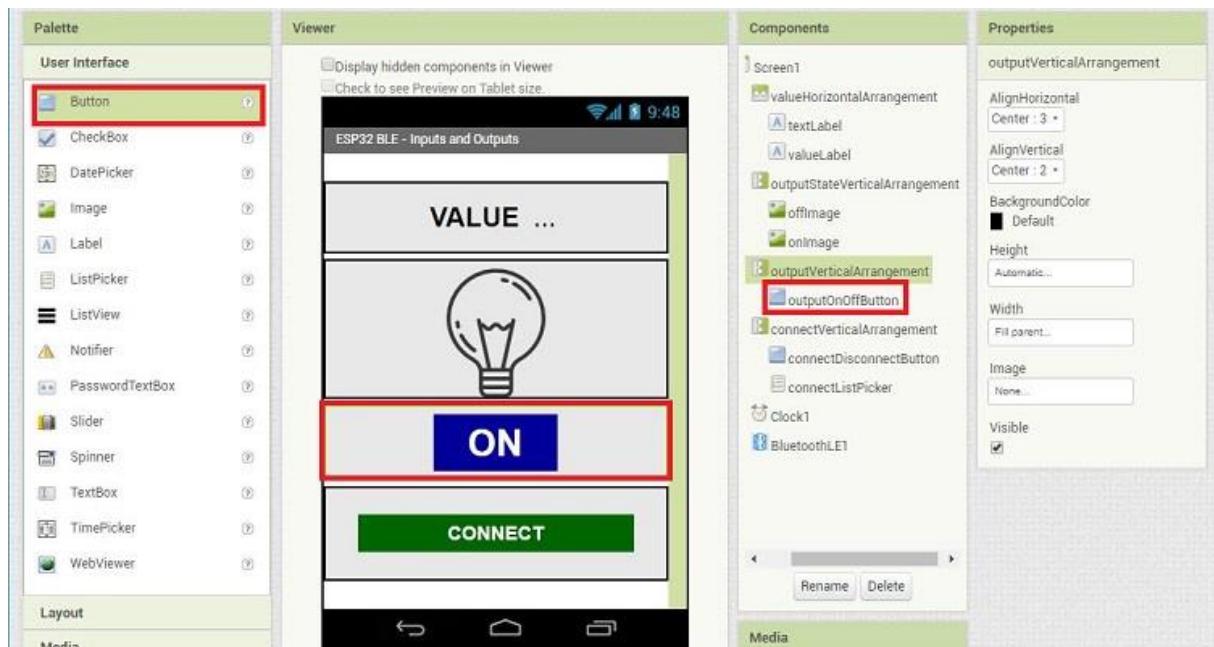
The next arrangement contains the lamp image.



To add an image to the viewer, you drag the Image component under **Palette** > **User Interface**. In this case, we have two Image fields, but only one is showing up – you can set that in the Image properties. To use your images in the app you need to upload the images using the Media section.

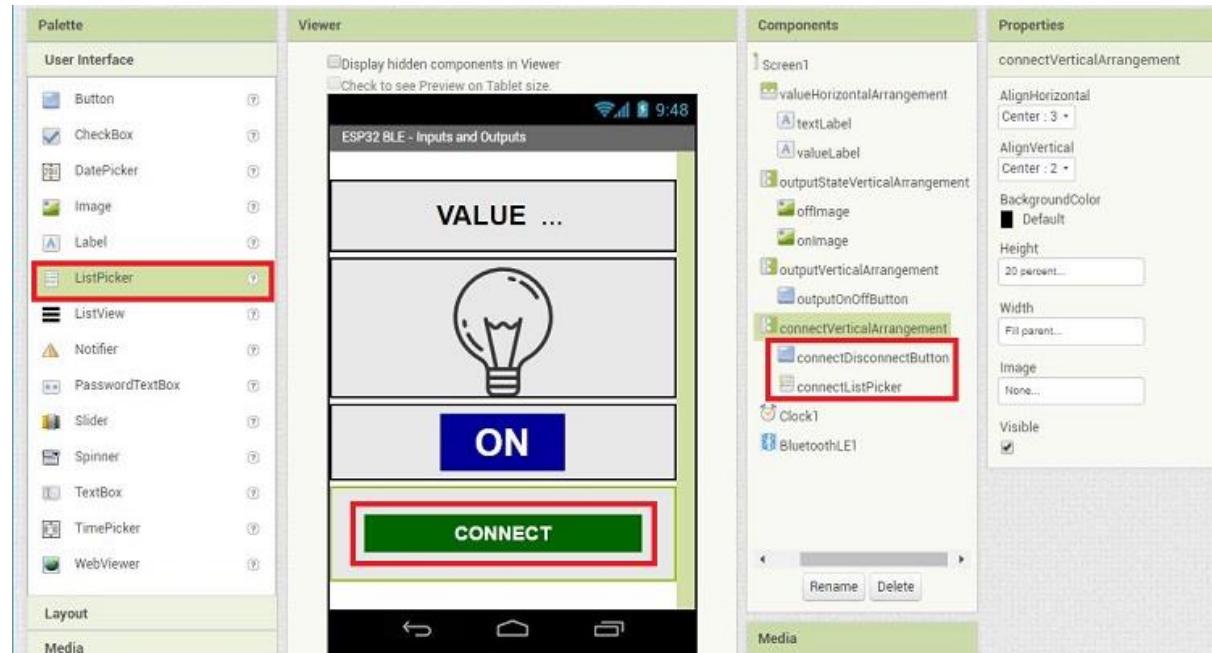
Button

The following arrangement contains the button to send the ON or OFF commands to the ESP32 to control the output.



Adding a button to the app is as simple as dragging the Button to the Viewer. Then, you can edit the button properties under the Properties section. At the moment, we're displaying the ON button. To display the OFF button, we'll change the button color and text in the Blocks tab. If you want to add other commands to control the ESP32, you can add new buttons.

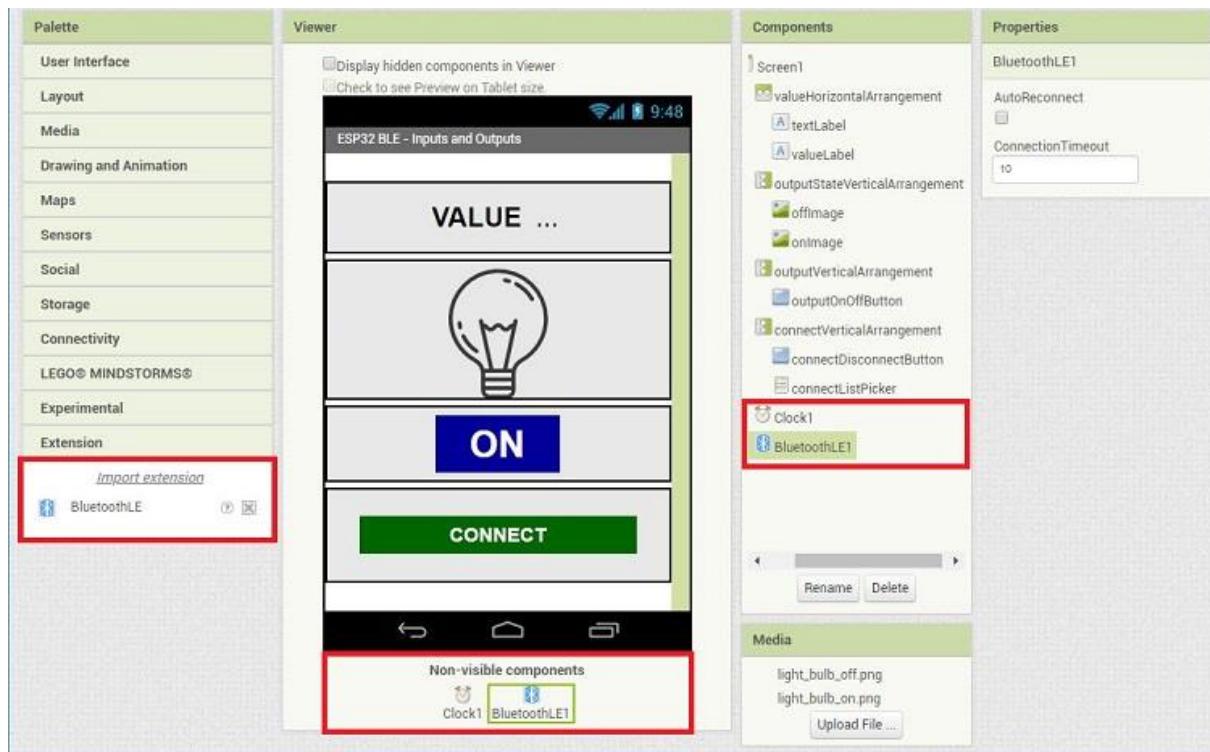
ListPicker



The last arrangement contains a button and a listPicker. The listPicker is not visible in the Viewer but it is called when you click the “CONNECT” button. The listPicker is an essential component in Bluetooth apps to list the Bluetooth devices that were found.

Non-Visible Components

Finally, in the design, you need to add two non-visible components: the Clock and BluetoothLE components.



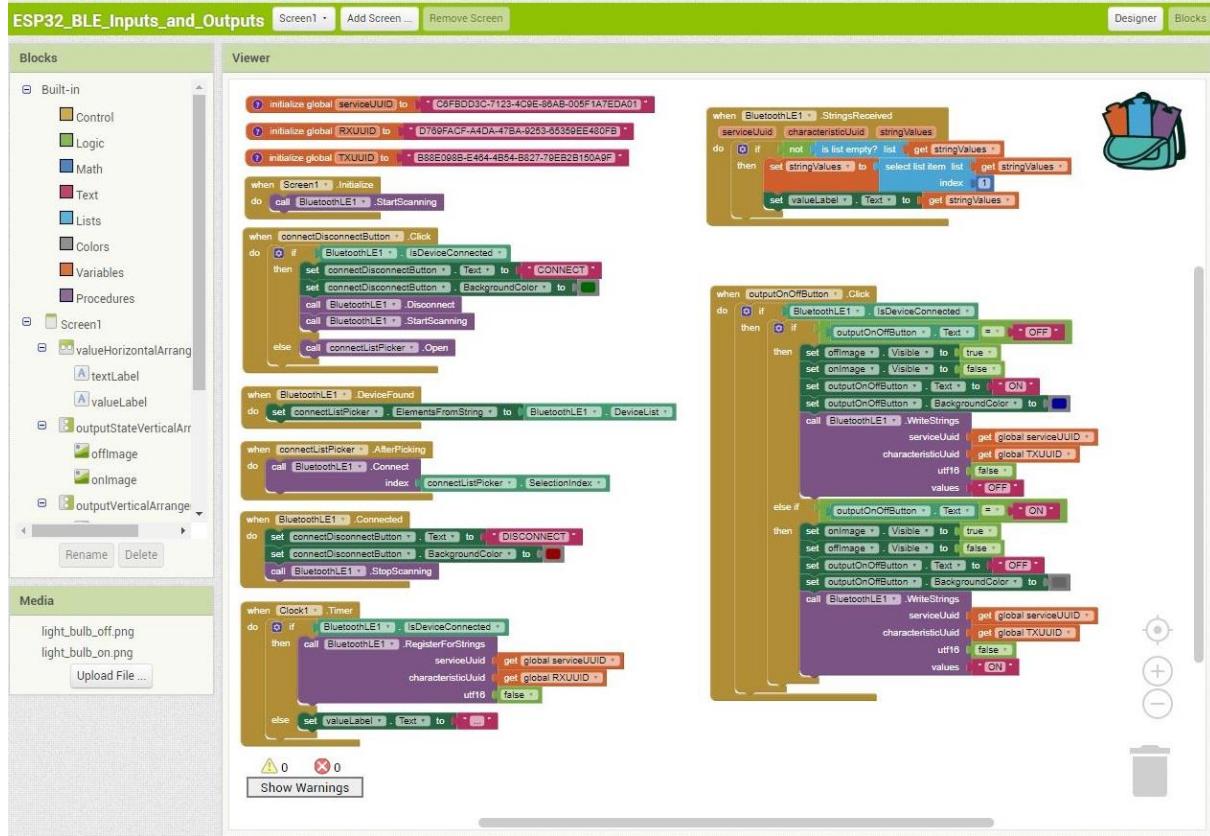
The MIT App Inventor 2 software doesn't support BLE by default. So, the BLE component shows up in the Extension section. Since the imported app contains a BluetoothLE component, it is automatically added to the extensions.

Exploring the App Logic

Now that we've explored the app design, let's look at the logic that makes the app work. For that, in the upright corner click the Blocks tab as shown below.



This new window loads:



This section contains the app logic. This is what allows you to create custom functionality for your app, so when you press the buttons, it actually does something. The logic of the app is built by combining different logic blocks together. This is what makes the app define the buttons' functionalities, search for nearby devices, connect to BLE devices, etc...

Let's explore what each block's section does.

Defining UUIDs

The following blocks define the UUIDs for the service, RX characteristic, and TX characteristic.

```

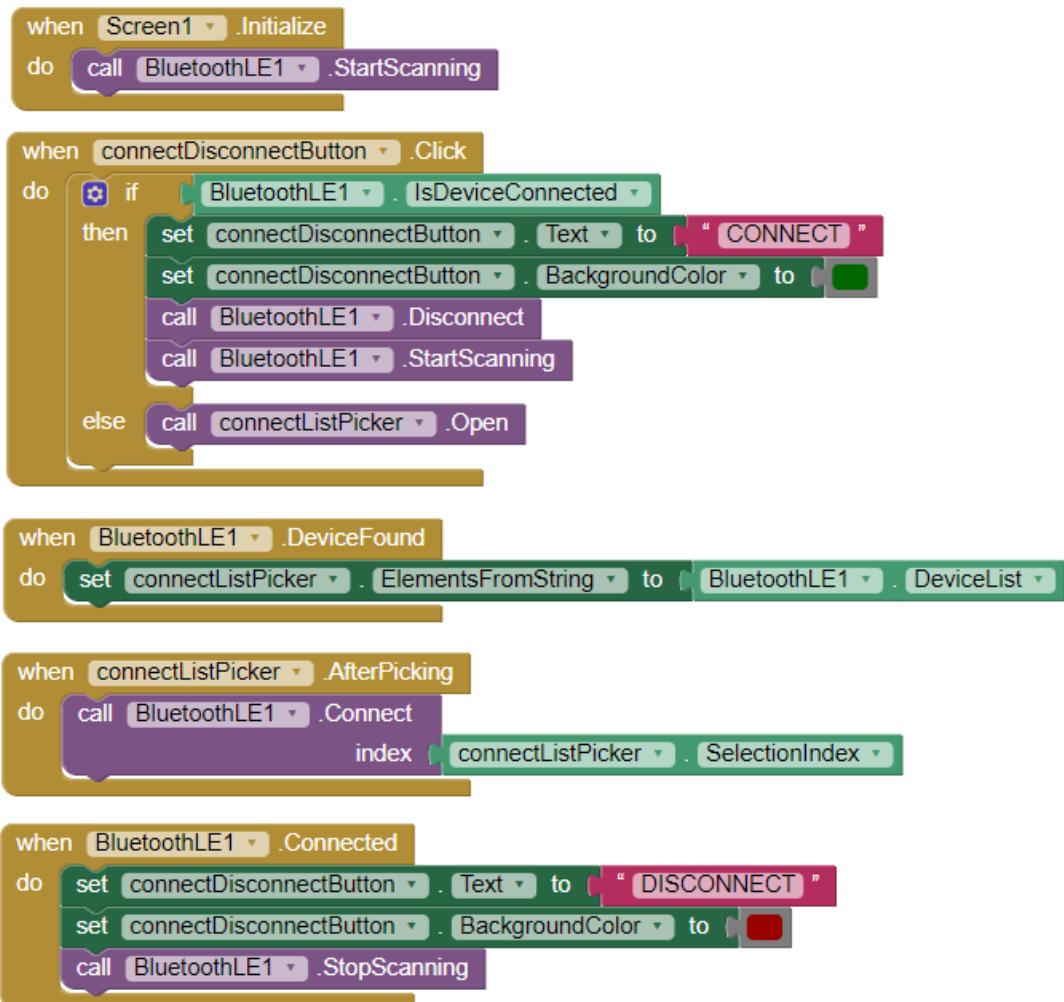
① initialize global [serviceUUID] to "C6FBDD3C-7123-4C9E-86AB-005F1A7EDA01"
② initialize global [RXUUID] to "D769FACF-A4DA-47BA-9253-65359EE480FB"
③ initialize global [TXUUID] to "B88E098B-E464-4B54-B827-79EB2B150A9F"

```

These UUIDs should match the UUIDs used in the Arduino IDE code. If you want to use different UUIDs, you need to modify them in both places. To generate your own UUIDs go to the [UUID generator website](#).

Listing and Connecting to a BLE Device

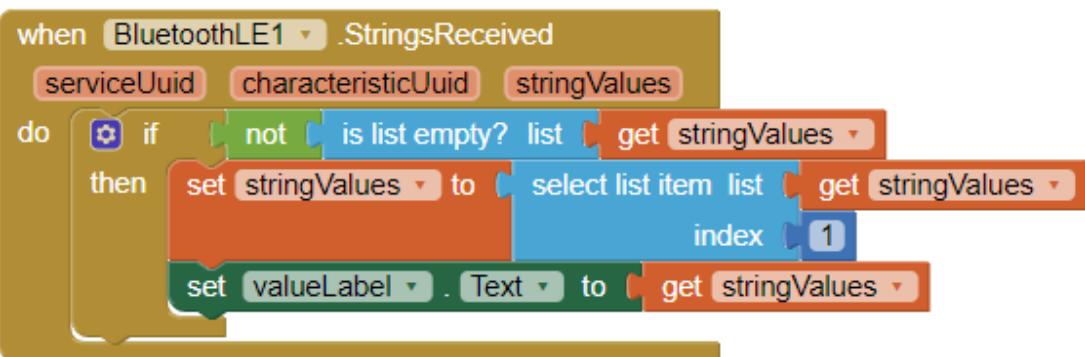
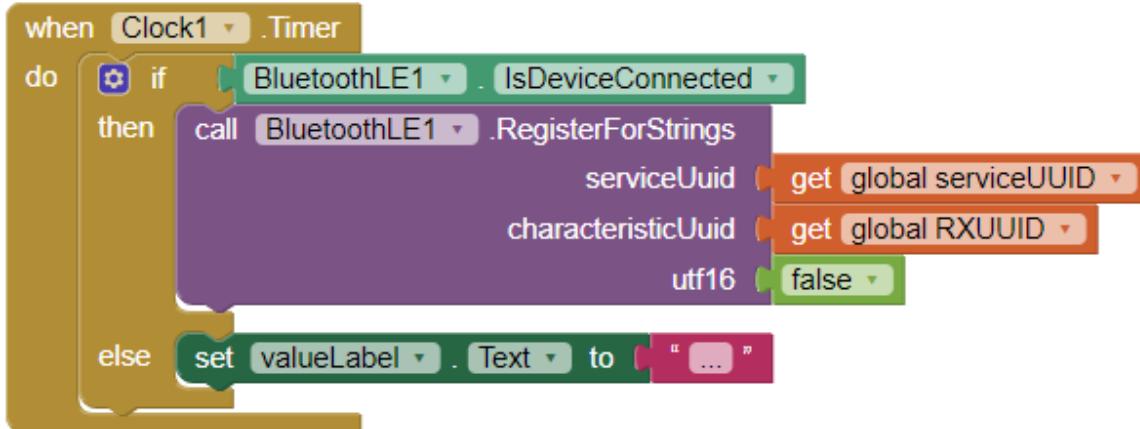
The next set of blocks is responsible for listing, picking, and connecting to a BLE device.



This set also changes the button appearance depending on whether the app is connected to a BLE device or not. We don't recommend changing these blocks, because in your BLE applications you'll always need a way to list and connect to a BLE device. However, there are different ways to do it, for example you can have two buttons—a button to connect and another to disconnect.

Receiving Notifications

The temperature characteristic (RX characteristic) has the *notify* property enabled – you receive new temperature values every 5 seconds via notification. The following two blocks are responsible for displaying the received values.

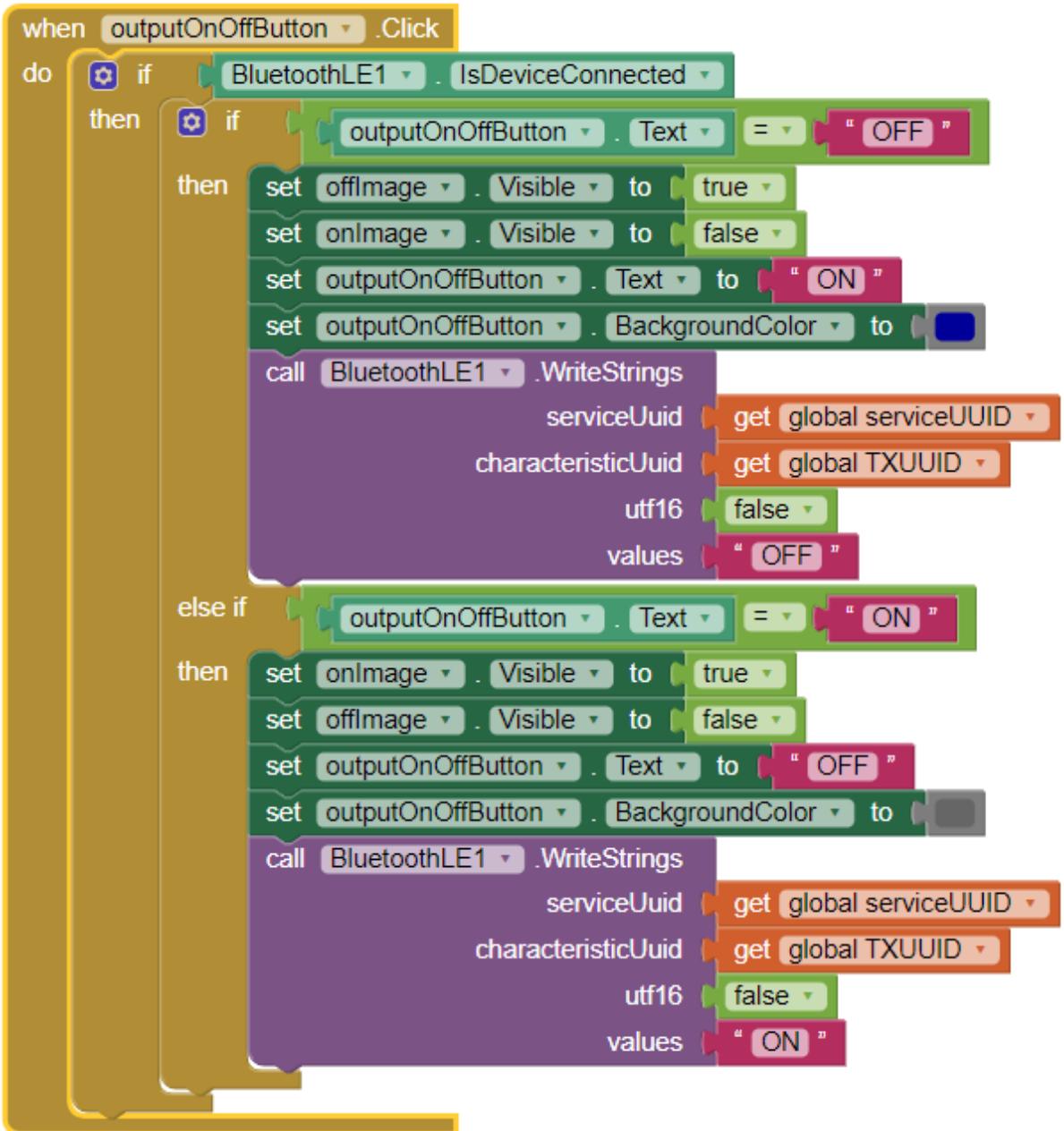


The first block starts the RX characteristic to receive new values or sets the temperature value to "...", if the client gets disconnected. The second block receives the message sent from the ESP32, saves it in the `stringValues` variable, and displays it in the `valueLabel`.

Note: this code block works whether you're sending temperature or any other value. So, you don't need to change any of these blocks even if you're receiving other values from the ESP32.

Sending Commands

The following block sends the "ON" and "OFF" commands to the ESP32 to control the output. It also changes the button text, button color, and lamp state in the app.



Basically, to send commands to the ESP32, you write "ON" or "OFF" in the TX characteristic. Here's what happens when you click the "ON" button:

- You set the image with the lamp ON to visible;
- You set the image with the lamp OFF to invisible;
- You change the button appearance – set text to "OFF" and change color to grey;
- Finally, you write "ON" in the TX characteristic.

If you want to send other commands to control other outputs, you just need to add more buttons to the app layout. Then, depending on the button pressed, write

different messages on the TX characteristic. After that, you just need to add conditions to the Arduino IDE code, to make the ESP32 do what you want.

Useful Tip: Testing the App

You can test the app on your smartphone while editing in real-time. You need to install the **MIT AI2 Companion** app on your smartphone. Go to Google Play Store and search for “MIT AI2 Companion” and install it.



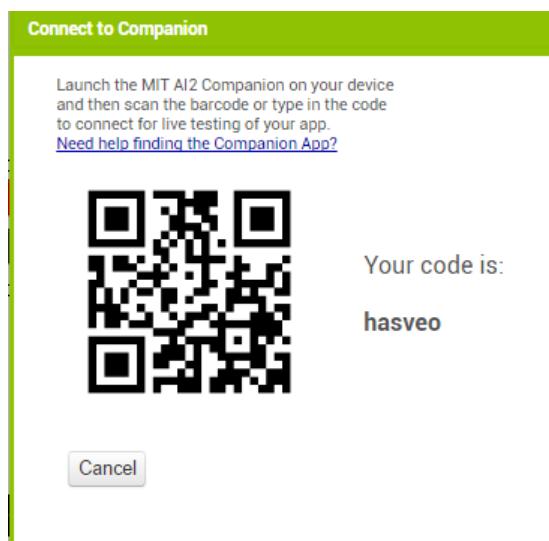
Open the app, you'll be presented with the following screen.



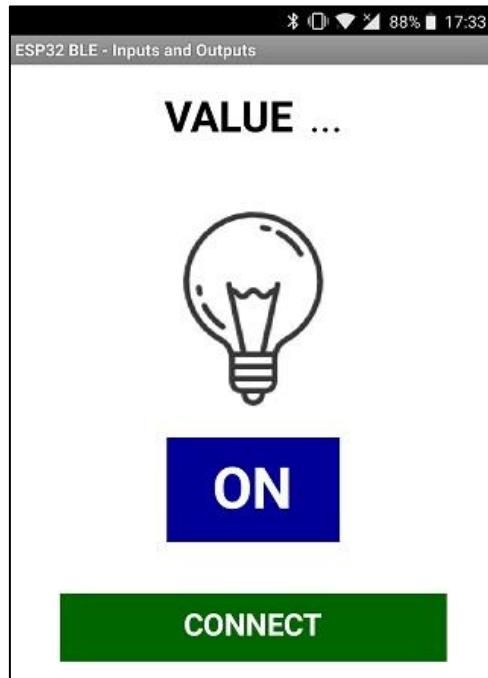
To test your app, in the MIT App Inventor 2 software, go to **Connect ▶ AI Companion**:



A QR code like the one in the following figure pops up:



On your smartphone, you can either enter the code or scan the QR code. After having the code, tap the “**connect with code**” button. Your smartphone shows how your app looks. It updates in real time, so every time you make a change, you can instantly see the result.



Wrapping Up

This was just a glimpse of how the BLE Application for the ESP32 works. We encourage you to modify this app to meet your specific needs – that's the best way to learn how to use the MIT App Inventor software.

If you'd like to know more about MIT App Inventor, we have a dedicated course on how to use MIT App Inventor with Arduino: [Android Apps for Arduino with MIT App Inventor 2](#).

We also recommend taking a look at the [MIT App Inventor website](#) for more resources.

PROJECT 4

**LoRa Long Range Sensor
Monitoring and Data Logging**

Unit 1 - LoRa Long Range Sensor Monitoring and Data Logging

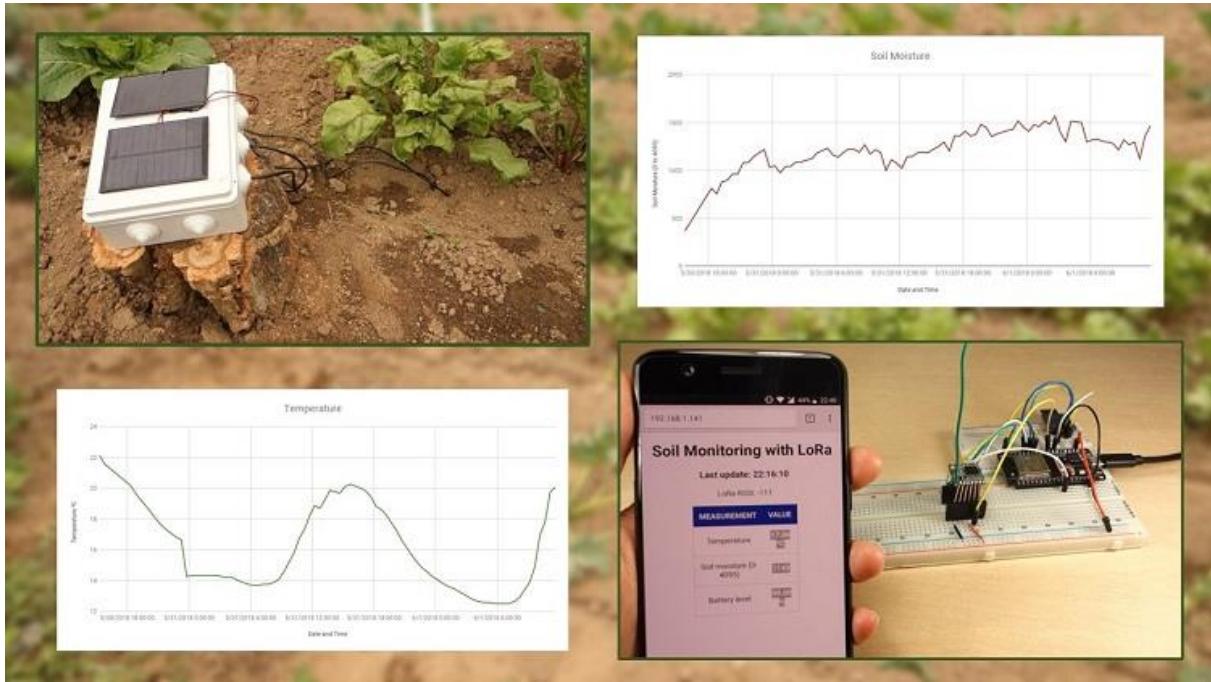
This is Part 1 of the “LoRa Long Range Sensor Monitoring and Data Logging – Reporting Sensor Readings from Outside: Soil Moisture and Temperature” project.



This project is quite long, so it is divided into 5 Parts:

- **Part 1:** Project Overview
- **Part 2:** LoRa Sender
- **Part 3:** LoRa Receiver
- **Part 4:** Solar Powered LoRa Sender
- **Part 5:** Final Tests, Demonstration, and Data Analysis

In this project, you're going to build an off-the-grid monitoring system that sends soil moisture and temperature readings to an indoor receiver. To establish a communication between the sender and the receiver, we'll use LoRa radio. The indoor LoRa receiver displays sensor readings on a web server and logs the data into a microSD card with timestamps.



This project applies several concepts addressed throughout the course. We recommend taking a look at the following Units:

- [**ESP32 – LoRa Sender and Receiver:** Unit 8.2](#)
- [**ESP32 Web Server – Display Sensor Readings:** Unit 6.6](#)
- [**ESP32 Deep Sleep – Timer Wake Up:** Unit 4.2.](#)

Parts Required

Here's a list of all the parts required to complete this project.

LoRa Sender:

- [ESP32 DOIT DEVKIT V1 board](#)
- [RFM95 LoRa transceiver module](#)
- RFM95 LoRa breakout board (optional)
- Temperature sensor:
 - [DS18B20 temperature sensor \(waterproof version\)](#)
 - [10K Ohm resistor](#)
- [Resistive Soil Moisture sensor](#)
- Power source and charger:
 - [Lithium Li-ion battery \(at least 3800mAh capacity\)](#)
 - Battery holder

- [Battery charger \(optional\)](#)
 - [TP4056 Lithium Battery Charger](#)
 - [2x Mini Solar Panel \(5V 1.2W\)](#)
- Battery voltage level monitor: [27K Ohm resistor + 100K Ohm resistor](#)
 - Voltage regulator:
 - [Low-dropout or LDO regulator \(MCP1700-3320E\)](#)
 - [100uF electrolytic capacitor](#)
 - [100nF ceramic capacitor](#)
- [2x Breadboards](#)
- [Jumper Wires](#)
- [Project box enclosure \(IP65/IP67\)](#)

LoRa Receiver:

- [ESP32 DOIT DEVKIT V1 board](#)
- [RFM95 LoRa transceiver module](#)
- RFM95 LoRa breakout board (optional)
- [MicroSD card module](#)
- [MicroSD card](#)
- [2x Breadboards](#)
- [Jumper Wires](#)

Useful tools for this project:

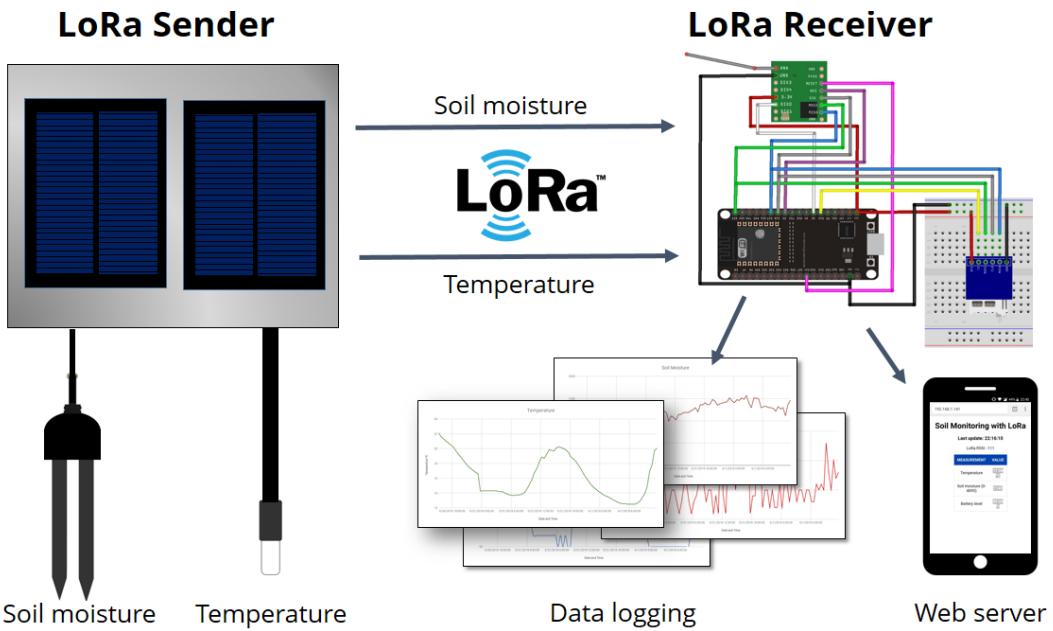
- [Soldering Iron](#)
- [Hot glue gun](#)
- [Multimeter](#)

Project Overview

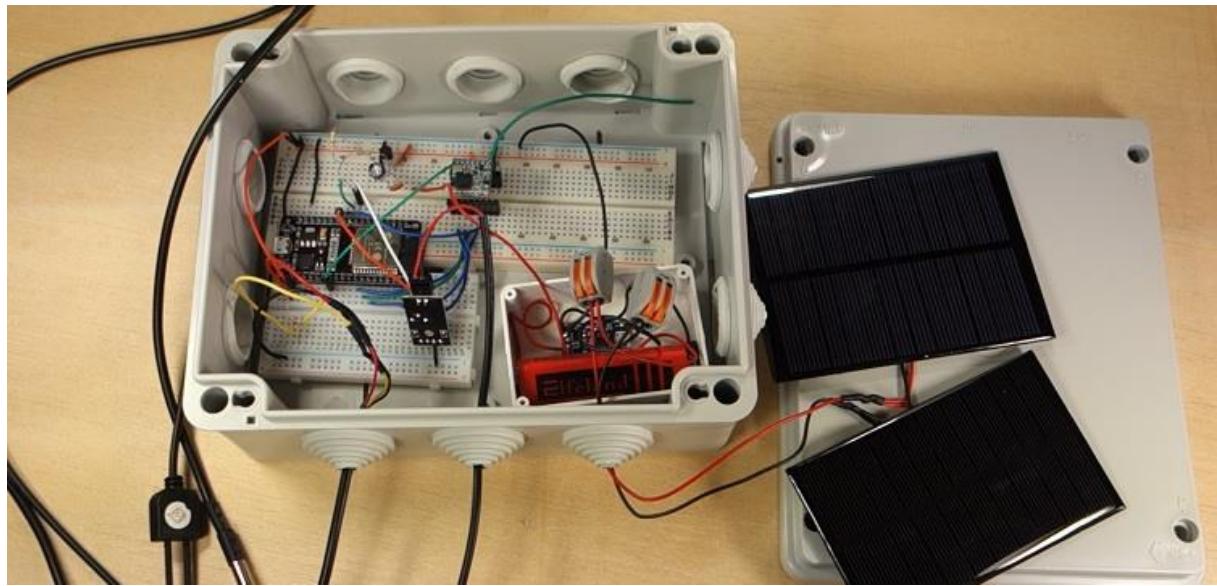
This project consists of a LoRa sender and a LoRa receiver. The LoRa sender sends outdoor soil moisture and temperature readings to an indoor LoRa receiver. The advantage of using LoRa is that you can easily establish a wireless connection

between two ESP32 boards that are more than 100 meters apart. Unlike Wi-Fi or Bluetooth that only support short-distance communication.

The receiver logs the readings on a microSD card and displays the latest sensor readings on a web server. The figure below illustrates the process.

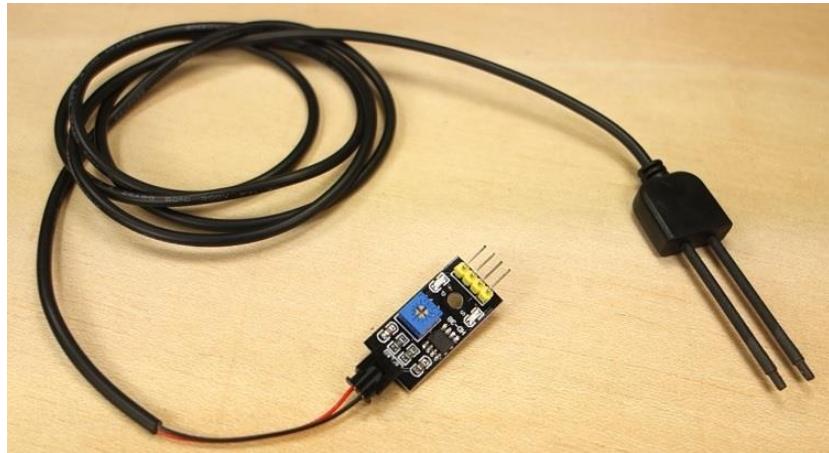


The LoRa Sender



Sensors

To read soil moisture, we're using the Resistive Soil Moisture Sensor shown in the next figure.



To read the temperature, we're going to use the waterproof version of the DS18B20 temperature sensor.

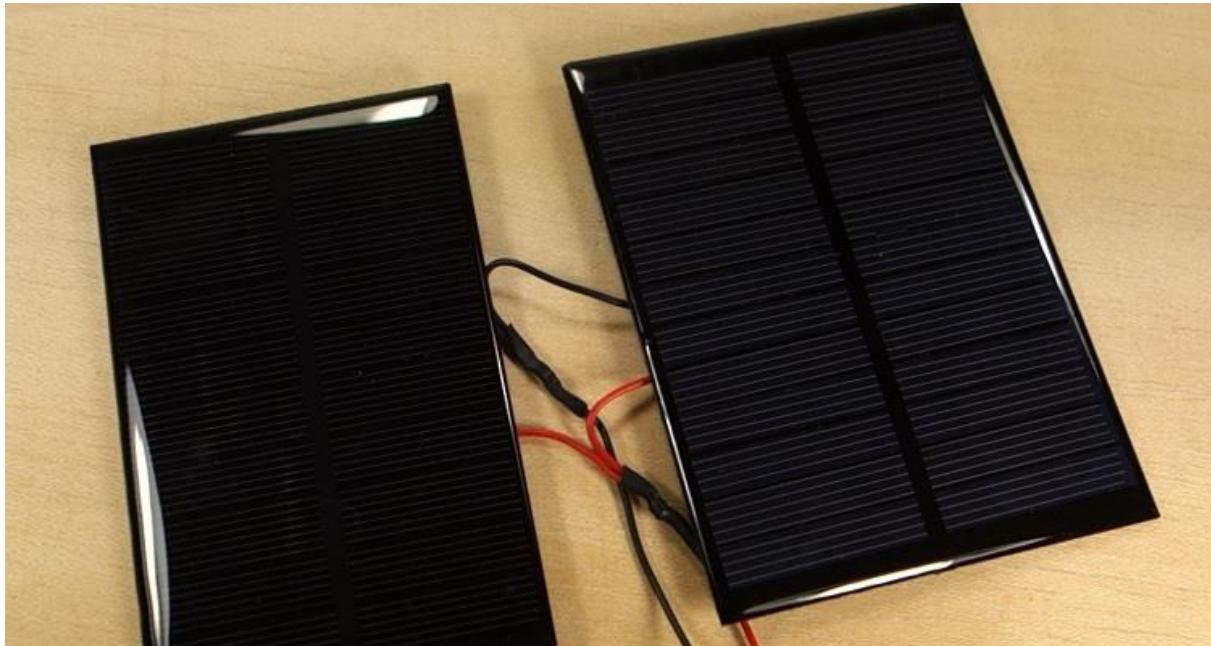


Power

To power the circuit, we're going to use a rechargeable Lithium battery with 3800mAh capacity.



The battery is charged using two mini 5V (1.2W) solar panels combined with the TP4056 lithium battery charger module.



The TP4056 battery charger module is shown in the figure below.



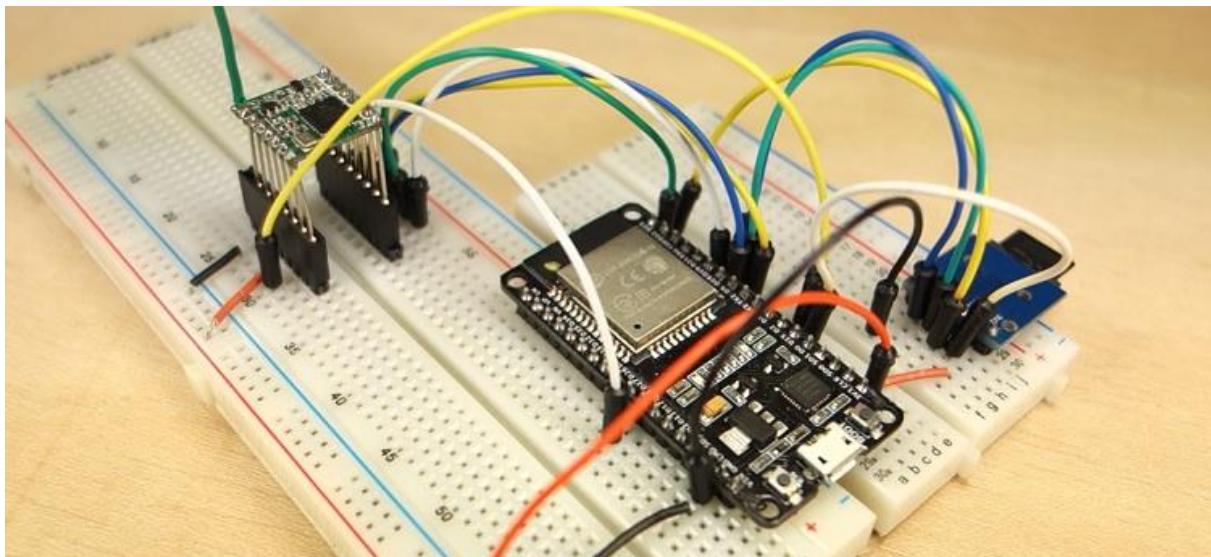
With this setup, we have a self-sustainable LoRa sender node. We'll also be monitoring the battery level in each reading.

Deep Sleep

To save power, the ESP32 will be in deep sleep mode. It wakes up every 30 minutes to take readings and send them using LoRa. After that, it goes back to deep sleep again.

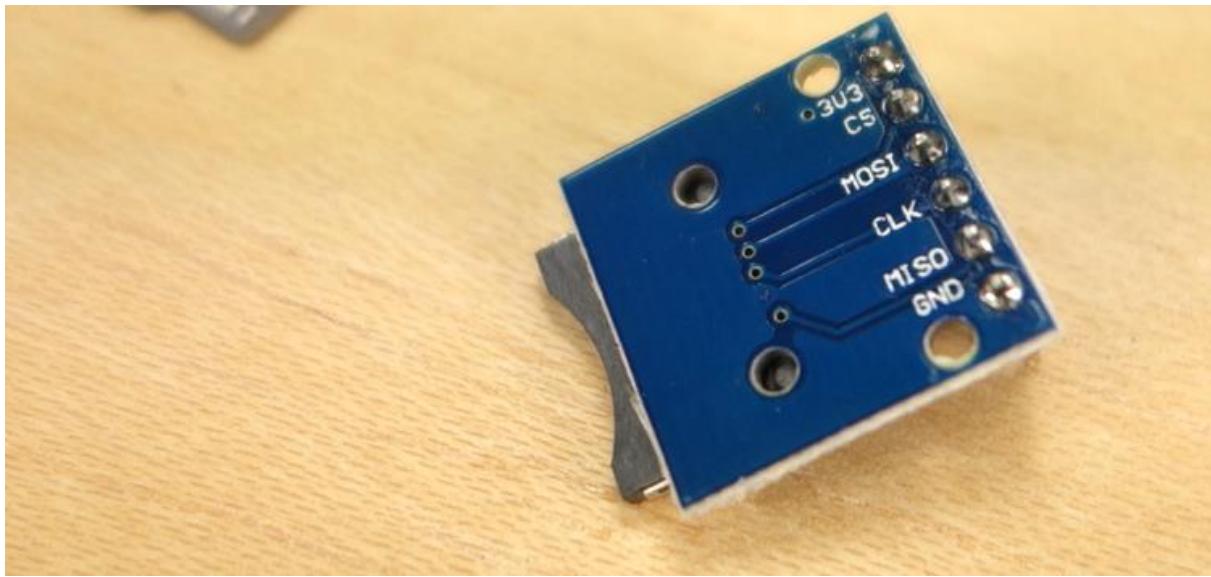


The LoRa Receiver



Data Logger

The LoRa receiver is always listening for incoming LoRa packets. When it receives a valid message, it saves the readings on the microSD card. To save data on the microSD card with the ESP32, we use the following microSD card module that communicates with the ESP32 using SPI communication protocol.

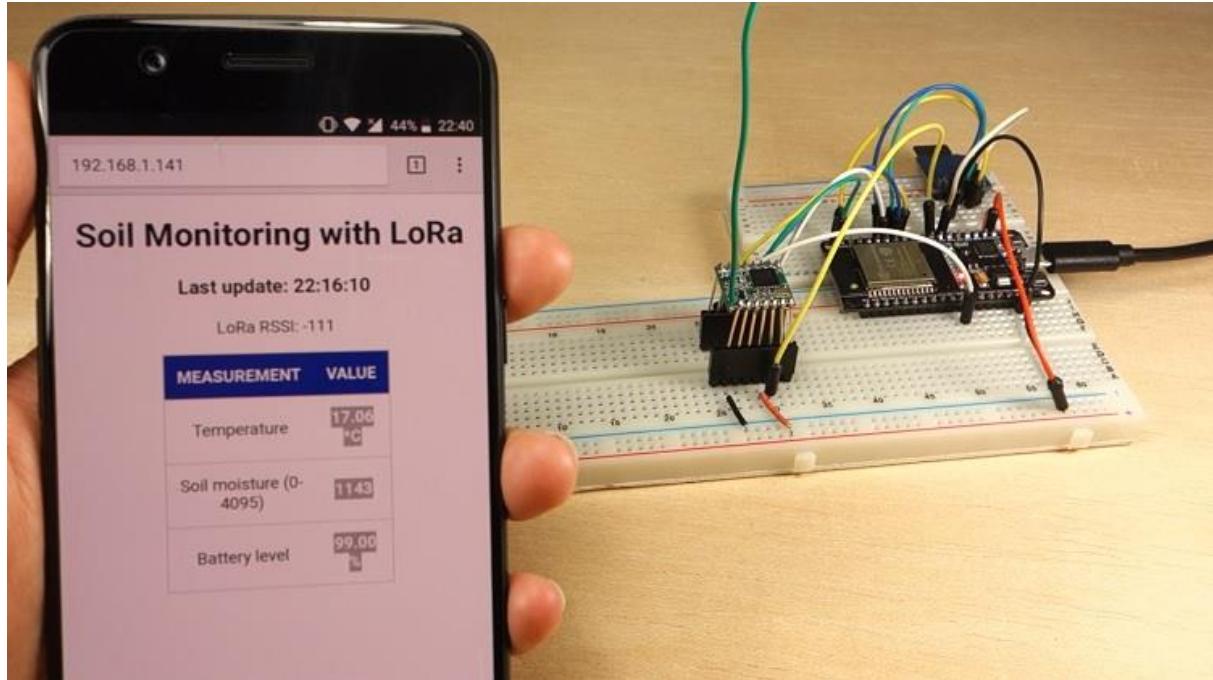


Timestamps

When the LoRa receiver gets a valid LoRa message, it requests the date and time using the Network Time Protocol (NTP). Each LoRa message is saved on the SD card with time stamps.

Web Server

The LoRa receiver also acts as a web server. The web server displays the latest sensor readings, timestamp, and battery level of the LoRa sender node. It also displays the RSSI to give us an idea of the LoRa signal's strength.



The web server should be only used occasionally to check the sensor readings.

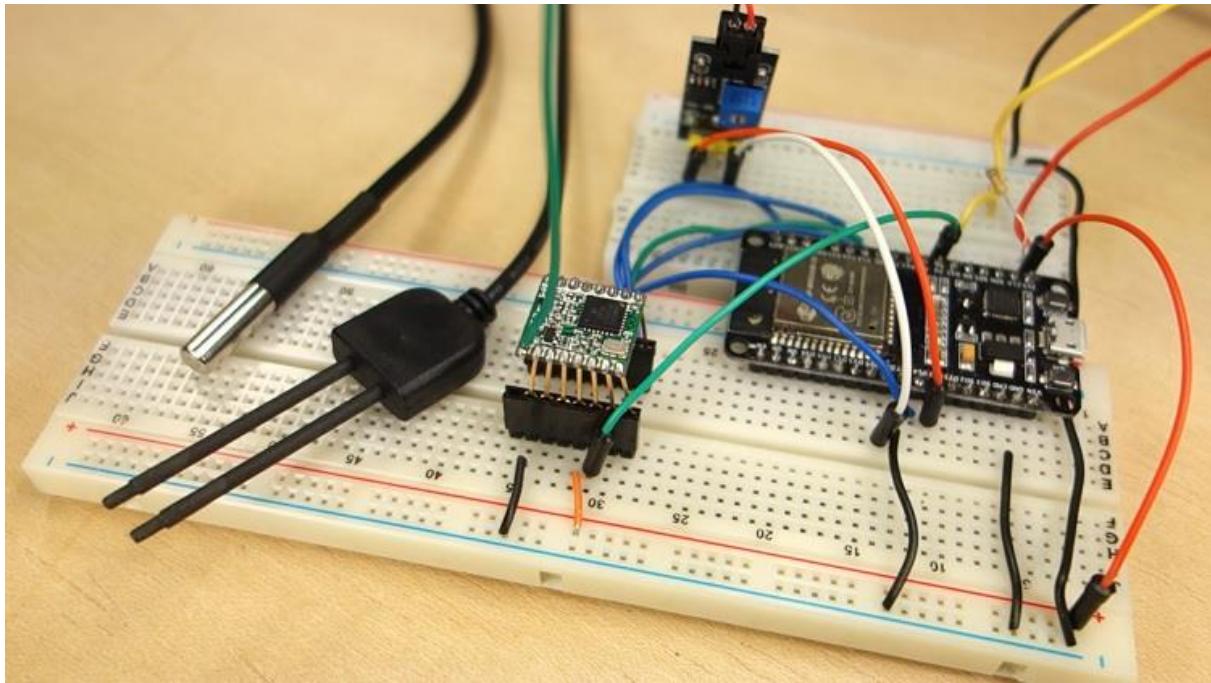
Important: after checking the readings on the web server, you must close the browser window, so that the receiver is able to get new LoRa packets.

Continue To The Next Unit...

Now that you know exactly what you're going to make, go to the next Unit to start building the LoRa sender.

Unit 2 - ESP32 LoRa Sender

In this part, we'll show you step by step how to build the LoRa Sender. The sender node reads soil moisture and temperature and sends those readings via LoRa radio to the receiver.



Parts Required:

Here's a list of the parts required for this Unit:

- [ESP32 DOIT DEVKIT V1 board](#)
- [RFM95 LoRa transceiver module](#)
- RFM95 LoRa breakout board (optional)
- Temperature sensor:
 - [DS18B20 temperature sensor \(waterproof version\)](#)
 - [10K Ohm resistor](#)
- [Resistive Soil Moisture sensor](#)
- [2x Breadboards](#)
- [Jumper Wires](#)

Preparing the Temperature Sensor

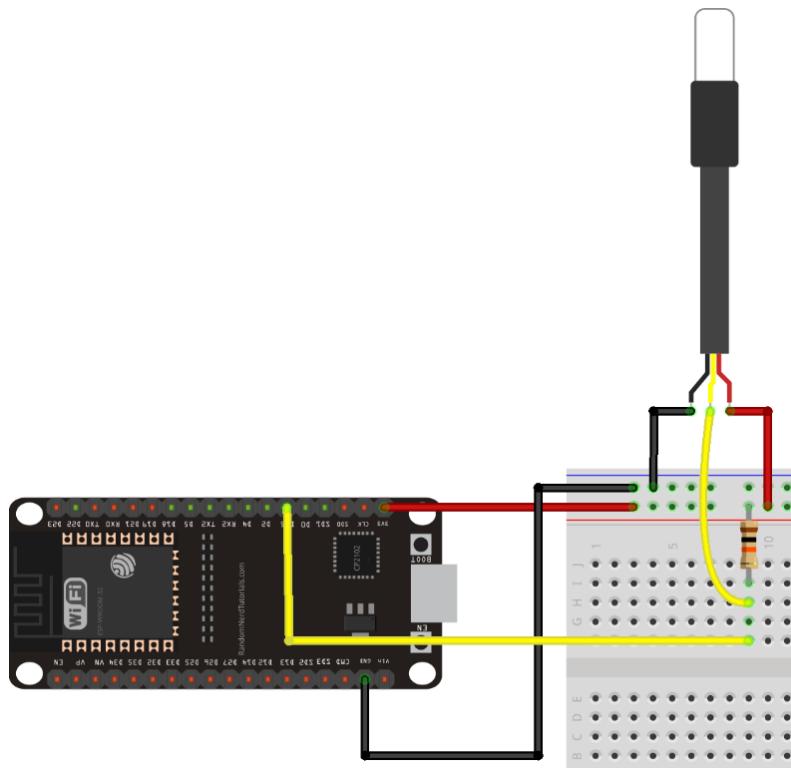
To read the temperature, we're going to use the waterproof version of the DS18B20 temperature sensor. The sensor is shown in the figure below.



The DS18B20 temperature sensor is a 1-wire digital temperature sensor. This means that you can read the temperature with a very simple circuit setup.

Wiring the DS18B20 temperature sensor

Wiring the DS18B20 temperature sensor is very straightforward. Just follow the next schematic diagram:



You can also follow the following table as a reference.

DS18B20	ESP32
VCC (red wire)	3.3V
Data (yellow wire)	GPIO 15 and 10K pull-up resistor
GND (black wire)	GND

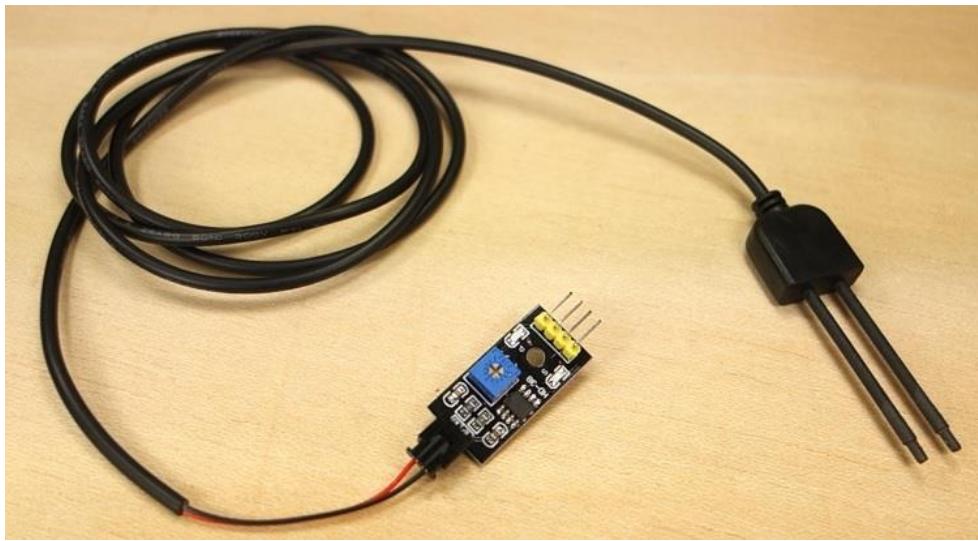
Installing Libraries (OneWire and DallasTemperature)

You need to install the [One Wire library by Paul Stoffregen](#) and the [Dallas Temperature library](#) to interface with the DS18B20 sensor.

To install these libraries, go to **Sketch > Include Libraries > Manage Libraries** and search for the libraries' names. Then, install the libraries.

Preparing the Soil Moisture Sensor

We'll use a resistive soil moisture sensor to read soil moisture, as shown in the figure below.



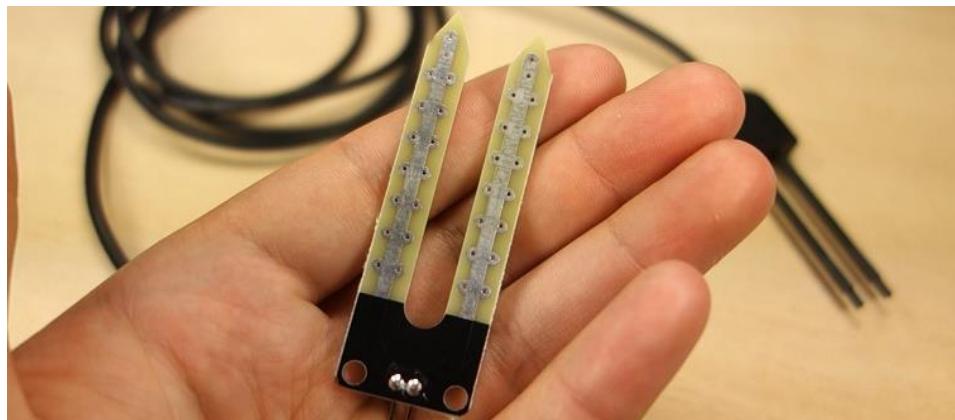
The sensor's output voltage changes accordingly to the soil moisture. When the soil is:

- **Wet:** the output voltage decreases – Minimum value: 0 → 100% moisture;
- **Dry:** the output voltage increases – Maximum value: 4095 → 0% moisture.

The analog pins of the ESP32, by default, have a 12-bit resolution. This means you can get a value between 0 and 4095, in which 0 corresponds to 100% moisture and 4095 to 0% moisture.

Increasing the lifespan of the soil moisture sensor

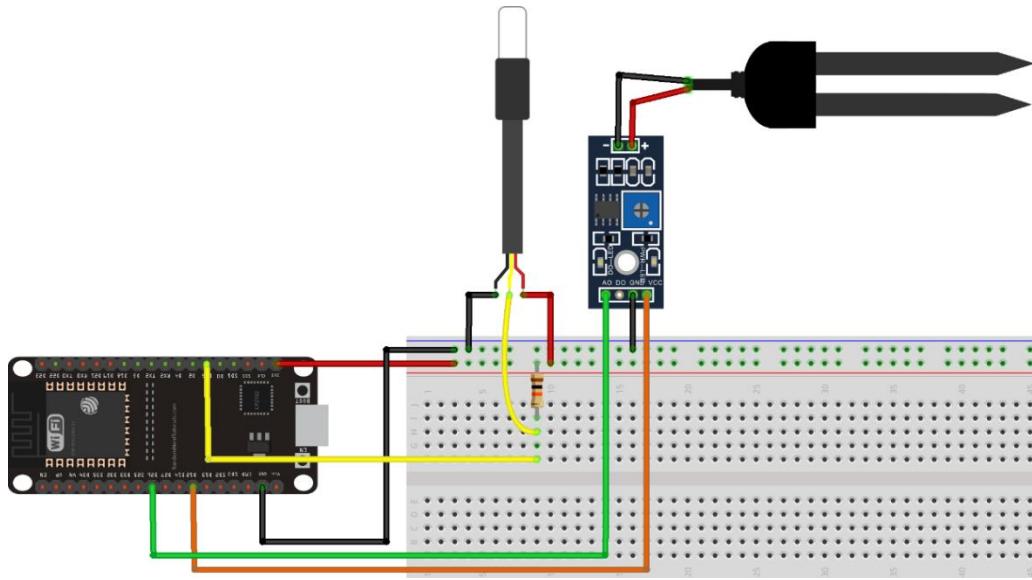
Cheap soil moisture sensors can oxidize very quickly and get damaged.



To increase the lifespan of the soil moisture sensor, we should power up the sensor only when we want to get readings. So, instead of connecting the sensor's power pin to 3.3V, we'll power the sensor using a GPIO. This way, if we want to take a reading, we apply power to the sensor by setting that GPIO to HIGH. Once the reading is taken, we set that GPIO to LOW again.

Wiring the soil moisture sensor

Add the soil moisture sensor to your setup by following the next schematic diagram.



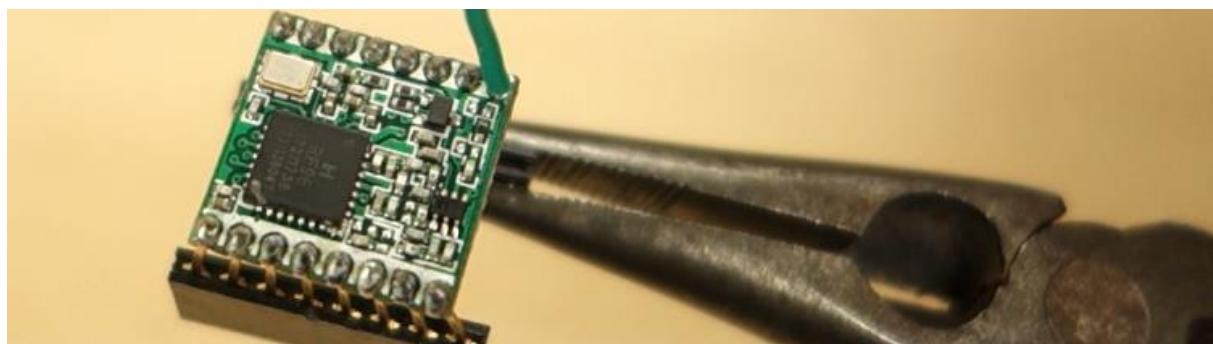
(This schematic uses the ESP32 DEVKIT V1 module version with 36 GPIOs – if you're using another model, please check the pinout for the board you're using.)

You can also use the following table as a reference to wire the soil moisture sensor:

Soil Moisture Sensor	ESP32
VCC	GPIO 12
GND	GND
A0	GPIO 26

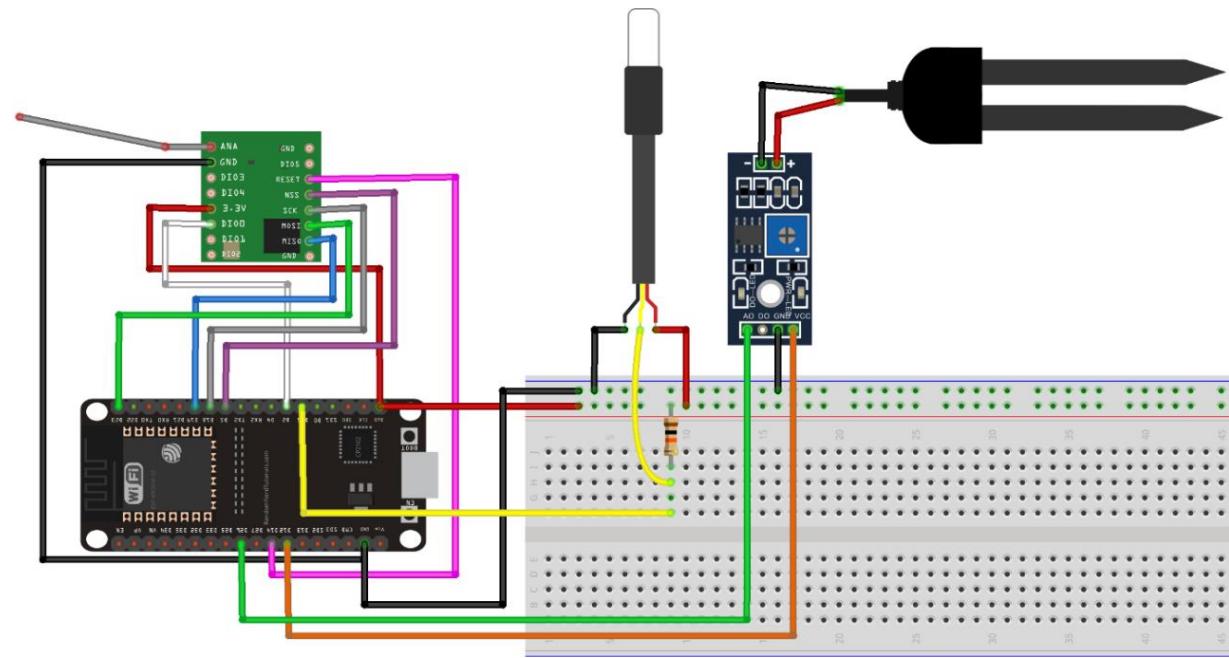
Preparing the LoRa Transceiver Module

The sensor readings are sent to the indoor receiver using LoRa. In this project, we'll use the LoRa transceiver module we've used in [Module 8](#). So, we recommend reading the LoRa Module first, to prepare your LoRa transceiver. Alternatively, you can use an ESP32 with a built-in LoRa transceiver module.



Wiring the LoRa transceiver module

After preparing the LoRa transceiver module as recommended, you can wire the module to the ESP32. It uses SPI communication protocol, so we're going to use the ESP32 default SPI pins. Add the LoRa transceiver module to your circuit by following the next schematic.



You can also follow the next wiring table.

RFM95 LoRa Transceiver Module				
ANA	Antenna	GND	-	
GND	GND	DIO5	-	
DIO3	-	RESET	GPIO 14	
DIO4	-	NSS	GPIO 5	
3.3V	3.3V	SCK	GPIO 18	
DIO0	GPIO 2	MOSI	GPIO 23	
DIO1	-	MISO	GPIO 19	
DIO2	-	GND	-	

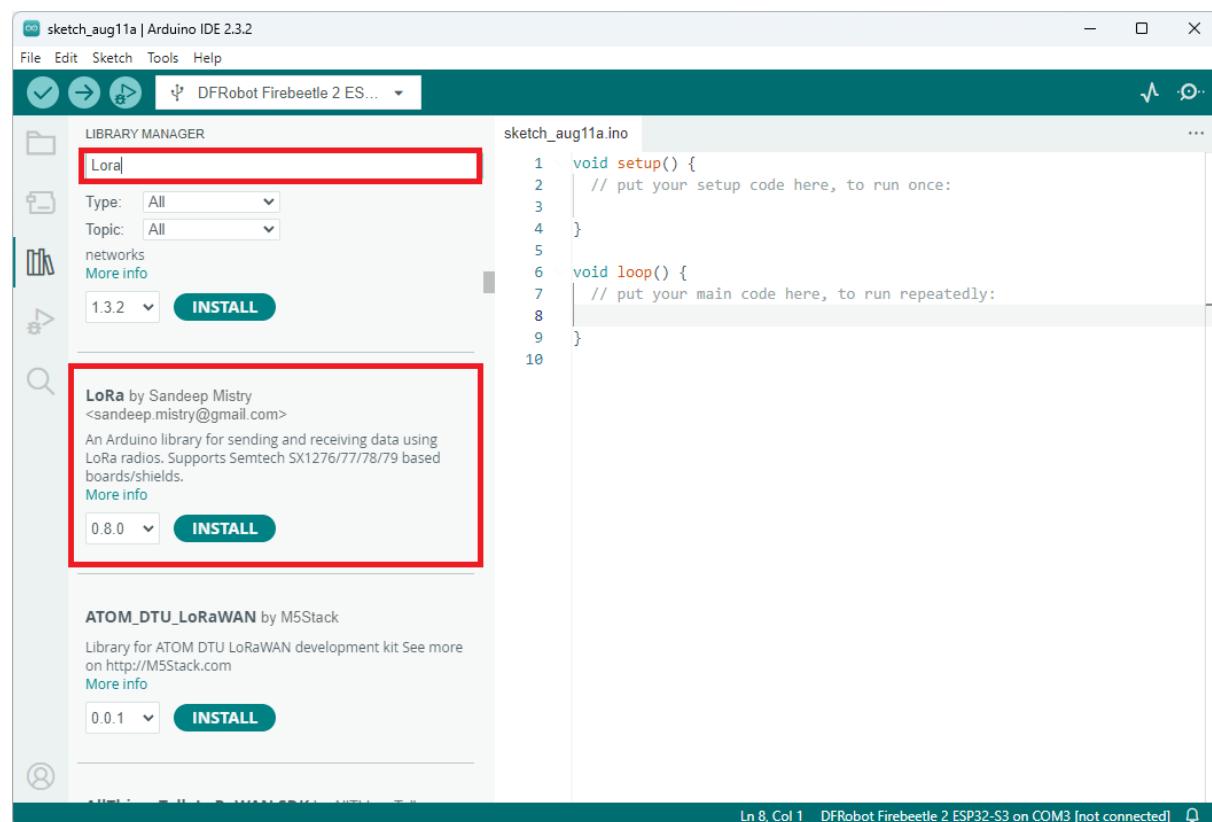
The circuit is ready for this Unit. We need to make sure that both the LoRa sender and receiver are working properly before making the sender off-the-grid. In Unit 4 of this Module, we'll be adding to the LoRa Sender:

- Rechargeable Lithium battery
- 2x solar panels
- Voltage regulator circuit
- Battery level monitoring circuit

Installing the LoRa library

To send and receive LoRa packets we'll be using the [arduino-LoRa library by sandeep mistry](#). If you've already installed the LoRa library in previous Units, you can skip this step. Otherwise, follow the installation steps below to install this library in your Arduino IDE.

Open your Arduino IDE, and go to **Sketch > Include Library > Manage Libraries** and search for “**LoRa**”. Scroll down and install the library developed by Sandeep Mistry.



The LoRa Sender Code

Copy the following code to your Arduino IDE. Don't upload it yet. First, we'll take a quick look at how it works:

- [Click here to download the code.](#)

```
// Libraries for LoRa Module
#include <SPI.h>
#include <LoRa.h>

//DS18B20 libraries
#include <OneWire.h>
#include <DallasTemperature.h>

// LoRa Module pin definition
// define the pins used by the transceiver module
#define ss 5
#define rst 14
#define dio0 2

// LoRa message variable
String message;

// Save reading number on RTC memory
RTC_DATA_ATTR int readingID = 0;

// Define deep sleep options
uint64_t uS_TO_S_FACTOR = 1000000; // Conversion factor for micro seconds to seconds
// Sleep for 30 minutes = 0.5 hours = 1800 seconds
uint64_t TIME_TO_SLEEP = 1800;

// Data wire is connected to ESP32 GPIO15
#define ONE_WIRE_BUS 15
// Setup a oneWire instance to communicate with a OneWire device
OneWire oneWire(ONE_WIRE_BUS);
// Pass our oneWire reference to Dallas Temperature sensor
DallasTemperature sensors(&oneWire);

// Moisture Sensor variables
const int moisturePin = 26;
const int moisturePowerPin = 12;
int soilMoisture;

// Temperature Sensor variables
float tempC;
float tempF;

//Variable to hold battery level;
float batteryLevel;
const int batteryPin = 27;

void setup() {
  pinMode(moisturePowerPin, OUTPUT);

  // Start serial communication for debugging purposes
  Serial.begin(115200);

  // Enable Timer wake_up
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);

  // Start the DallasTemperature library
  sensors.begin();

  // Initialize LoRa
```

```

// replace the LoRa.begin(---E-) argument with your location's frequency
// note: the frequency should match the sender's frequency
//433E6 for Asia
//866E6 for Europe
//915E6 for North America
LoRa.setPins(ss, rst, dio0);
Serial.println("initializing LoRa");

int counter = 0;
while (!LoRa.begin(866E6) && counter < 10) {
    Serial.print(".");
    counter++;
    delay(500);
}
if (counter == 10) {
    // Increment readingID on every new reading
    readingID++;
    // Start deep sleep
    Serial.println("Failed to initialize LoRa. Going to sleep now");
    esp_deep_sleep_start();
}
// Change sync word (0xF3) to match the receiver
// The sync word assures you don't get LoRa messages from other LoRa transceivers
// ranges from 0-0xFF
LoRa.setSyncWord(0xF3);
Serial.println("LoRa initializing OK!");

getReadings();
Serial.print("Battery level = ");
Serial.println(batteryLevel, 2);
Serial.print("Soil moisture = ");
Serial.println(soilMoisture);
Serial.print("Temperature Celsius = ");
Serial.println(tempC);
Serial.print("Temperature Fahrenheit = ");
Serial.println(tempF);
Serial.print("Reading ID = ");
Serial.println(readingID);

sendReadings();
Serial.print("Message sent = ");
Serial.println(message);

// Increment readingID on every new reading
readingID++;

// Start deep sleep
Serial.println("DONE! Going to sleep now.");
esp_deep_sleep_start();
}

void loop() {
    // The ESP32 will be in deep sleep
    // it never reaches the loop()
}

void getReadings() {
    digitalWrite(moisturePowerPin, HIGH);

    // Measure temperature
    sensors.requestTemperatures();
}

```

```

tempC = sensors.getTempCByIndex(0); // Temperature in Celsius
tempF = sensors.getTempFByIndex(0); // Temperature in Fahrenheit

// Measure moisture
soilMoisture = analogRead(soilMoisturePin);
digitalWrite(soilMoisturePowerPin, LOW);

//Measure battery level
batteryLevel = map(analogRead(batteryPin), 0.0f, 4095.0f, 0, 100);
}

void sendReadings() {
    // Send packet data
    // Send temperature in Celsius
    message = String(readingID) + "/" + String(tempC) + "&" +
              String(soilMoisture) + "#" + String(batteryLevel);
    // Uncomment to send temperature in Fahrenheit
    //message = String(readingID) + "/" + String(tempF) + "&" +
    //          String(soilMoisture) + "#" + String(batteryLevel);
    delay(1000);
    LoRa.beginPacket();
    LoRa.print(message);
    LoRa.endPacket();
}

```

How Does the Code Work?

Importing libraries

First, we include the libraries required for the LoRa module:

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

We also include the libraries to read from the DS18B20 temperature sensor:

```
#include <OneWire.h>
#include <DallasTemperature.h>
```

LoRa transceiver module pin definition

In the following lines, we define the pins used by the LoRa transceiver module. If you're using an ESP32 with built-in LoRa check the pins used by the LoRa module on your board and make the right pin assignment.

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

After that, we define a String variable called `message` to save the LoRa message that will be sent to the receiver.

```
String message;
```

Saving the readingID variable on the RTC memory

The `readingID` variable is saved in the RTC memory. To save that variable in the RTC memory add `RTC_DATA_ATTR` before the variable declaration. Saving the variable in the RTC memory, makes sure its value is saved during deep sleep.

```
RTC_DATA_ATTR int readingID = 0;
```

This variable is useful to keep track of the number of readings. This way, you can easily notice if the sender failed to send a new reading or if the receiver failed to receive it.

Setting the sleep time

In the following lines we set the deep sleep time. In this project the ESP32 wakes up every 30 minutes, which corresponds to 1800 seconds.

```
// Define deep sleep options
uint64_t uS_TO_S_FACTOR = 1000000; // Conversion factor for micro seconds to seconds
// Sleep for 30 minutes = 0.5 hours = 1800 seconds
uint64_t TIME_TO_SLEEP = 1800;
```

If you want to change the deep sleep time, you can modify the `TIME_TO_SLEEP` variable.

Sensors pins and variables definitions

Next, create the instances needed for the temperature sensor. The temperature sensor is connected to GPIO 15.

```
// Data wire is connected to ESP32 GPIO15
#define ONE_WIRE_BUS 15
// Setup a oneWire instance to communicate with a OneWire device
OneWire oneWire(ONE_WIRE_BUS);
// Pass our oneWire reference to Dallas Temperature sensor
DallasTemperature sensors(&oneWire);
```

We create variables for the moisture sensor data pin (GPIO 26) and the sensor power pin (GPIO 12). We also create a variable called `soilMoisture` to hold the soil moisture value.

```
const int moisturePin = 26;
```

```
const int moisturePowerPin = 12;
int soilMoisture;
```

The following float variables will be used to save the temperature in Celsius and Fahrenheit degrees:

```
float tempC;
float tempF;
```

Finally, create a float variable to hold the battery level, and define the pin to read it. In this case, we'll be using GPIO 27.

```
float batteryLevel;
const int batteryPin = 27;
```

setup()

Because this is a deep sleep code, all your code should be added to the `setup()` function. The ESP32 never reaches the `loop()`.

Setting up the temperature sensor and the soil moisture sensor

First, declare the `moisturePowerPin` as an OUTPUT.

```
pinMode(moisturePowerPin, OUTPUT);
```

Enable the timer to wake up the ESP32 every 30 minutes.

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

Start the Dallas temperature library for the DS18B20 sensor.

```
sensors.begin();
```

Setting up the LoRa module

Set the pins for the LoRa transceiver module.

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

Then, you need to initialize the LoRa module. You might need to modify the LoRa module frequency for your specific location inside the `begin()` method.

```
while (!LoRa.begin(866E6) && counter < 10) {
```

The `counter` variable ensures that the code doesn't get stuck in an infinite loop trying to connect to the LoRa module. We make 10 attempts to initialize the LoRa module.

```
while (!LoRa.begin(866E6) && counter < 10) {  
    Serial.print(".");
    counter++;
    delay(500);
}
```

If the LoRa module fails to begin after 10 attempts, we increment the reading ID, and go back to sleep.

```
if (counter == 10) {
    // Increment readingID on every new reading
    readingID++;
    // Start deep sleep
    Serial.println("Failed to initialize LoRa. Going to sleep now");
    esp_deep_sleep_start();
}
```

This ensures you don't drain your battery trying to begin the LoRa module. Also, the `readingID` variable is incremented, which means you'll notice if you don't receive that ID on the receiver.

Setting a sync word

As we've seen previously, 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.

Getting and sending readings

After initializing the LoRa module, we get the sensor readings.

```
getReadings();
```

And send them via LoRa.

```
sendReadings();
```

Then, we increment the `readingID` variable.

```
readingID++;
```

And put the ESP32 in deep sleep mode.

```
esp_deep_sleep_start();
```

The `getReadings()` and `sendReading()` functions were created to simplify the code.

getReadings() function

The `getReadings()` function starts by powering the moisture sensor by setting the `moisturePowerPin` to `HIGH`.

```
digitalWrite(moisturePowerPin, HIGH);
```

Then, we request the temperature in both Celsius and Fahrenheit degrees.

```
sensors.requestTemperatures();
tempC = sensors.getTempCByIndex(0); // Temperature in Celsius
tempF = sensors.getTempFByIndex(0); // Temperature in Fahrenheit
```

Read the soil moisture with the `analogRead()` function. After reading the value, set the moisture sensor power pin to `LOW`.

```
soilMoisture = analogRead(moisturePin);
digitalWrite(moisturePowerPin, LOW);
```

Finally, read the battery level on the `batteryPin`. This will only work in Unit 4, after adding the battery level monitoring circuit. At the moment you can connect the `batteryPin`, GPIO 27 to GND (you'll always get a battery level of 0%).

```
batteryLevel = map(analogRead(batteryPin), 0.0f, 4095.0f, 0, 100);
```

sendReadings() fucntion

The `sendReadings()` function concatenates all the readings in the `message` variable.

```
message = String(readingID) + "/" + String(tempC) + "&" +
String(soilMoisture) + "#" + String(batteryLevel);
```

Notice that we separate each reading with a special character, so the receiver can easily separate each value.

By default, we're sending the temperature in Celsius degrees, you can comment these two lines:

```
message = String(readingID) + "/" + String(tempC) + "&" +
          String(soilMoisture) + "#" + String(batteryLevel);
```

And uncomment these other two to send the temperature in Fahrenheit:

```
//message = String(readingID) + "/" + String(tempF) + "&" +
//           String(soilMoisture) + "#" + String(batteryLevel);
```

After the String message is ready, send the LoRa packet.

```
LoRa.beginPacket();
LoRa.print(message);
LoRa.endPacket();
```

That's it for the code.

Uploading and Testing the Code

The code contains many `Serial.print()` commands for debugging purposes. However, for power efficiency and during normal usage we recommend commenting or removing all the `Serial.print()` lines, after making sure everything is working properly.

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

Once the code is uploaded, open the Serial Monitor at a baud rate of 115200.

Press the ESP32 enable button and check if everything is working well. You should get a similar message in your Serial Monitor.

```
rst:0x10 (RTCWDT_RTC_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
initializing LoRa
LoRa initializing OK!
Battery level = 0.00
Soil moisture = 4095
Temperature Celsius = 22.25
Temperature Fahrenheit = 72.05
Reading ID = 0
Message sent = 0/22.25&4095#0.00
DONE! Going to sleep now.
```

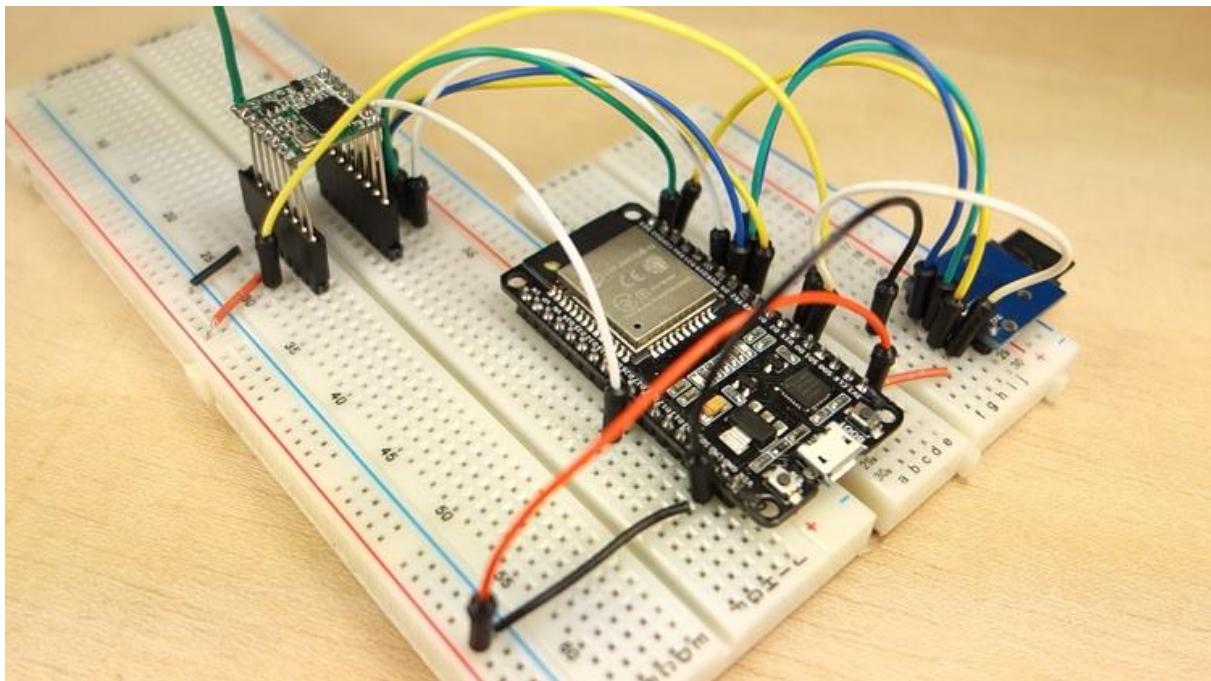
Autoscroll No line ending 115200 baud Clear output

Continue To The Next Unit ...

Your LoRa sender is ready. Now, you can go to the next Unit to start building the LoRa receiver.

Unit 3 - ESP32 LoRa Receiver

This is Part 3 of the LoRa Long Range Sensor Monitoring Project. In this Unit, we'll show you step by step how to build the LoRa receiver. We'll also test the communication between the sender and the receiver. So, you need to complete the previous Unit first.



Project Overview

Let's take a look at some of the LoRa receiver features:

- the LoRa receiver picks up the LoRa packets from the LoRa sender;
- it decodes the sensor readings from the received message and saves them on a microSD card with timestamps;
- the LoRa receiver also acts as a web server to display the latest sensor readings. So, it must have an Internet connection.

Parts Required:

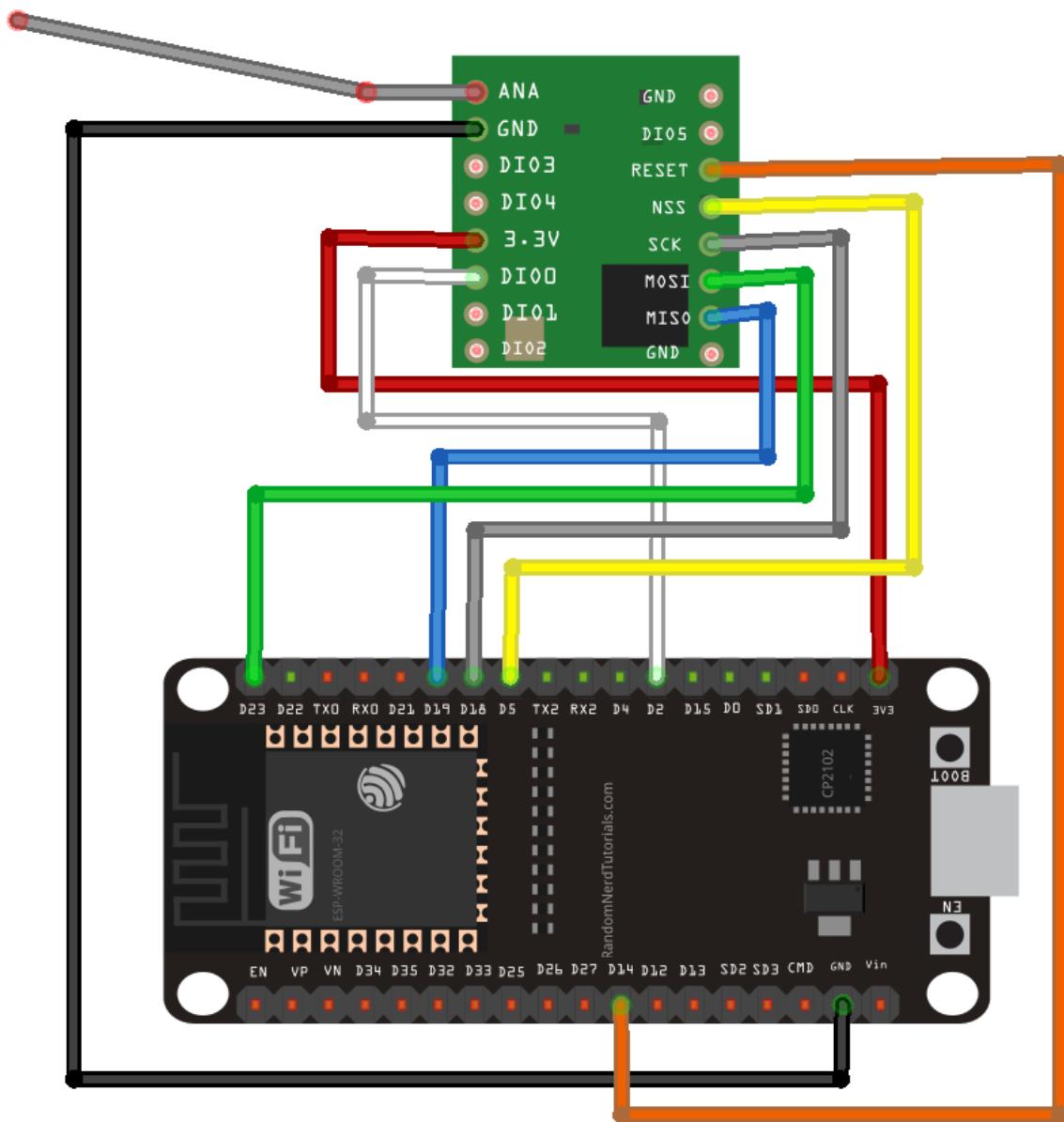
Here's a list of the parts required for this project:

- [ESP32 DOIT DEVKIT V1 board](#)
- [RFM95 LoRa transceiver module](#)

- RFM95 LoRa breakout board (optional)
- [MicroSD card module](#)
- [MicroSD card](#)
- [2x Breadboards](#)
- [Jumper Wires](#)

Preparing the LoRa Transceiver Module

Start by wiring the LoRa transceiver module to the ESP32 as we did in the previous Unit. Simply follow the next schematic diagram.



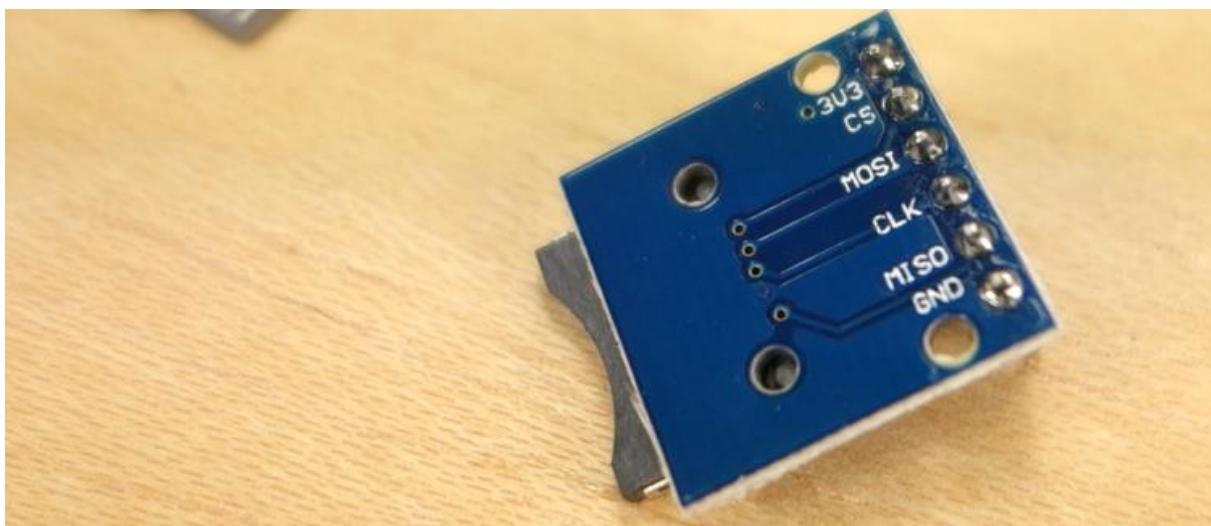
(This schematic uses the ESP32 DEVKIT V1 module version with 36 GPIOs – if you're using another model, please check the pinout for the board you're using.)

Or use the following table as a reference:

RFM95 LoRa Transceiver Module			
ANA	Antenna	GND	-
GND	GND	DIO5	-
DIO3	-	RESET	GPIO 14
DIO4	-	NSS	GPIO 5
3.3V	3.3V	SCK	GPIO 18
DIO0	GPIO 2	MOSI	GPIO 23
DIO1	-	MISO	GPIO 19
DIO2	-	GND	-

Preparing the MicroSD card module

To save data into a microSD card, we're going to use the microSD card module shown in the figure below.

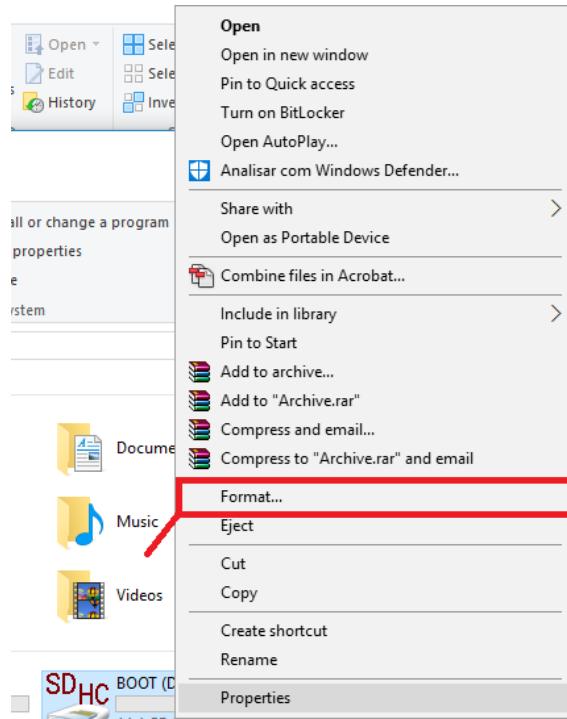


There are different models from different suppliers, but they all work in a similar way. They use SPI communication protocol. To communicate with the microSD card, we're going to use the `SD.h` library that comes installed in the Arduino IDE by default.

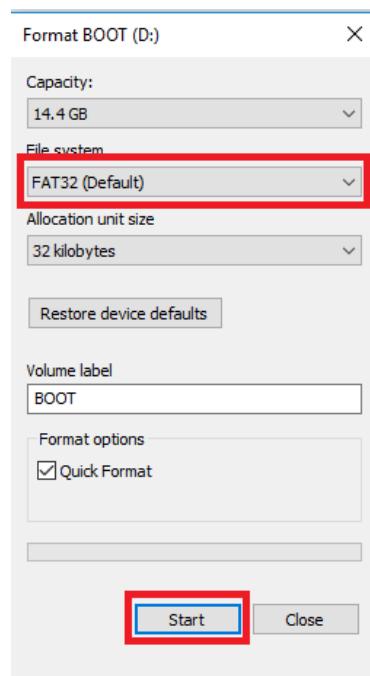
Formatting the microSD card

When using a microSD card with the ESP32, you should format it first. Follow the next instructions to format your microSD card.

1. Insert the microSD card in your computer. Go to My Computer and right click on the SD card. Select Format as shown in figure below.

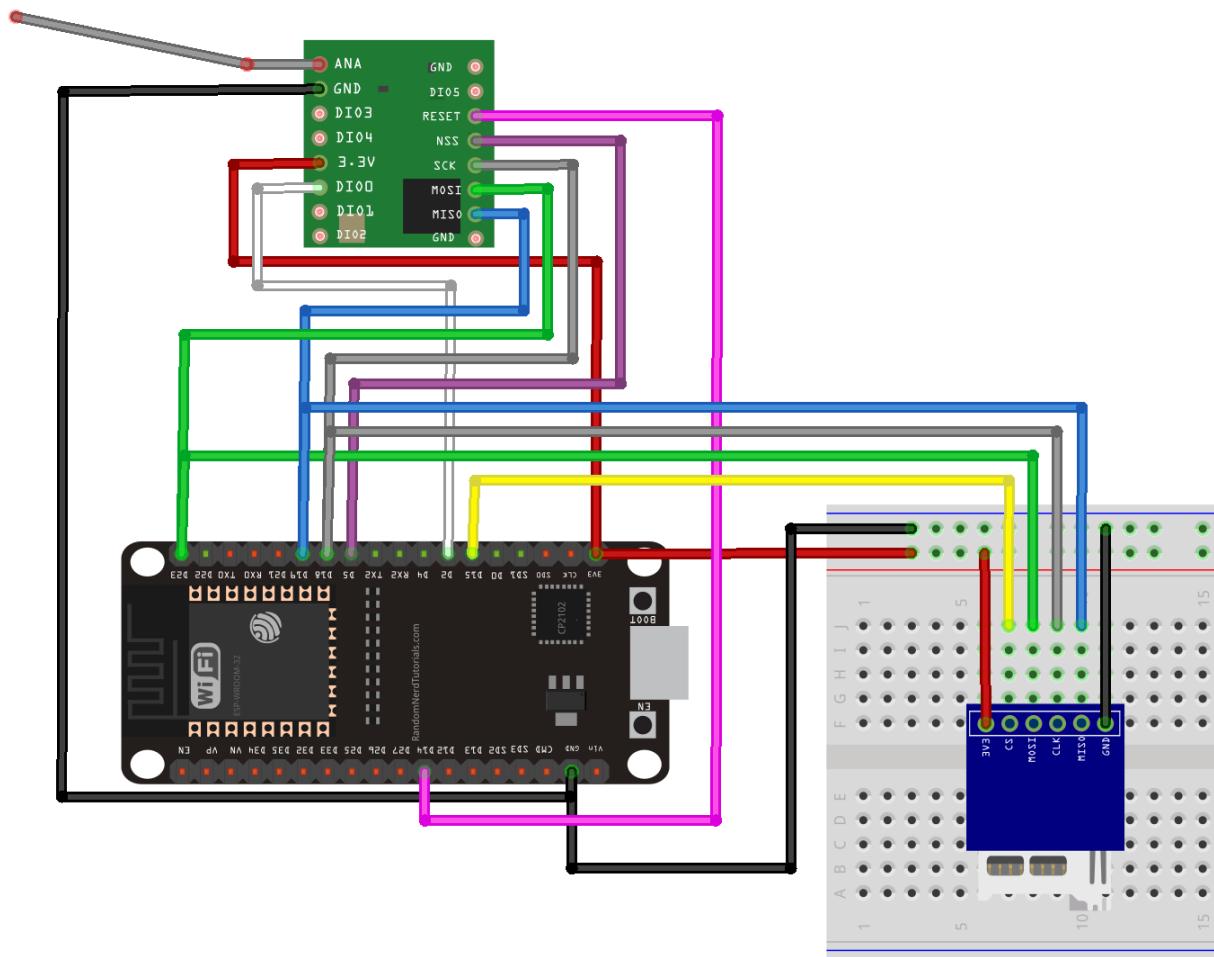


2. A new window pops up. Select FAT32, press Start to initialize the formatting process and follow the onscreen instructions.



Wiring the microSD card Module

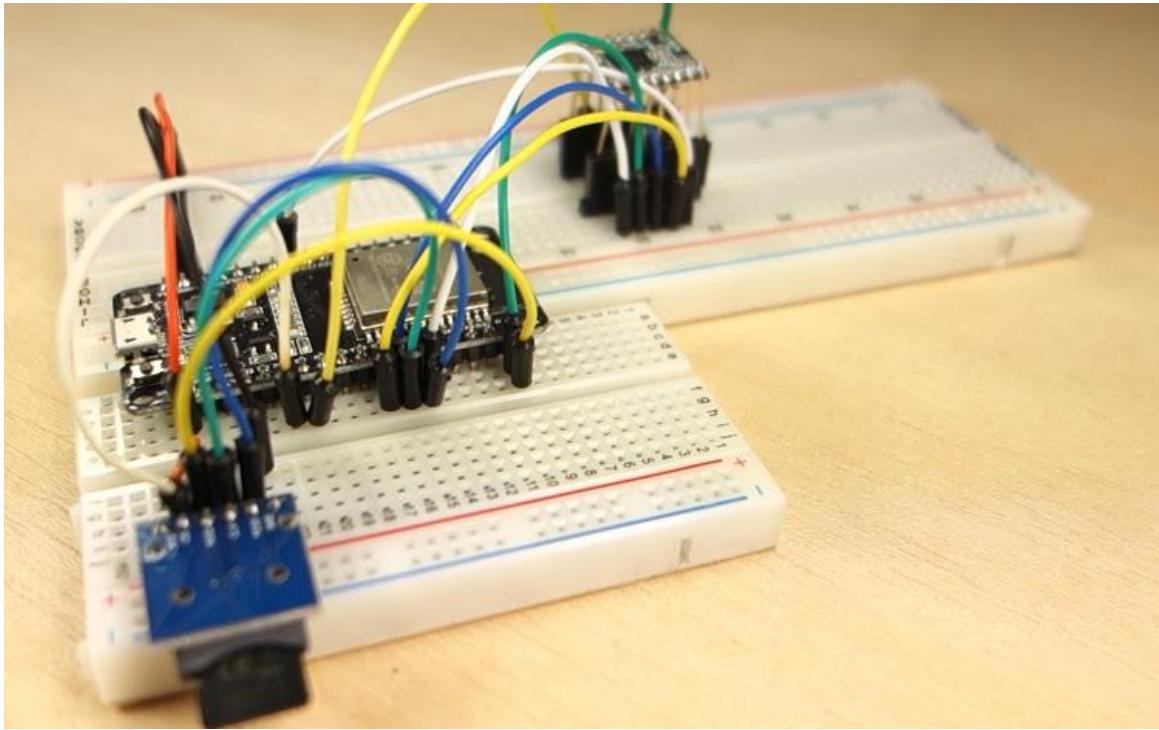
After formatting the microSD card, insert it into the module. Then, wire it to the ESP32 by following the next schematic diagram.



You can also use the following table as a reference

MicroSD Card Module	ESP32
3V3	3V3
CS	GPIO 15
MOSI	GPIO 23
CLK	GPIO 18
MISO	GPIO 19
GND	GND

The circuit is completed. Here's how your LoRa receiver circuit should look like:



Getting Date and Time

Every time the LoRa receiver picks up a new LoRa message, it will request the date and time from an NTP server to add timestamps to the data logger.

For that, we'll be using the [NTPClient library forked by Taranaïs](#). Follow the next steps to install this library in your Arduino IDE:

1. [Click here to download the NTPClient library](#). You should have a .zip folder in your Downloads folder.
2. In your Arduino IDE, go to **Sketch > Include Libraries > Add .ZIP Library...** and choose the library you've just downloaded.

If you get any issues installing the library, make sure you temporarily uninstall any other NTPClient libraries on the Arduino IDE before trying to install this one.

The LoRa Receiver Code

Copy the following code to your Arduino IDE, and let's see how it works.

- [Click here to download the code.](#)

```
// Import libraries
```

```

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

// Libraries to get time from NTP Server
#include <NTPClient.h>
#include <WiFiUdp.h>

// Libraries for SD card
#include "FS.h"
#include "SD.h"

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

// Define NTP Client to get time
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP);

// Variables to save date and time
String formattedDate;
String dayStamp;
String timeStamp;

// LoRa Module pin definition
// define the pins used by the LoRa transceiver module
#define ss 5
#define rst 14
#define dio0 2

// Initialize variables to get and save LoRa data
int rssi;
String loraMessage;
String temperature;
String soilMoisture;
String batteryLevel;
String readingID;

// Define CS pin for the SD card module
#define SD_CS 15

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

// Variable to store the HTTP request
String header;

void setup() {
    // Initialize Serial Monitor
    Serial.begin(115200);

    // Initialize LoRa
    // replace the LoRa.begin(---E-) argument with your location's frequency
    // note: the frequency should match the sender's frequency
    //433E6 for Asia
    //866E6 for Europe
    //915E6 for North America
    LoRa.setPins(ss, rst, dio0);
    while (!LoRa.begin(866E6)) {
        Serial.println(".");
}

```

```

    delay(500);
}
// Change sync word (0xF3) to match the sender
// The sync word assures you don't get LoRa messages from other LoRa transceivers
// ranges from 0-0xFF
LoRa.setSyncWord(0xF3);
Serial.println("LoRa Initializing OK!");

// 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();

// Initialize a NTPClient to get time
timeClient.begin();
// Set offset time in seconds to adjust for your timezone, for example:
// GMT +1 = 3600
// GMT +8 = 28800
// GMT -1 = -3600
// GMT 0 = 0
timeClient.setTimeOffset(3600);

// Initialize SD card
SD.begin(SD_CS);
if(!SD.begin(SD_CS)) {
    Serial.println("Card Mount Failed");
    return;
}
uint8_t cardType = SD.cardType();
if(cardType == CARD_NONE) {
    Serial.println("No SD card attached");
    return;
}
Serial.println("Initializing SD card...");
if (!SD.begin(SD_CS)) {
    Serial.println("ERROR - SD card initialization failed!");
    return; // init failed
}

// If the data.txt file doesn't exist
// Create a file on the SD card and write the data labels
File file = SD.open("/data.txt");
if(!file) {
    Serial.println("File doesn't exist");
    Serial.println("Creating file...");
    writeFile(SD, "/data.txt", "Reading ID, Date, Hour, Temperature,
                                Soil Moisture (0-4095), RSSI, Battery Level(0-100)\r\n");
}
else {
    Serial.println("File already exists");
}

```

```

        file.close();
    }

void loop() {
    // Check if there are LoRa packets available
    int packetSize = LoRa.parsePacket();
    if (packetSize) {
        getLoRaData();
        getTimeStamp();
        logSDCard();
    }
    WiFiClient client = server.available();    // Listen for incoming clients

    if (client) {                            // If a new client connects,
        Serial.println("New Client."); // print a message out in the serial port
        String currentLine = ""; // make a String to hold incoming data from the client
        while (client.connected()) { // loop while the client's connected
            if (client.available()) { // if there's bytes to read from the client,
                char c = client.read(); // read a byte, then
                Serial.write(c); // print it out the serial monitor
                header += c;
                if (c == '\n') { // if the byte is a newline character
                    // if the current line is blank, you got two newline characters in a row.
                    // that's the end of the client HTTP request, so send a response:
                    if (currentLine.length() == 0) {
                        // HTTP headers always start with a response code (e.g. HTTP/1.1 200 OK)
                        // and a content-type so the client knows what's coming, then a blank line:
                        client.println("HTTP/1.1 200 OK");
                        client.println("Content-type:text/html");
                        client.println("Connection: close");
                        client.println();

                        // Display the HTML web page
                        client.println("<!DOCTYPE html><html>");
                        client.println("<head><meta name=\"viewport\""
                                      "content=\"width=device-width, initial-scale=1\">");
                        client.println("<link rel=\"icon\" href=\"data:,\">");
                        // CSS to style the table
                        client.println("<style>body { text-align: center; font-family:"
                                      "Trebuchet MS\", Arial;}");
                        client.println("table { border-collapse: collapse; width:35%;"
                                      "margin-left:auto; margin-right:auto; }");
                        client.println("th { padding: 12px; background-color: #0043af;"
                                      "color: white; }");
                        client.println("tr { border: 1px solid #ddd; padding: 12px; }");
                        client.println("tr:hover { background-color: #bcbcbc; }");
                        client.println("td { border: none; padding: 12px; }");
                        client.println(".sensor { color:white; font-weight: bold;"
                                      "background-color: #bcbcbc; padding: 1px; }");

                        // Web Page Heading
                        client.println("</style></head><body><h1>Soil Monitoring"
                                      "with LoRa </h1>");
                        client.println("<h3>Last update: " + timeStamp + "</h3>");
                        client.println("<table><tr><th>MEASUREMENT</th><th>VALUE</th></tr>");
                        client.println("<tr><td>Temperature</td><td><span class=\"sensor\">\"");
                        client.println(temperature);
                        client.println(" *C</span></td></tr>");
                        // Uncomment the next line to change to the *F symbol
                        //client.println(" *F</span></td></tr>");
                        client.println("<tr><td>Soil moisture (0-4095)</td><td>");

```

```

        <span class=\"sensor\">>");
client.println(soilMoisture);
client.println("</span></td></tr>");
client.println("<tr><td>Battery level</td><td>
        <span class=\"sensor\">>");
client.println(batteryLevel);
client.println(" %</span></td></tr>");
client.println("<p>LoRa RSSI: " + String(rssi) + "</p>");
client.println("</body></html>");

// The HTTP response ends with another blank line
client.println();
// Break out of the while loop
break;
} else { // if you got a newline, then clear currentLine
currentLine = "";
}
} else if (c != '\r') {
// if you got anything else but a carriage return character,
currentLine += c;      // add it to the end of the currentLine
}
}
// Clear the header variable
header = "";
// Close the connection
client.stop();
Serial.println("Client disconnected.");
Serial.println("");
}

// Read LoRa packet and get the sensor readings
void getLoRaData() {
Serial.print("Lora packet received: ");
// Read packet
while (LoRa.available()) {
String LoRaData = LoRa.readString();
// LoRaData format: readingID/temperature&soilMoisture#batteryLevel
// String example: 1/27.43&654#95.34
Serial.print(LoRaData);

// Get readingID, temperature and soil moisture
int pos1 = LoRaData.indexOf('/');
int pos2 = LoRaData.indexOf('&');
int pos3 = LoRaData.indexOf('#');
readingID = LoRaData.substring(0, pos1);
temperature = LoRaData.substring(pos1 +1, pos2);
soilMoisture = LoRaData.substring(pos2+1, pos3);
batteryLevel = LoRaData.substring(pos3+1, LoRaData.length());
}
// Get RSSI
rss = LoRa.packetRssi();
Serial.print(" with RSSI ");
Serial.println(rss);
}

// Function to get date and time from NTPClient
void getTimeStamp() {
while(!timeClient.update()) {
timeClient.forceUpdate();
}
}

```

```

}

// The formattedDate comes with the following format:
// 2018-05-28T16:00:13Z
// We need to extract date and time
formattedDate = timeClient.getFormattedDate();
Serial.println(formattedDate);

// Extract date
int splitT = formattedDate.indexOf("T");
dayStamp = formattedDate.substring(0, splitT);
Serial.println(dayStamp);
// Extract time
timeStamp = formattedDate.substring(splitT+1, formattedDate.length()-1);
Serial.println(timeStamp);
}

// Write the sensor readings on the SD card
void logSDCard() {
    loRaMessage = String(readingID) + "," + String(dayStamp) + "," +
        String(timeStamp) + "," + String(temperature) +
        "," + String(soilMoisture) + "," + String(rssi) +
        "," + String(batteryLevel) + "\r\n";
    appendFile(SD, "/data.txt", loRaMessage.c_str());
}

// Write to the SD card (DON'T MODIFY THIS FUNCTION)
void writeFile(fs::FS &fs, const char * path, const char * message) {
    Serial.printf("Writing file: %s\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file) {
        Serial.println("Failed to open file for writing");
        return;
    }
    if(file.print(message)) {
        Serial.println("File written");
    } else {
        Serial.println("Write failed");
    }
    file.close();
}

// Append data to the SD card (DON'T MODIFY THIS FUNCTION)
void appendFile(fs::FS &fs, const char * path, const char * message) {
    Serial.printf("Appending to file: %s\n", path);

    File file = fs.open(path, FILE_APPEND);
    if(!file) {
        Serial.println("Failed to open file for appending");
        return;
    }
    if(file.print(message)) {
        Serial.println("Message appended");
    } else {
        Serial.println("Append failed");
    }
    file.close();
}

```

Note: if you get an error similar to "*getFormatted Date()* is not defined or not found", make sure you uninstall any other NTPClient libraries and install the one we recommend using the described installation procedure.

Importing libraries

First, you import the needed libraries for Wi-Fi and LoRa Module:

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

The following libraries allow you to request the date and time from an NTP server.

```
#include <NTPClient.h>
#include <WiFiUdp.h>
```

Import these libraries to work with the microSD card module.

```
#include "FS.h"
#include "SD.h"
```

Setting your network credentials

Type your network credentials in the following variables, so that the ESP32 can connect to your local network.

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

The following two lines define an `NTPClient` to request date and time from an NTP server.

```
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP);
```

Then, initialize String variables to save the date and time.

```
String formattedDate;
String dayStamp;
String timeStamp;
```

This next part defines the pins used by the LoRa transceiver module. If you're using an ESP32 with built-in LoRa check the pins used by the LoRa module on your board and make the right pin assignment.

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

Then, initialize variables to decode the data received in the LoRa message and to get the RSSI.

```
int rss; 
String loraMessage;
String temperature;
String soilMoisture;
String batteryLevel;
String readingID;
```

Assign GPIO 15 to the microSD card Chip Select (CS) pin:

```
#define SD_CS 15
```

These following lines are needed to create a web server:

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

// Variable to store the HTTP request
String header;
```

Initializing the LoRa transceiver module

In the `setup()`, initialize the LoRa transceiver module.

```
LoRa.setPins(ss, rst, dio0);
while (!LoRa.begin(866E6)) {
  Serial.println(".");
  delay(500);
}
```

The sync word should be the same as the LoRa sender. Otherwise, your receiver won't get the messages.

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

Also, check the LoRa frequency used for your specific location.

```
while (!LoRa.begin(866E6)) {
```

Connecting to Wi-Fi

The following snippet of code connects to the Wi-Fi network and prints the ESP32 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();
```

Initializing an NTP client

Next, initialize the NTP client to get date and time from an NTP server.

```
timeClient.begin();
```

You can use the `setTimeOffset()` method to adjust the time for your timezone.

```
timeClient.setTimeOffset(3600);
```

Here are some examples for different timezones:

- GMT +1 = 3600
- GMT +8 = 28800
- GMT -1 = -3600
- GMT 0 = 0

Initializing the microSD card module

Then, initialize the microSD card. The following if statements check if the microSD card is properly attached.

```
// Initialize SD card
SD.begin(SD_CS);
if(!SD.begin(SD_CS)) {
  Serial.println("Card Mount Failed");
  return;
}
uint8_t cardType = SD.cardType();
if(cardType == CARD_NONE) {
  Serial.println("No SD card attached");
  return;
}
Serial.println("Initializing SD card...");
if (!SD.begin(SD_CS)) {
  Serial.println("ERROR - SD card initialization failed!");
```

```
    return; // init failed  
}
```

Then, try to open the `data.txt` file on the microSD card.

```
File file = SD.open("/data.txt");
```

If that file doesn't exist, we need to create it and write the heading for the `.txt` file.

```
writeFile(SD, "/data.txt", "Reading ID, Date, Hour, Temperature, Soil Moisture  
(0-4095), RSSI, Battery Level(0-100)\r\n");
```

If the file already exists, the code continues.

```
else {  
    Serial.println("File already exists");  
}
```

Finally, we close the file.

```
file.close();
```

loop()

In the `loop()`, we check if there are any LoRa packets available. If there are, we'll get the data received in the LoRa packet, request the current date and time to create a timestamp, and log the results to the microSD card.

```
int packetSize = LoRa.parsePacket();  
if (packetSize) {  
    getLoRaData();  
    getTimeStamp();  
    logSDCard();  
}
```

In the `loop()`, we also create a web server that displays a table with the latest sensor readings and the battery level. We've covered in great detail how to build a web server in previous Units. Consult the web server Module for more information.

getLoRaData()

Let's take a look at the `getLoRaData()` function. This function reads the LoRa packet and saves the message in the `LoRaData` variable. We receive data in the following format:

```
readingID/temperature&soilMoisture#batteryLevel
```

Each reading is separated by a special character. The following part of the code splits the LoRaData variable to get each value: the readingID, temperature, soilMoisture, and batteryLevel.

```
int pos1 = LoRaData.indexOf('/');
int pos2 = LoRaData.indexOf('&');
int pos3 = LoRaData.indexOf('#');
readingID = LoRaData.substring(0, pos1);
temperature = LoRaData.substring(pos1 +1, pos2);
soilMoisture = LoRaData.substring(pos2+1, pos3);
batteryLevel = LoRaData.substring(pos3+1, LoRaData.length());
```

We also get the LoRa RSSI to gives us a general idea of the signal's strength.

```
rssi = LoRa.packetRssi();
```

getTimeStamp()

The `getTimeStamp()` function gets the date and time. These next lines ensure that we get a valid date and time:

```
while(!timeClient.update()) {
    timeClient.forceUpdate();
}
```

Sometimes the NTPClient retrieves 1970. To ensure that doesn't happen we force the update.

Then, convert the date and time to a readable format with the `getFormattedDate()` method.

```
formattedDate = timeClient.getFormattedDate();
```

The date and time are returned in this format:

```
2018-04-30T16:00:13Z
```

So, we need to split that string to get date and time separately. That's what we do here:

```
// Extract date
int splitT = formattedDate.indexOf("T");
dayStamp = formattedDate.substring(0, splitT);
Serial.println(dayStamp);
// Extract time
timeStamp = formattedDate.substring(splitT+1, formattedDate.length()-1);
```

```
Serial.println(timeStamp);
```

The date is saved on the `dayStamp` variable, and the time on the `timeStamp` variable.

logSDCard()

The `logSDCard()` function concatenates all the information in the `LoRaMessage` String variable. Each reading is separated by commas.

```
LoRaMessage = String(readingID) + "," + String(dayStamp) + "," +
String(timeStamp) + "," + String(temperature) + "," +
String(soilMoisture) + "," + String(rssi) + "," +
String(batteryLevel) + "\r\n";
```

Then, with the following line, we write all the information to the `data.txt` file in the microSD card.

```
appendFile(SD, "/data.txt", LoRaMessage.c_str());
```

Note: the `appendFile()` function only accepts variables of type `const char` for the message. So, use the `c_str()` method to convert the `LoRaMessage` variable.

writeFile() and appendFile()

The last two functions: `writeFile()` and `appendFile()` are used to write and append data to the microSD card. They come with the SD card library examples and you shouldn't modify them.

Uploading the Code

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

Testing the LoRa Receiver

To test your LoRa receiver, open the serial Monitor at a baud rate of 115200. Press the ESP32 enable button and copy the ESP32 IP address.

```
LoRa Initializing OK!
Connecting to MEO-620B4B
.....
WiFi connected.
IP address:
192.168.1.141
Initializing SD card...
File already exists
```

Autoscroll

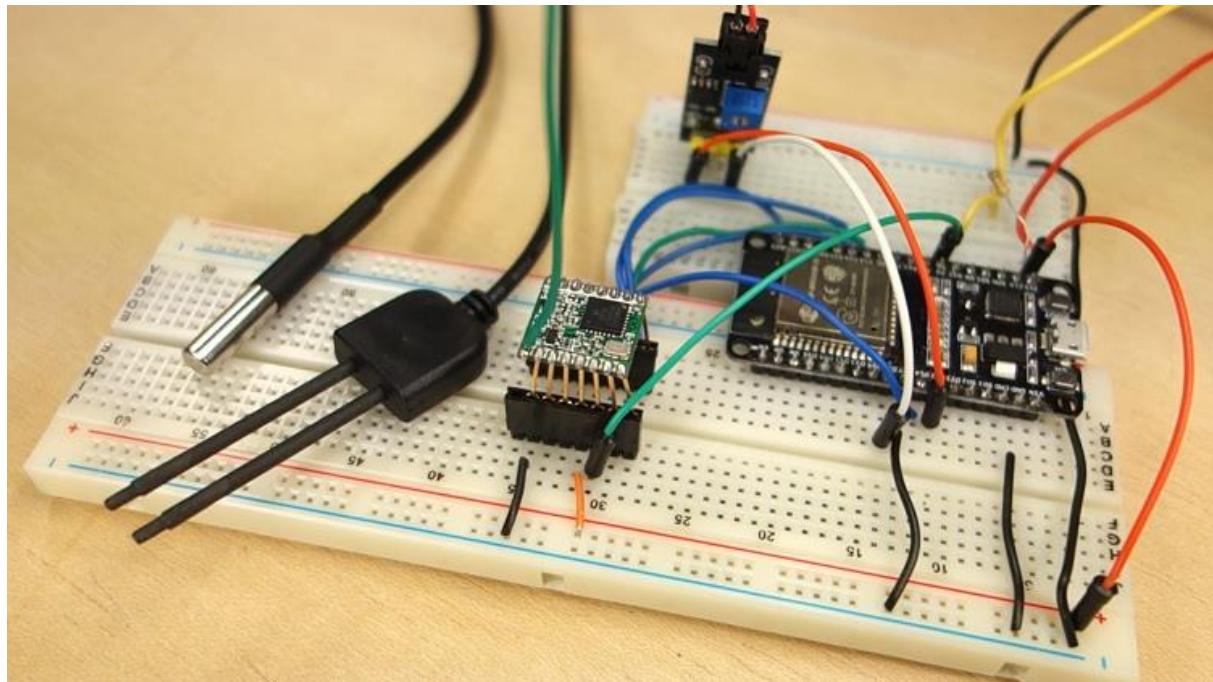
No line ending

115200 baud

Clear output

Also, check if the LoRa transceiver and the microSD card module were successfully initialized. If everything is working properly, let's see if it can receive the LoRa sender messages.

Power the sender circuit built in the previous Unit.



New readings should instantly appear on your receiver's Serial monitor. Press the ESP32 sender enable button a few times to get more readings.

The screenshot shows a terminal window titled "COM4" with the following text output:

```
File already exists
Lora packet received: 0/22.69&4095#0.00 with RSSI -36
2018-05-28T11:39:30Z
2018-05-28
11:39:30
Appending to file: /data.txt
Message appended
Lora packet received: 0/22.69&4095#0.00 with RSSI -38
2018-05-28T11:39:36Z
2018-05-28
11:39:36
Appending to file: /data.txt
Message appended
Lora packet received: 0/22.69&4095#0.00 with RSSI -36
2018-05-28T11:39:42Z
2018-05-28
11:39:42
Appending to file: /data.txt
Message appended
```

At the bottom of the terminal window, there are three buttons: "No line ending", "115200 baud", and "Clear output".

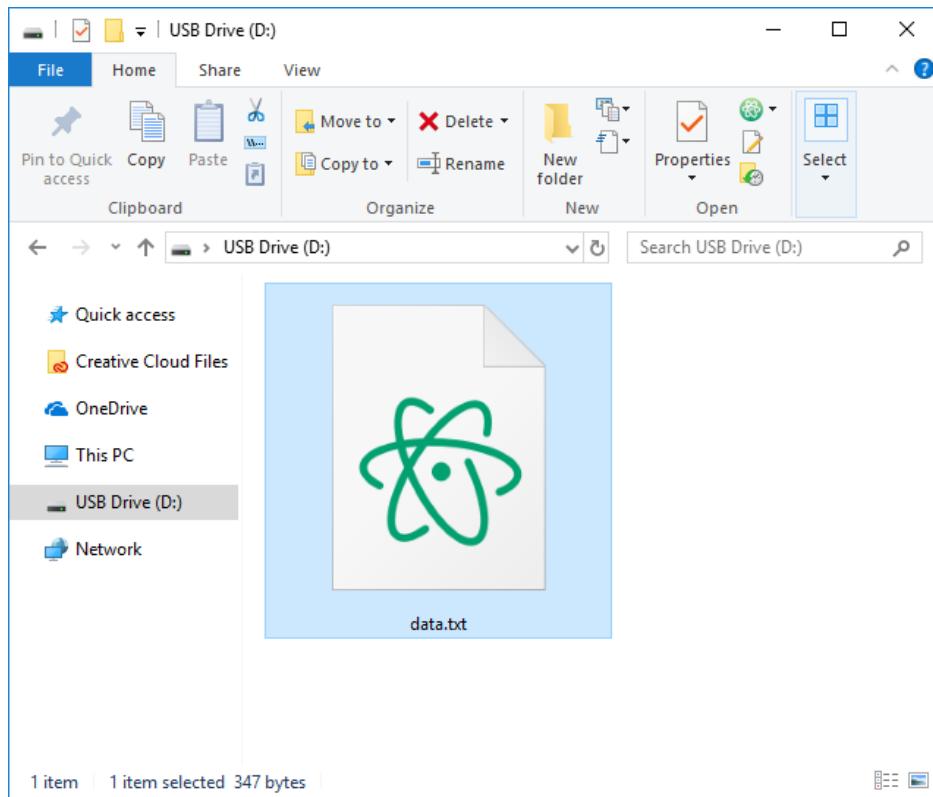
You can view the latest readings by accessing the web server. Type ESP32 IP address in your browser, and check the readings.

The screenshot shows a web browser window with the URL "192.168.1.141" in the address bar. The page title is "Soil Monitoring with LoRa". Below the title, it says "Last update: 11:39:42" and "LoRa RSSI: -36". A table displays the following data:

MEASUREMENT	VALUE
Temperature	22.69 *C
Soil moisture (0-4095)	4095
Battery level	0.00 %

Close your browser and let the sender and receiver circuits on for a few hours to test if everything is working correctly.

After the testing period, insert the microSD card into your computer and open the *data.txt* file.



You should have new readings on the file.

A screenshot of the Atom text editor. The title bar says "data.txt — D:\ — Atom". The menu bar includes "File", "Edit", "View", "Selection", "Find", "Packages", "Help", and "PlatformIO". The toolbar has icons for file operations like Open, Save, Find, and Settings. The main editor area shows the following text:

```
1 Reading ID, Date, Hour, Temperature, Soil Moisture (0-4095), RSSI, Battery Level(0-100)
2 0,2018-05-28,11:35:43,21.94,4095,-36,0.00
3 0,2018-05-28,11:39:30,22.69,4095,-36,0.00
4 0,2018-05-28,11:39:36,22.69,4095,-38,0.00
5 0,2018-05-28,11:39:42,22.69,4095,-36,0.00
6 1,2018-05-28,12:09:40,22.87,4095,-36,0.00
7 2,2018-05-28,12:39:36,22.71,4095,-36,0.00
8
```

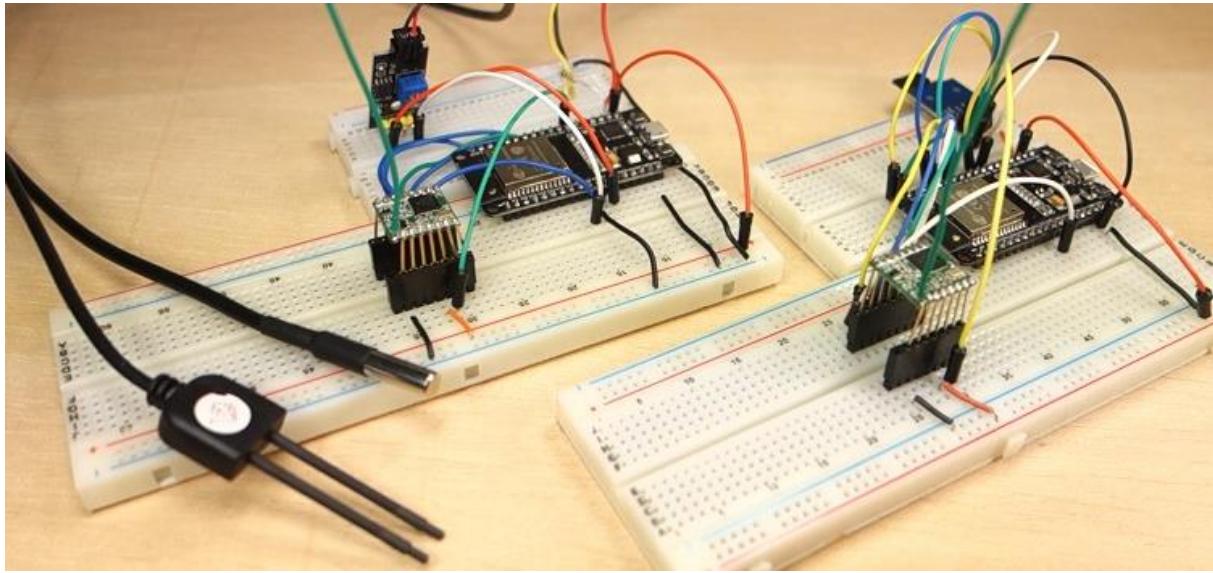
The status bar at the bottom shows "D:\data.txt", "0 files", "8 updates", and file encoding information.

Continue To The Next Unit ...

If everything is working, go to the next Unit to make your LoRa Sender solar powered.

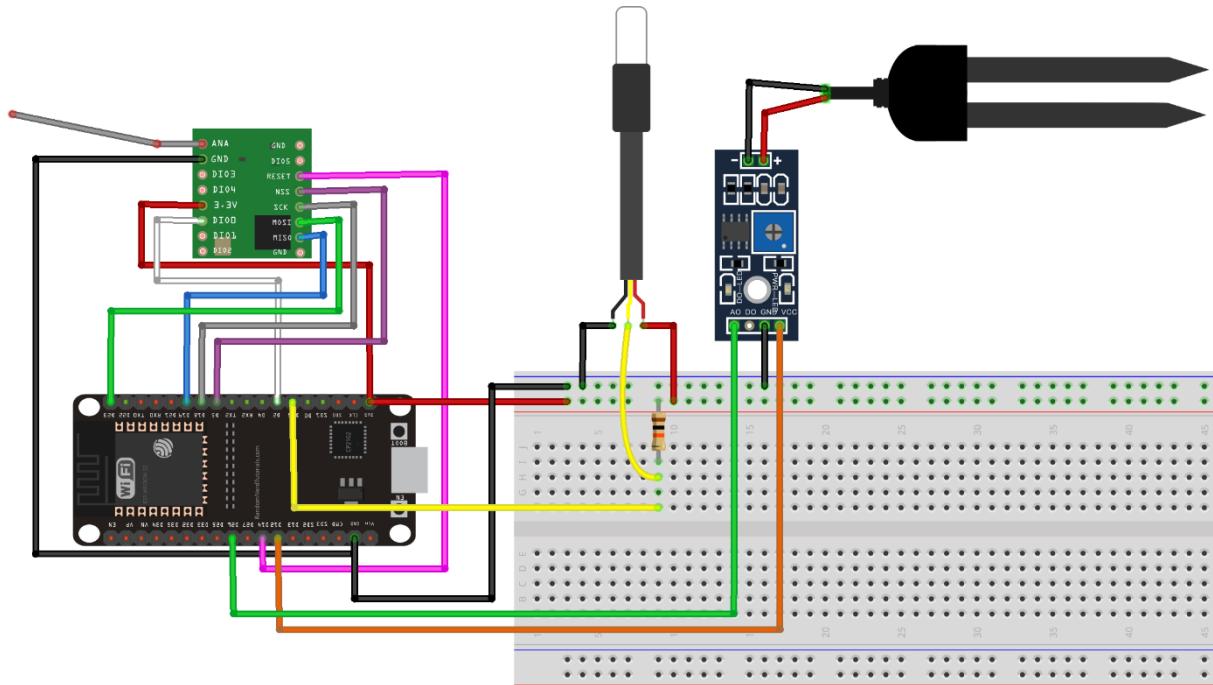
Unit 4 - LoRa Sender Solar Powered

In Part 2 you've built the LoRa sender circuit and in Part 3 the LoRa receiver.



In this Unit, you will add a lithium battery, two solar panels, a voltage regulator circuit, and a battery voltage level monitoring circuit.

To proceed with the project, your LoRa sender circuit should look like the following schematic diagram. This is the schematic diagram shown in Part 2.



(This schematic uses the **ESP32 DEVKIT V1** module version with 36 GPIOs – if you're using another model, please check the pinout for the board you're using.)

Note: you should only make the LoRa Sender circuit solar powered after testing the code and making sure it's working properly and the receiver is getting the readings (Part 2 and Part 3).

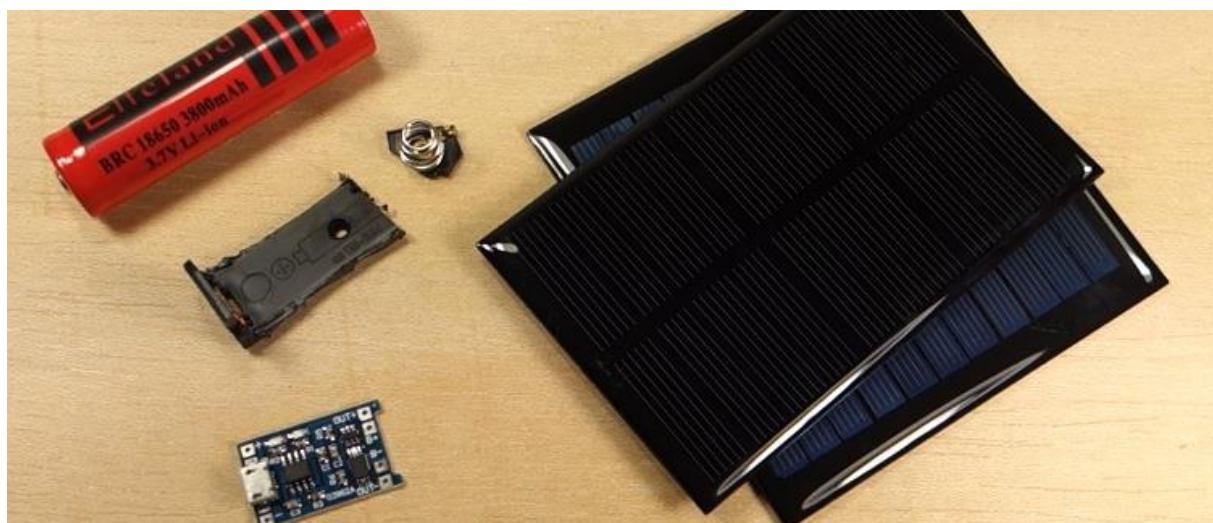
Parts Required:

Here are the parts required for this part of the project:

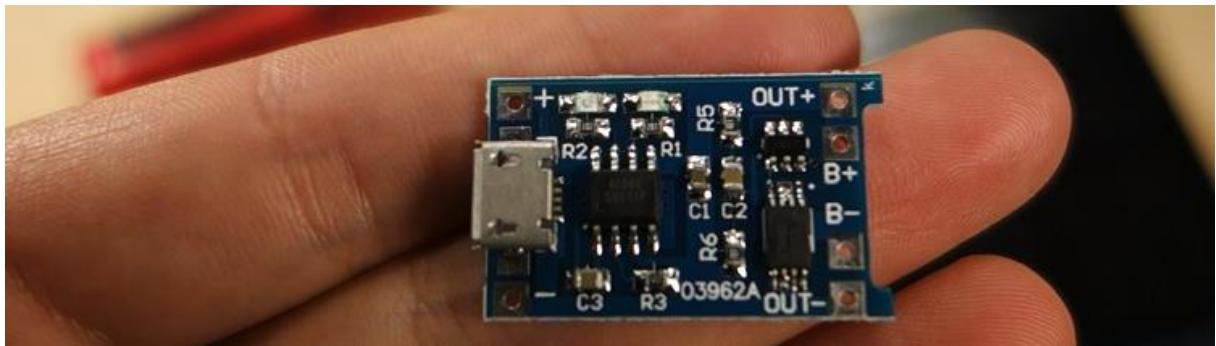
- [Lithium Li-ion battery \(at least 3800mAh capacity\)](#)
- Battery holder
- [Battery charger \(optional\)](#)
- [TP4056 Lithium Battery Charger](#)
- [2x Mini Solar Panel \(5V 1.2W\)](#)
- Battery voltage level monitor: [27K Ohm resistor + 100K Ohm resistor](#)
- Voltage regulator:
 - [Low-dropout or LDO regulator \(MCP1700-3320E\)](#)
 - [100uF electrolytic capacitor](#)
 - [100nF ceramic capacitor](#)

Preparing the Solar Panels and Lithium Battery

We've designed the LoRa sender node to be self-sustainable. To make your circuit solar powered, you need a lithium battery fully charged, a battery holder, two mini solar panels (5V and at least 1.2W), and the TP4056 lithium battery charger module.

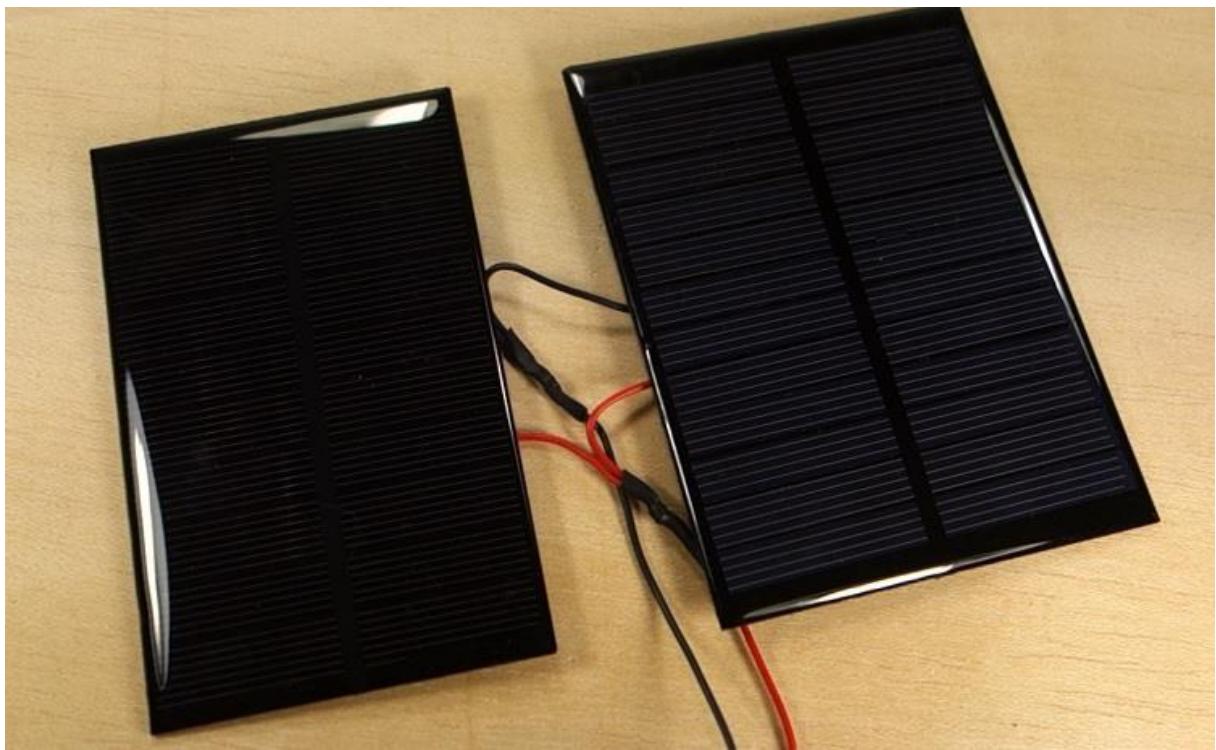


The TP4056 lithium battery charger module comes with circuit protection. It prevents battery over-voltage and reverse polarity connection.



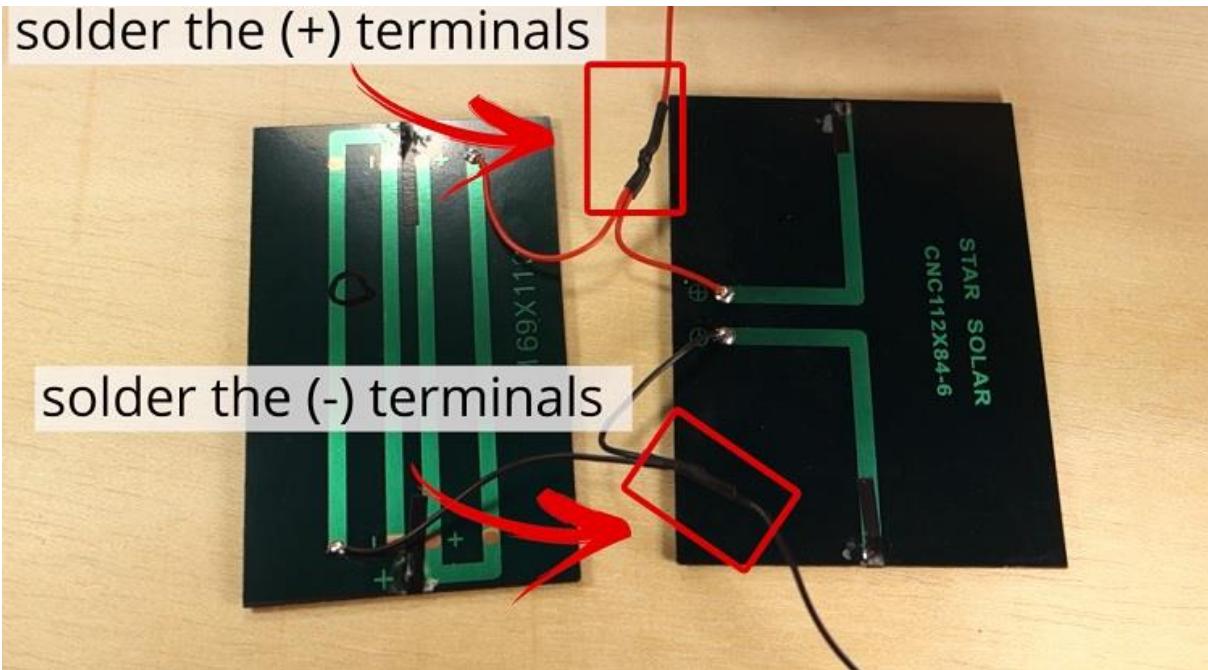
Solar panels in parallel

To charge the battery faster, we're going to use two solar panels in parallel (instead of one).



To wire solar panels in parallel, solder the plus terminals with each other, as well as the minus terminals.

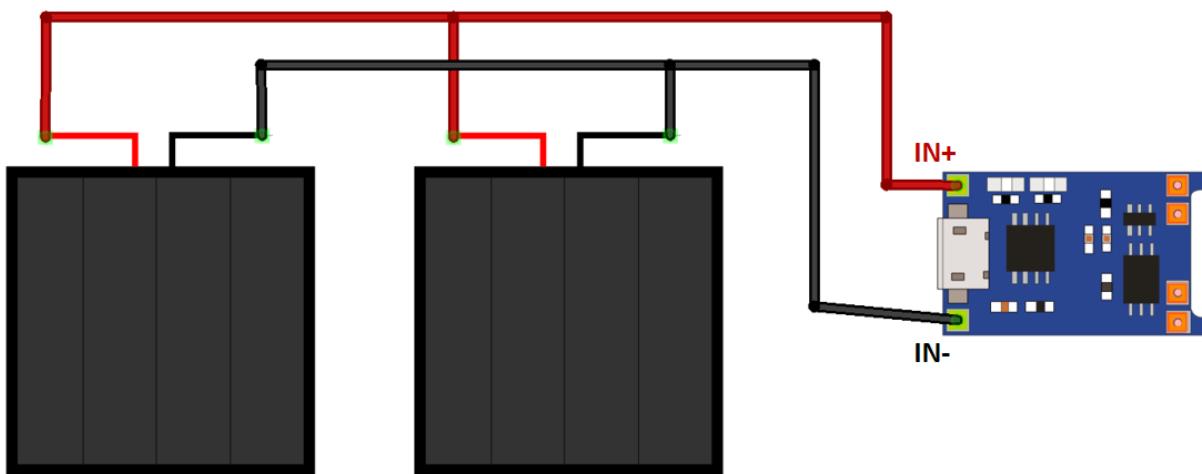
solder the (+) terminals



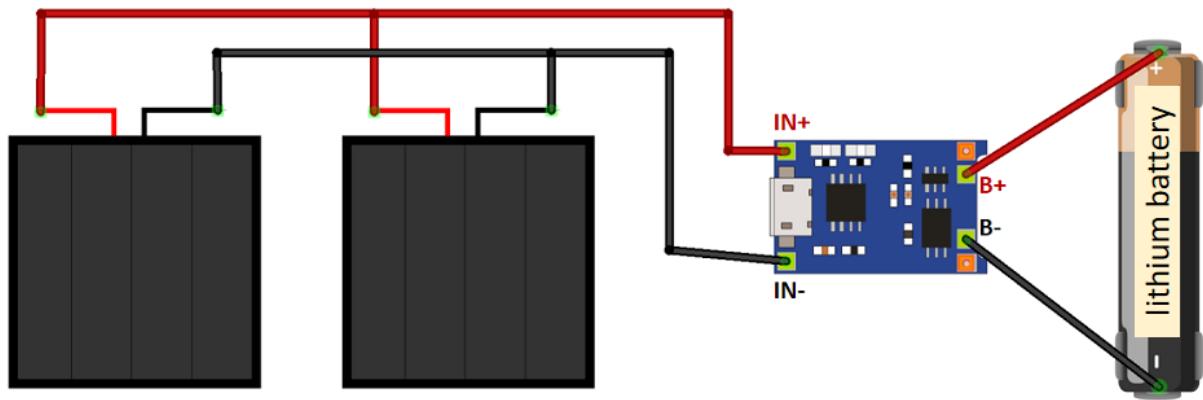
When wiring solar panels in parallel, you'll get the same output voltage, and double the current (for identical solar panels).

Wiring the TP4056 charger module

Wire the solar panels to the TP4056 lithium battery charger module as shown in the schematic diagram below. With the positive terminals connected to the pad marked with IN+ and the negative terminals to the pad marked with IN-.



Then, connect the battery holder positive terminal to the B+ pad, and the battery holder negative terminal to the B- pad.



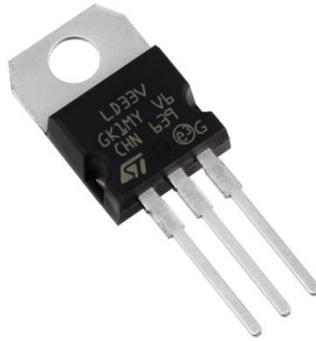
The OUT+ and OUT- will be powering the ESP32. However, lithium batteries output up to 4.2V when fully charged, but the ESP32 recommended operating voltage is 3.3V.



Important: you can't plug the Lithium battery directly to an ESP32, you need a voltage regulator circuit.

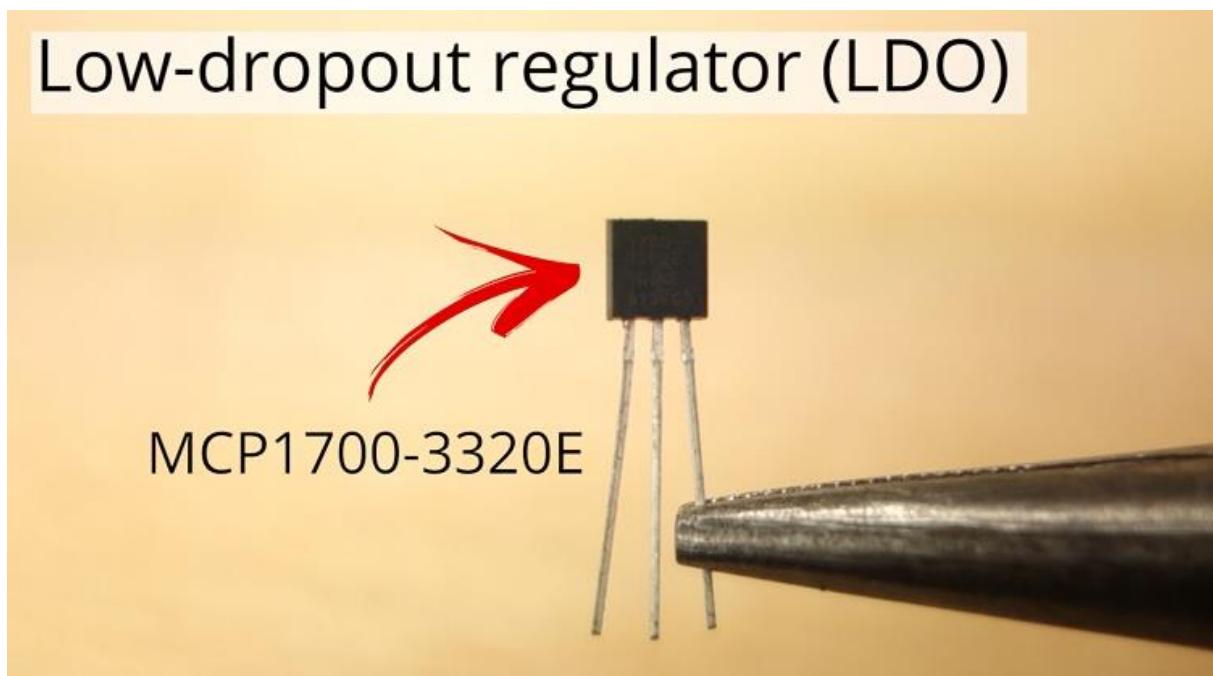
Voltage Regulator

Using a typical linear voltage regulator to drop the voltage from 4.2V to 3.3V isn't a good idea, because as the battery discharges to, for example 3.7V, your voltage regulator would stop working, because it has a high cutoff voltage.



To drop the voltage efficiently with batteries, you need to use a low-dropout regulator, or LDO for short, that can regulate the output voltage.

After researching LDOs, the [MCP1700-3320E](#) is the best for what we want to do.



There is also a good alternative like the HT7333-A.



Any LDO that has similar specifications to these two are also good alternatives. Your LDO should have similar specs when it comes to output voltage, quiescent current, output current and a low dropout voltage. Take a look at the datasheet below.

Low Quiescent Current LDO

Features:

- 1.6 μ A Typical Quiescent Current
- Input Operating Voltage Range: 2.3V to 6.0V
- Output Voltage Range: 1.2V to 5.0V
- 250 mA Output Current for Output Voltages \geq 2.5V
- 200 mA Output Current for Output Voltages < 2.5V
- Low Dropout (LDO) Voltage
 - 178 mV Typical @ 250 mA for $V_{OUT} = 2.8V$
- 0.4% Typical Output Voltage Tolerance
- Standard Output Voltage Options:
 - 1.2V, 1.8V, 2.5V, 2.8V, 3.0V, 3.3V, 5.0V
- Stable with 1.0 μ F Ceramic Output Capacitor
- Short Circuit Protection
- Overtemperature Protection

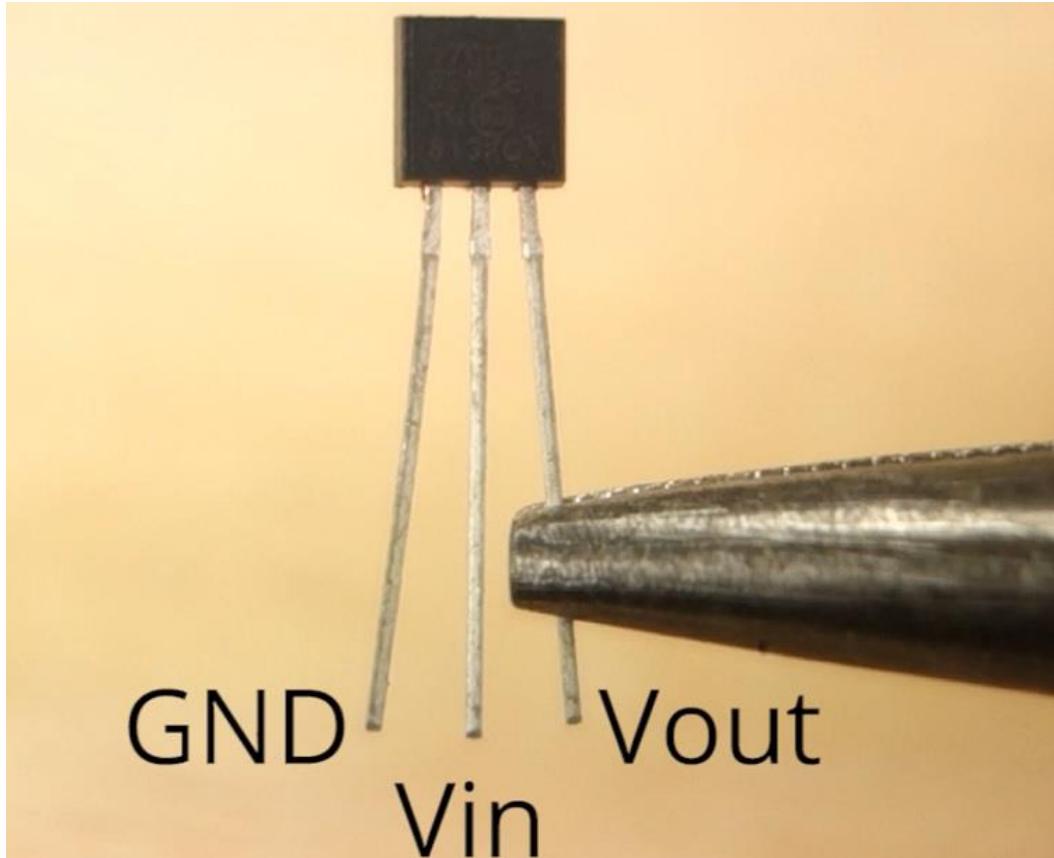
General Description:

The MCP1700 is a family of CMOS low dropout (LDO) voltage regulators that can deliver up to 250 mA of current while consuming only 1.6 μ A of quiescent current (typical). The input operating range is specified from 2.3V to 6.0V, making it an ideal choice for two and three primary cell battery-powered applications, as well as single cell Li-Ion-powered applications.

The MCP1700 is capable of delivering 250 mA with only 178 mV of input to output voltage differential ($V_{OUT} = 2.8V$). The output voltage tolerance of the MCP1700 is typically $\pm 0.4\%$ at $+25^\circ C$ and $\pm 3\%$ maximum over the operating junction temperature range of $-40^\circ C$ to $+125^\circ C$.

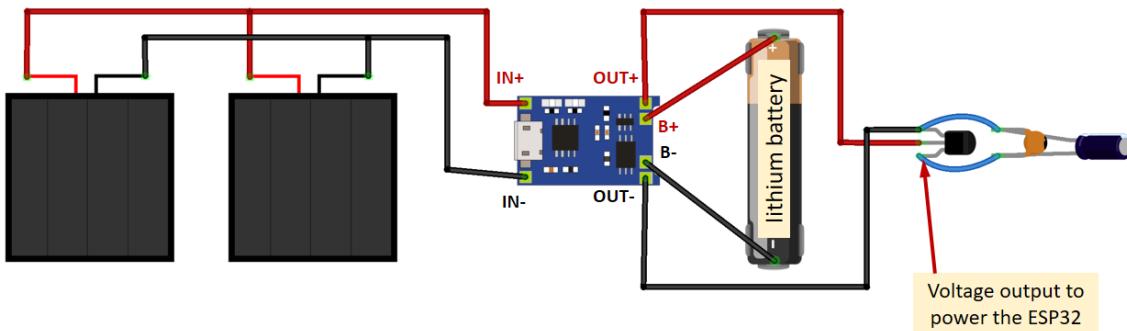
Output voltages available for the MCP1700 range from 1.2V to 5.0V. The LDO output is stable when using only 1 μ F output capacitance. Ceramic, tantalum or aluminum electrolytic capacitors can all be used for

Here's the MCP1700-3320E pinout. It has GND, Vin, and Vout.



The LDOs should have a ceramic capacitor and an electrolytic capacitor connected in parallel to GND and Vout to smooth the voltages peaks. Here we're using a 100uF electrolytic capacitor, and a 100nF ceramic capacitor.

Follow the next schematic to add the voltage regulator circuit to the previous setup.

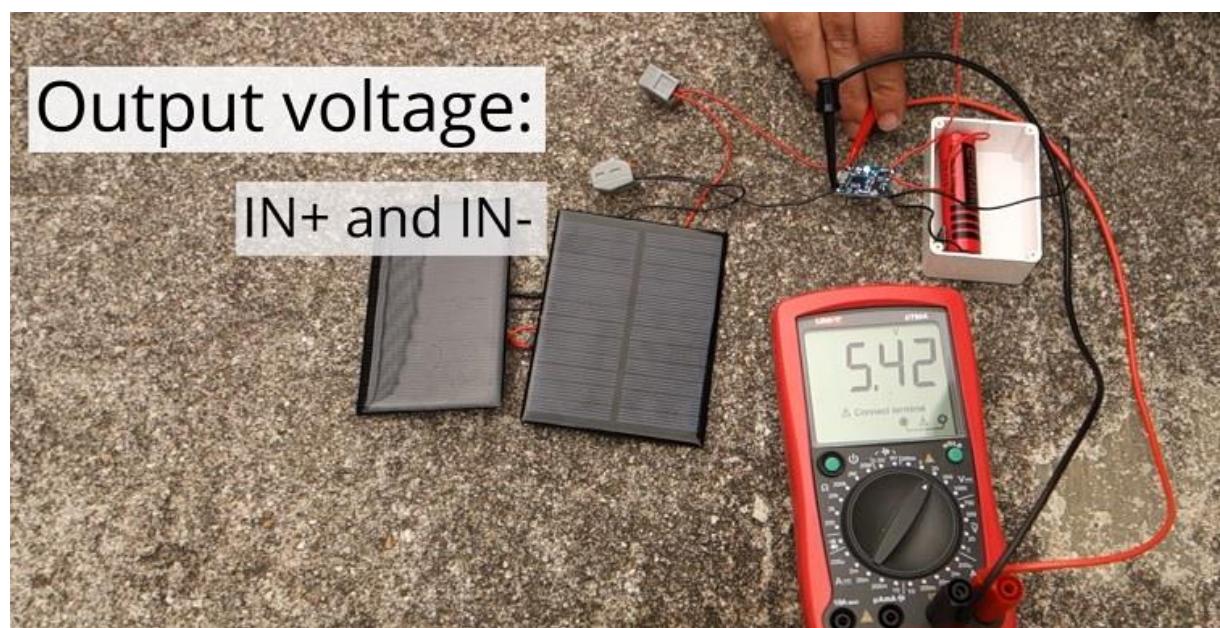


Warning: electrolytic capacitors have polarity! The lead with the white band should be connected to GND.

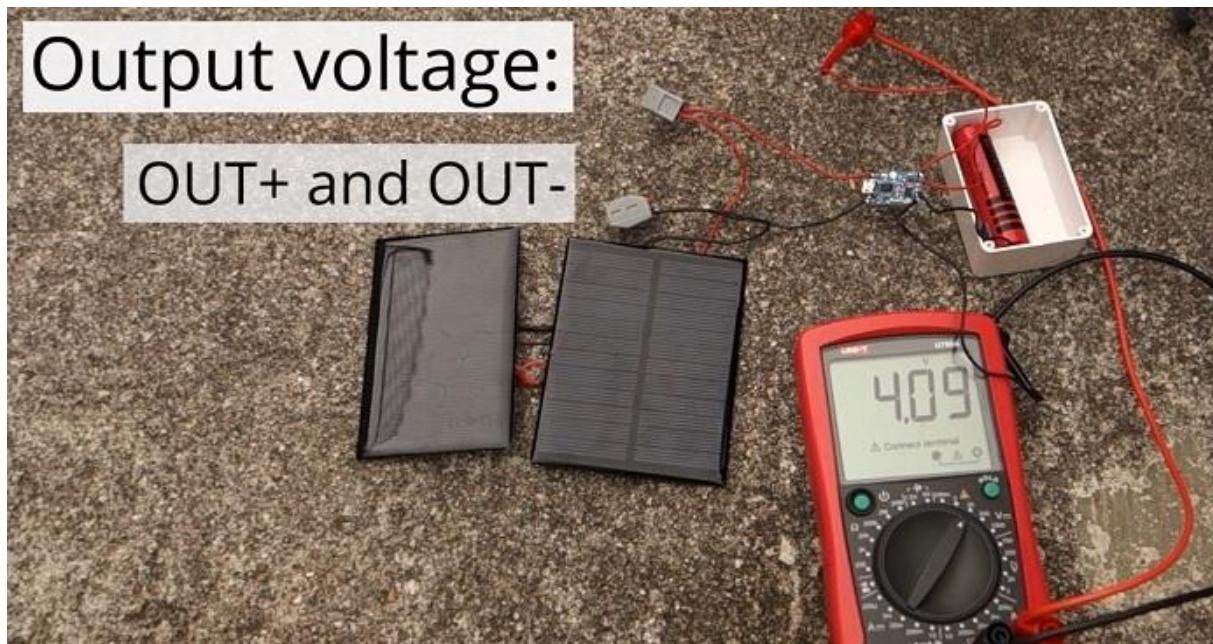
The Vout pin of the voltage regulator should output 3.3V. That is the pin that will be powering the ESP32.

Measuring Output Voltages

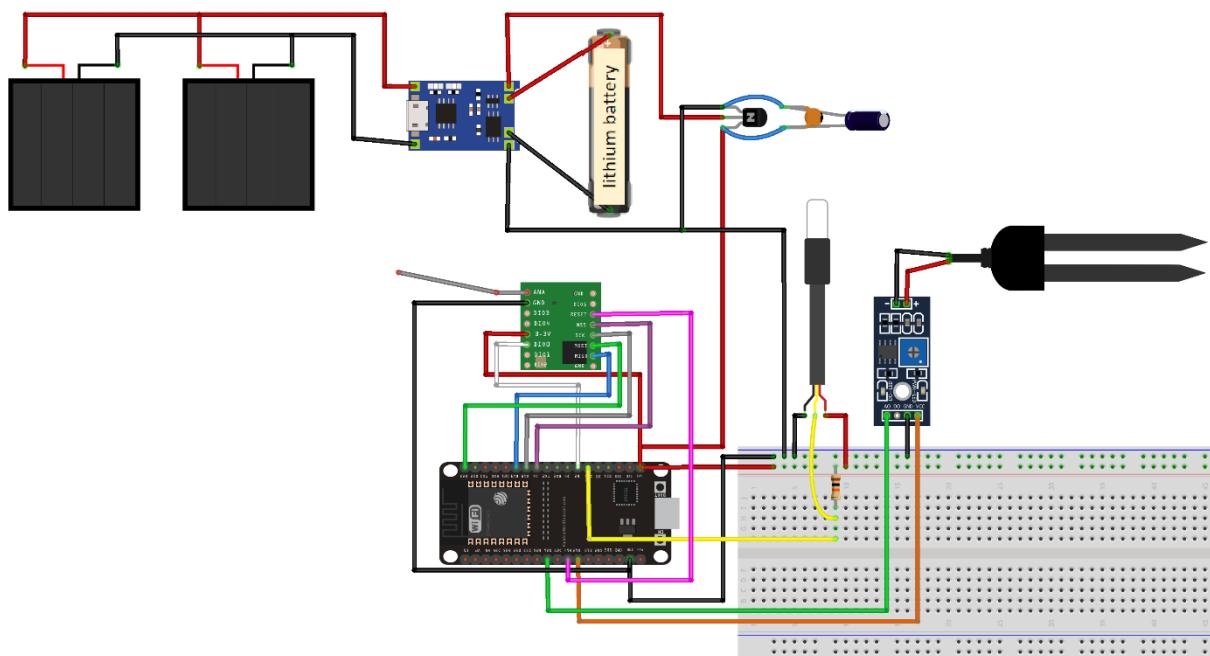
Now, let's test if everything is working properly. Start by measuring the output voltage of the IN+ and IN- pads. It should output approximately 5V, which is the output voltage of the solar panels.



If the batteries are fully charged, the OUT+ and OUT- should output approximately 4.2V.



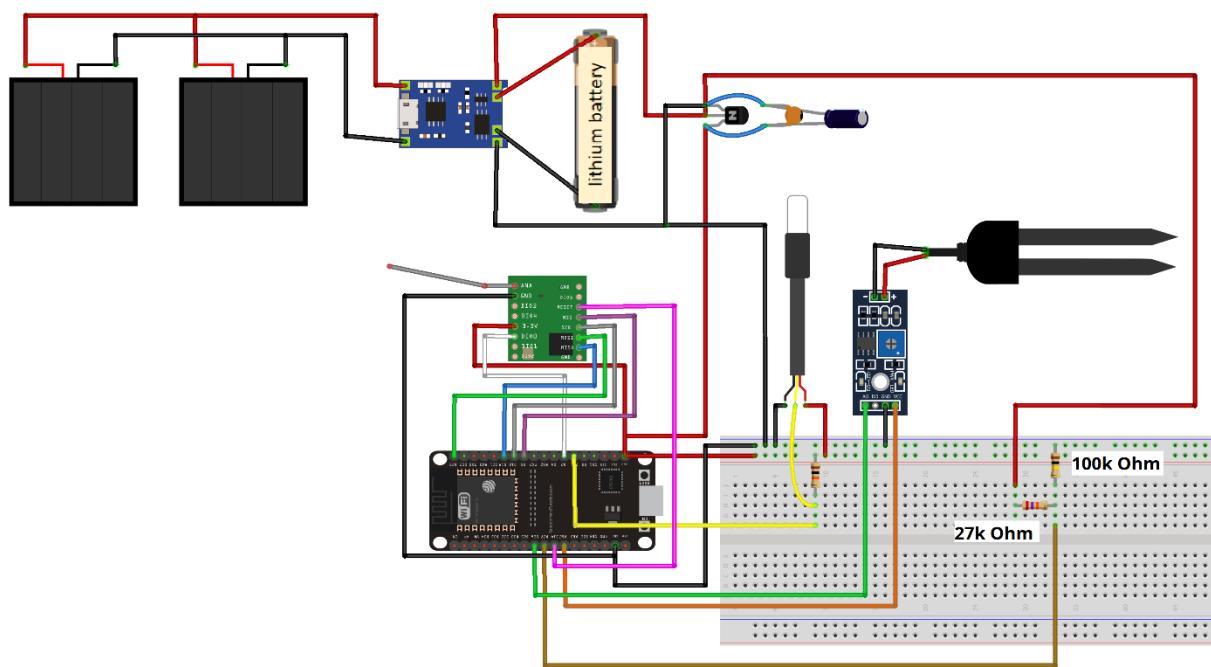
Those 4.2V will be connected to the voltage regulator circuit. That will output approximately 3.3V to power the ESP32 board. While having the lithium battery removed, connect all those components to the ESP32 as shown in this schematic diagram.



Battery Voltage Level Monitoring Circuit

Now, let's prepare the battery voltage level monitoring circuit. For this circuit, you need a voltage divider, because a Lithium battery outputs a maximum of 4.2V when fully charged, and the ESP32 analog pins can only handle up to 3.3V.

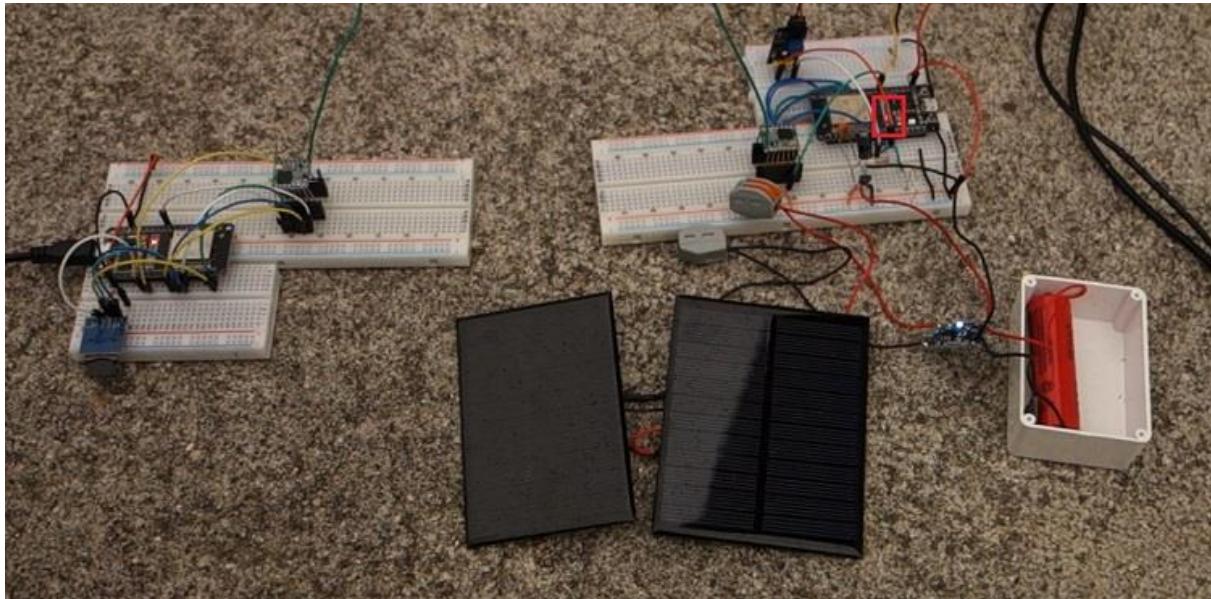
Add two resistors to create a voltage divider: 27k Ohm resistor and 100k Ohm resistor connected to GPIO 27. The LoRa sender code is already prepared to read the battery level on GPIO 27. So, you just need to add the battery voltage level monitoring circuit as shown in the following schematic diagram.



Testing the Solar Powered LoRa Sender

Finally, your LoRa Sender circuit is ready to be tested.

Grab your LoRa Sender with the solar power circuitry. Go outside where your solar panels can get direct sunlight. Having this circuit fully assembled, connect the lithium battery to the battery holder. The ESP32 should be powered on with the red LED on.



Note: the TP4056 module lights up a red LED when it's charging the battery and lights up a blue LED when the battery is fully charged.

Also power LoRa receiver circuit with the Arduino IDE serial monitor open. Press the ESP32 LoRa Sender Enable button and check if the receiver is picking up the LoRa packets.

```
COM4
Connecting to MEO-620B4B
.....
WiFi connected.
IP address:
192.168.1.141
Initializing SD card...
File doesn't exist
Creating file...
Writing file: /data.txt
File written
Lora packet received: 0/17.94&4095#100.00 with RSSI -35
2018-05-29T17:43:18Z
2018-05-29
17:43:18
Appending to file: /data.txt
Message appended
Lora packet received: 0/17.94&4095#100.00 with RSSI -33
2018-05-29T17:43:23Z
2018-05-29
17:43:23
Appending to file: /data.txt
Message appended
Lora packet received: 0/17.94&4095#100.00 with RSSI -35
2018-05-29T17:43:29Z
2018-05-29
17:43:29
Appending to file: /data.txt
Message appended
```

Autoscroll No line ending 115200 baud Clear output

Let your circuit run for a few hours to check if everything is working.

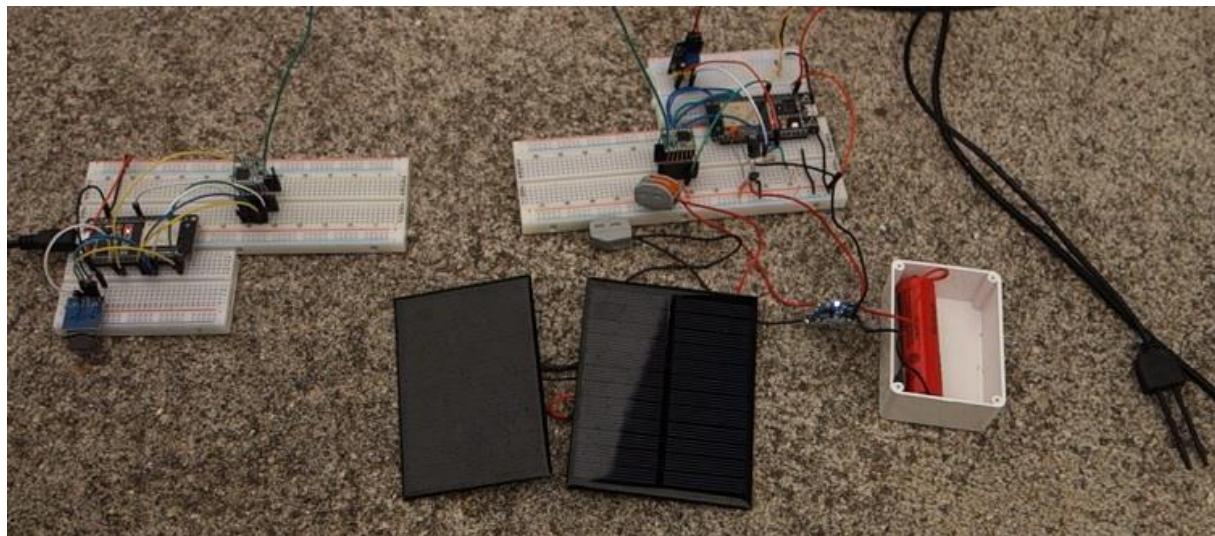
Congratulations, your project is finally completed!

Continue To The Next Unit ...

Proceed to the next Unit to see the final setup, demonstration, and tests.

Unit 5 - Final Tests, Demonstration, and Data Analysis

At this point, you should have your LoRa sender node equipped with a solar panel and the LoRa receiver circuit.



Parts Required:

Parts required for this unit:

- [Project box enclosure \(IP65/IP67\)](#)

Adding an Enclosure

We'll leave both circuits in a breadboard for this project, but you can use some stripboards to make a permanent circuit.

We'll put the LoRa sender circuit in an IP65 enclosure because the circuit will be placed outside.



This kind of enclosure is rated as "dust tight" and protected against water projected from a nozzle.

Note: depending on your weather conditions, you might consider using a waterproof enclosure (IP67 or IP68). The following chart may help you decide your enclosure rating.

IP (Ingress Protection) Ratings Guide

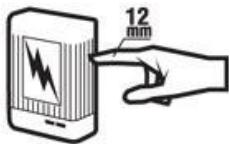
SOLIDS

1



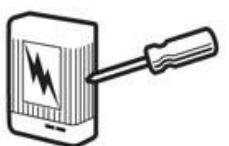
Protected against a solid object greater than 50 mm such as a hand.

2



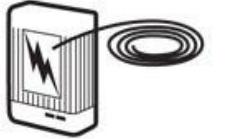
Protected against a solid object greater than 12.5 mm such as a finger.

3



Protected against a solid object greater than 2.5 mm such as a screwdriver.

4



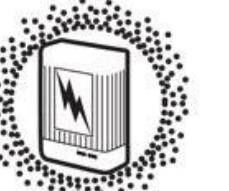
Protected against a solid object greater than 1 mm such as a wire.

5



Dust Protected.
Limited ingress of dust permitted. Will not interfere with operation of the equipment.
Two to eight hours.

6



Dust tight.
No ingress of dust.
Two to eight hours.

Rating Example:

IP65

INGRESS PROTECTION

WATER

1



Protected against vertically falling drops of water.
Limited ingress permitted.

2



Protected against vertically falling drops of water with enclosure tilted up to 15 degrees from the vertical.
Limited ingress permitted.

3



Protected against sprays of water up to 60 degrees from the vertical.
Limited ingress permitted for three minutes.

4



Protected against water splashed from all directions.
Limited ingress permitted.

5



Protected against jets of water.
Limited ingress permitted.

6



Water from heavy seas or water projected in powerful jets shall not enter the enclosure in harmful quantities.

7



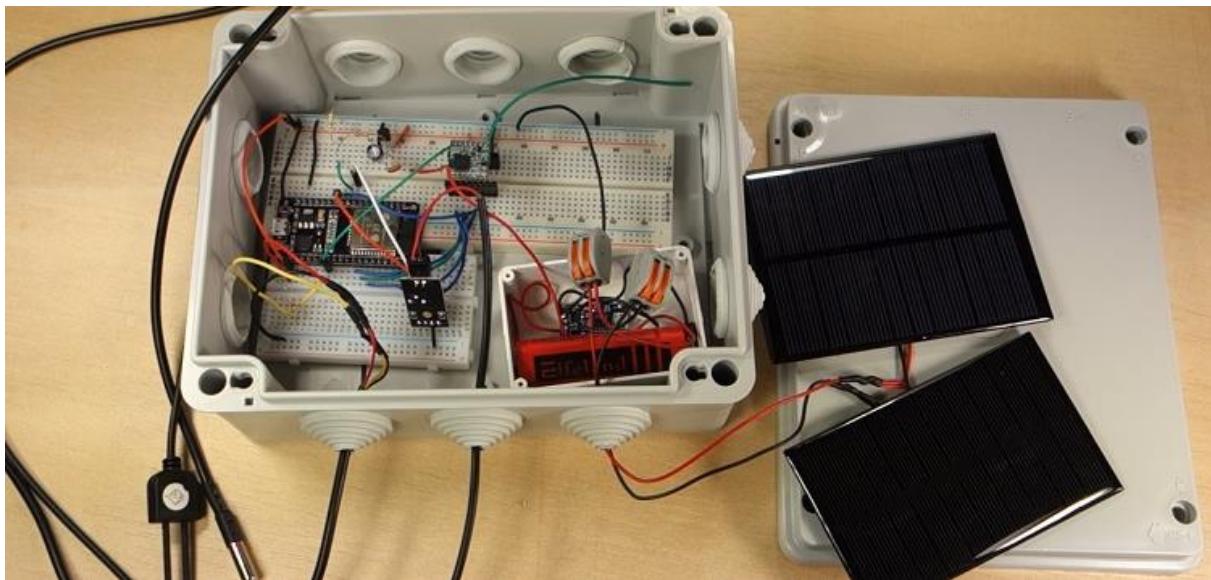
Protection against the effects of immersion in water between 15 cm and 1 m for 30 minutes.

8



Protection against the effects of immersion in water under pressure for long periods.

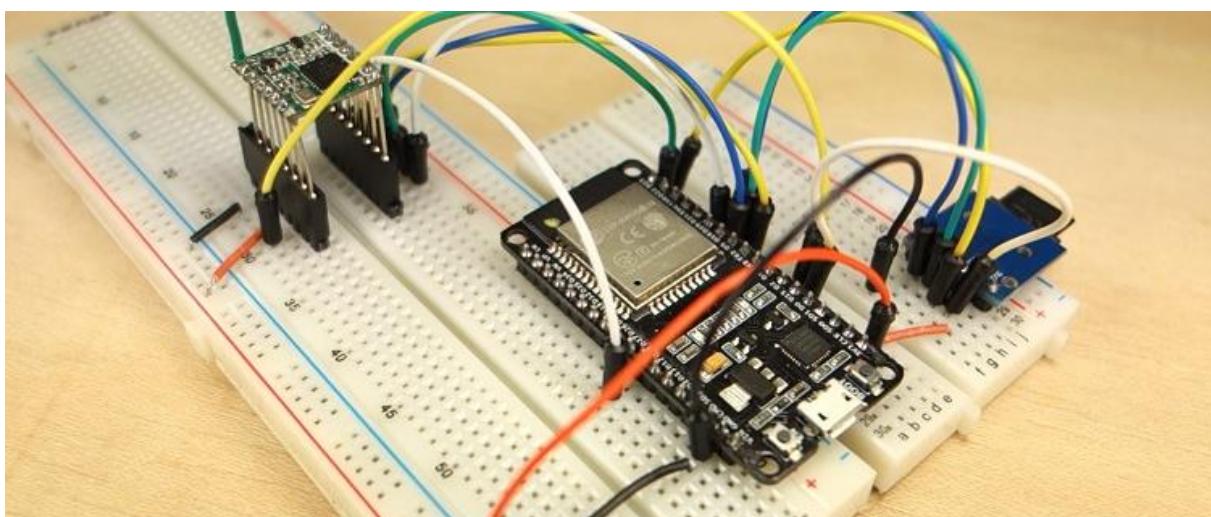
Here's how our enclosure looks like with all the circuitry inside.



You can use some hot glue to fix the breadboard at the bottom. The solar panels are fixed at the top of the lid. After having everything prepared, choose a place on your field to monitor and close your enclosure. Your LoRa sender node is ready!



The LoRa receiver will be sitting indoors, so we don't need an enclosure.

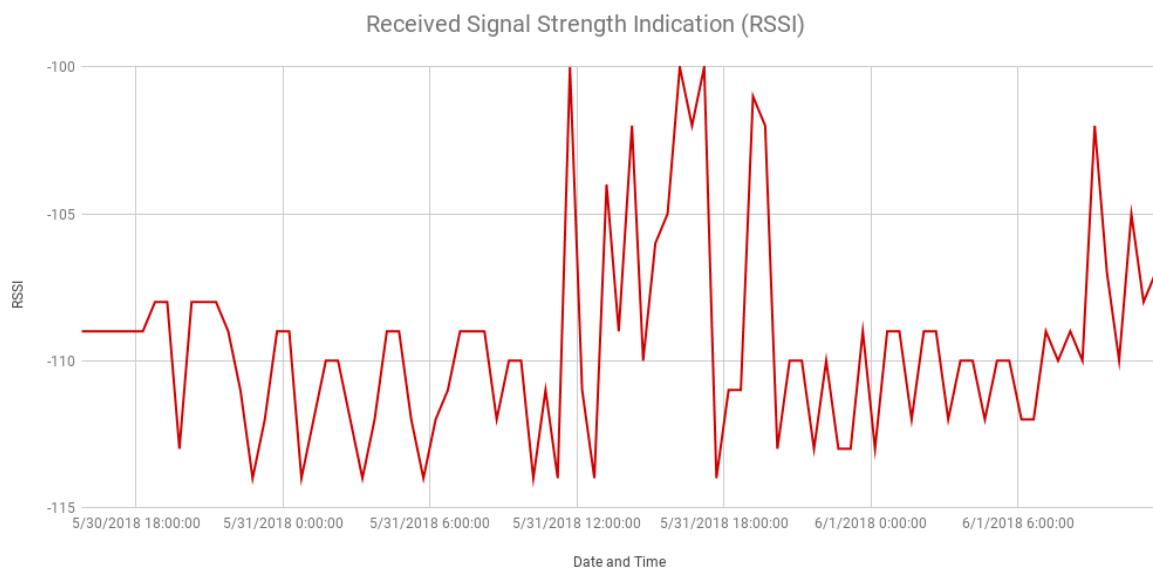


Communication Range

We've placed the LoRa sender in our field at approximately 130 meters from the receiver and it received the messages reliably.



At this distance, we get the messages with an RSSI of approximately -110 (see the chart below). But in our environment, we were able to get a good communication of up to 250 meters distance.



During our tests, at this distance, the receiver was able to get all the LoRa messages.

Power Consumption

The solar panels with the lithium batteries were able to power up the LoRa sender circuit even on foggy and rainy days.

We've performed some tests, and even after 48 hours without light, the circuit was still working. We've found that if the lithium battery is almost discharged, it takes approximately 2 hours to completely charge with bright sun.

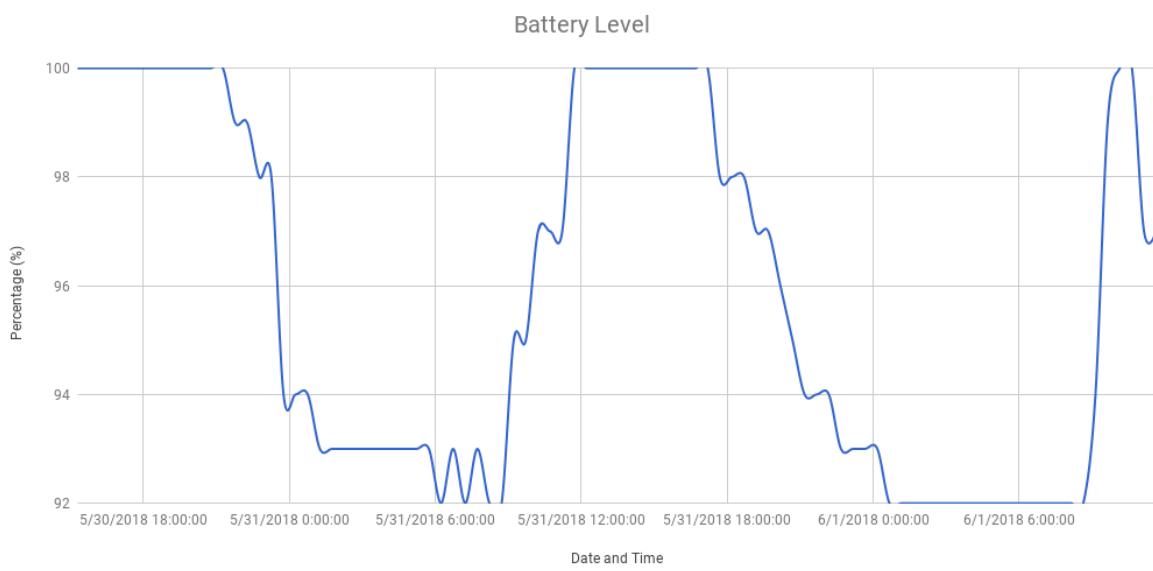
To charge your battery faster, you can add more solar panels in parallel.



Our battery has 3800mAh. If you want to ensure you have enough power for several days without sun, you can use a lithium battery with more capacity.



With the collected data, we can clearly see the battery behavior throughout the day.



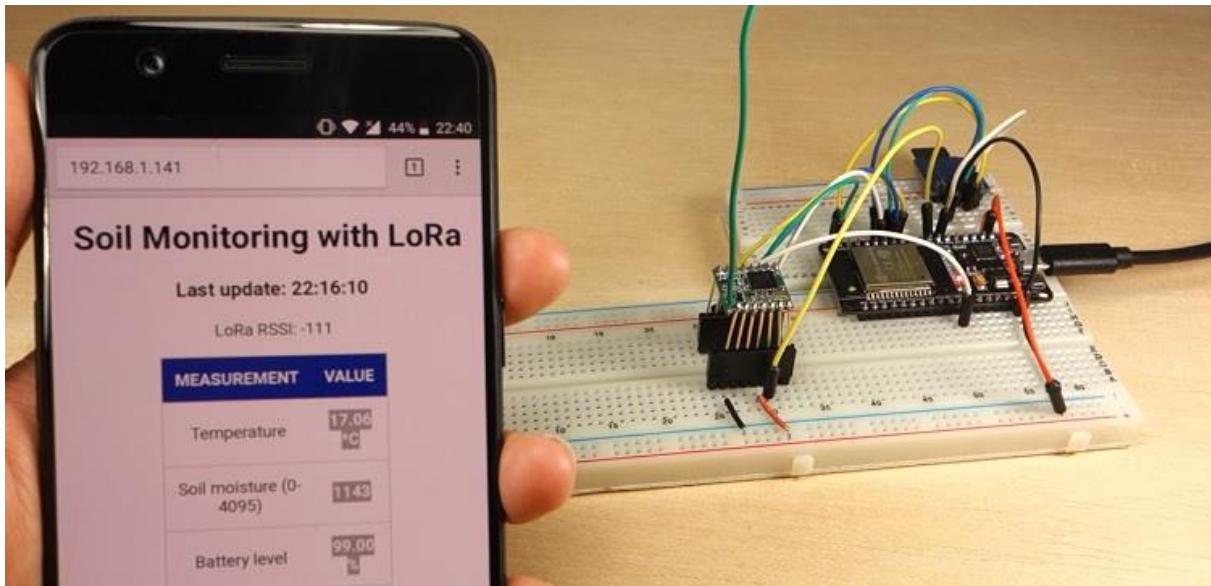
During the night, the battery discharges to about 92%. At 6:00 AM, when the sun rises, the battery starts charging. At 12:00 the battery is fully charged and keeps fully charged until 18:00 (6PM). After that, it starts decreasing again. Throughout these experimenting days, the battery never reached less than 92%. The weather on these days was cloudy with light showers.



Testing the Web Server

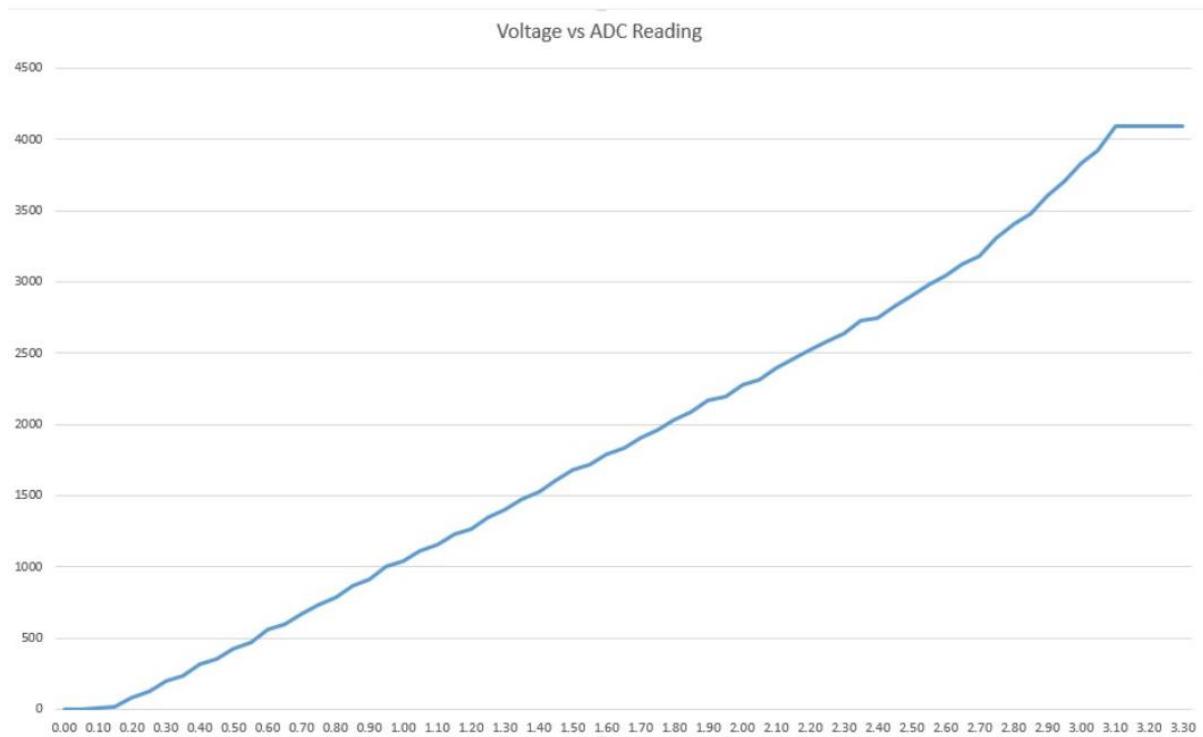
Every time you want to check the latest sensor readings, go to the ESP32 IP address to access the web server.

You can see when the latest sensor readings were taken, the soil moisture, temperature, battery level, and RSSI.



Battery Level

The battery level monitoring circuit gives you an idea of the current battery voltage level. However, you should keep in mind that the ESP32 analog pins don't have a linear behavior. Instead, they work as shown in the following chart.



[View image source](#)

This means that your ESP32 is not able to know the difference between 3.3V and 3.2V. You'll get the same value for both voltages: 4095, which means 100% battery. The ESP32 stops working when the battery goes below 75%.

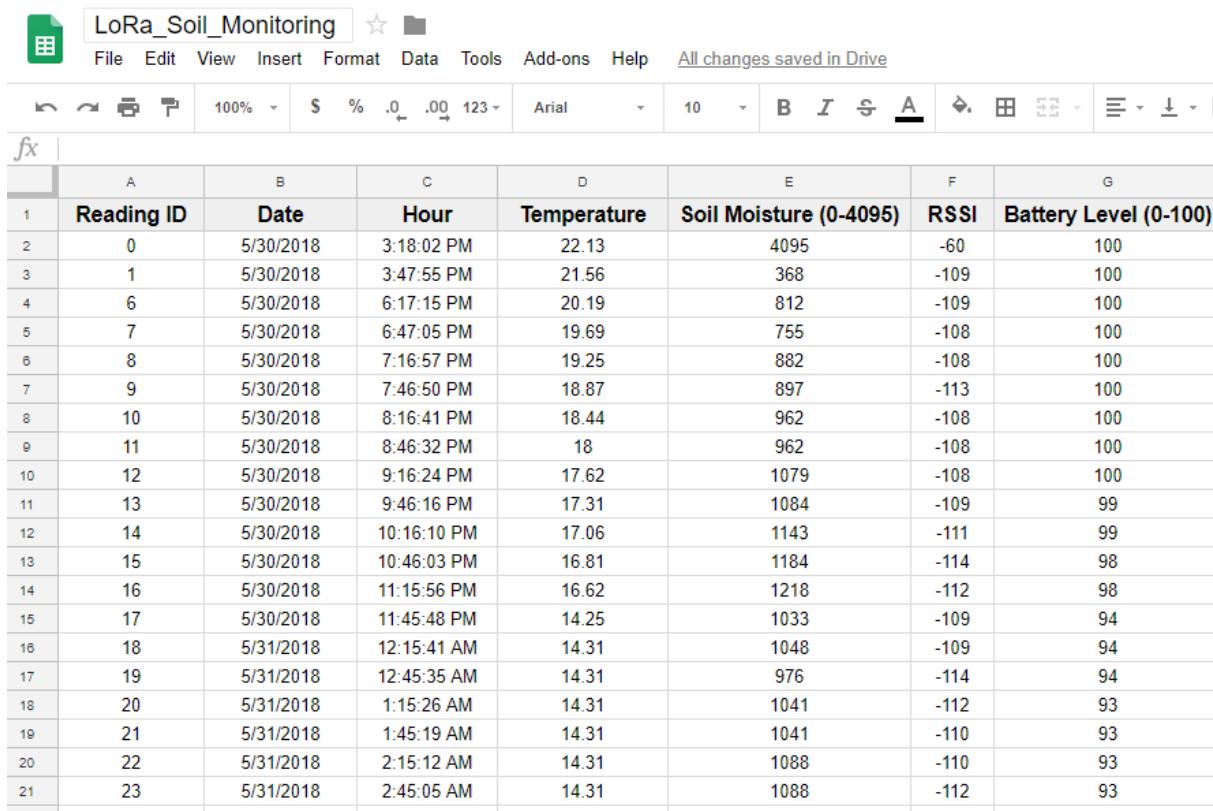
After checking the readings, close your browser to ensure that your receiver gets the LoRa packets.

Soil Moisture

We receive the soil moisture as a value from 0 to 4095, in which 0 corresponds to totally wet, and 4095 to totally dry. You can do some math to convert these values to a percentage that may make more sense for what you want to do.

Accessing the MicroSD Card Data

After having the setup running for a few days, you can collect the data from the microSD card. Insert the microSD card on your computer, and you should have the data.txt file. You can copy the file content to a spreadsheet on Google Sheets, for example, and then split the data by commas. To split data by commas, select the column where you have your data, then go to **Data > Split by comma**.

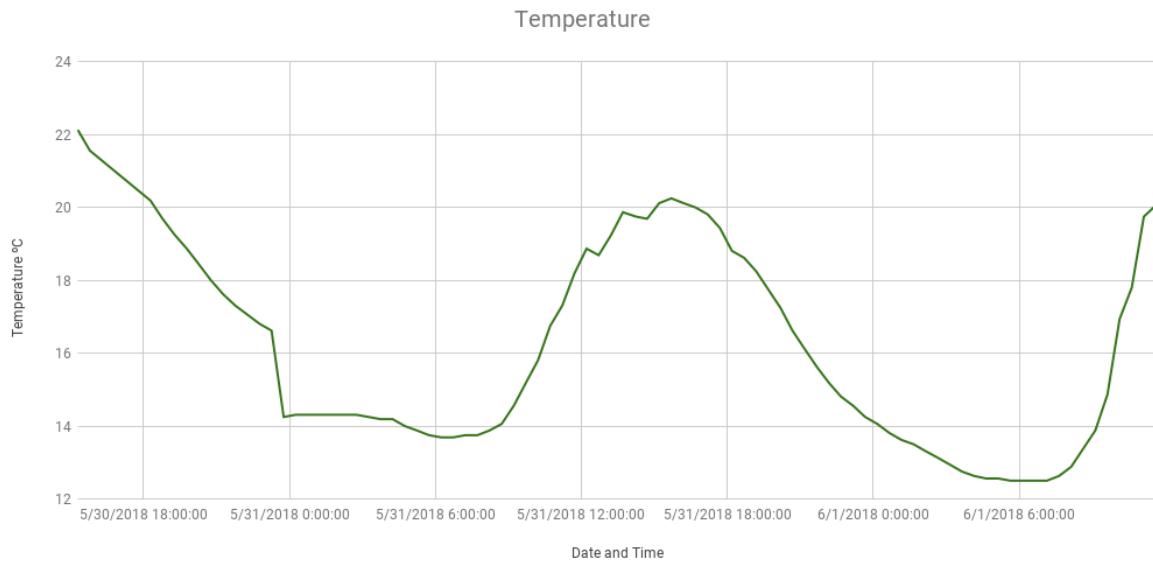


The screenshot shows a Google Sheets interface with the title "LoRa_Soil_Monitoring". The table has the following structure:

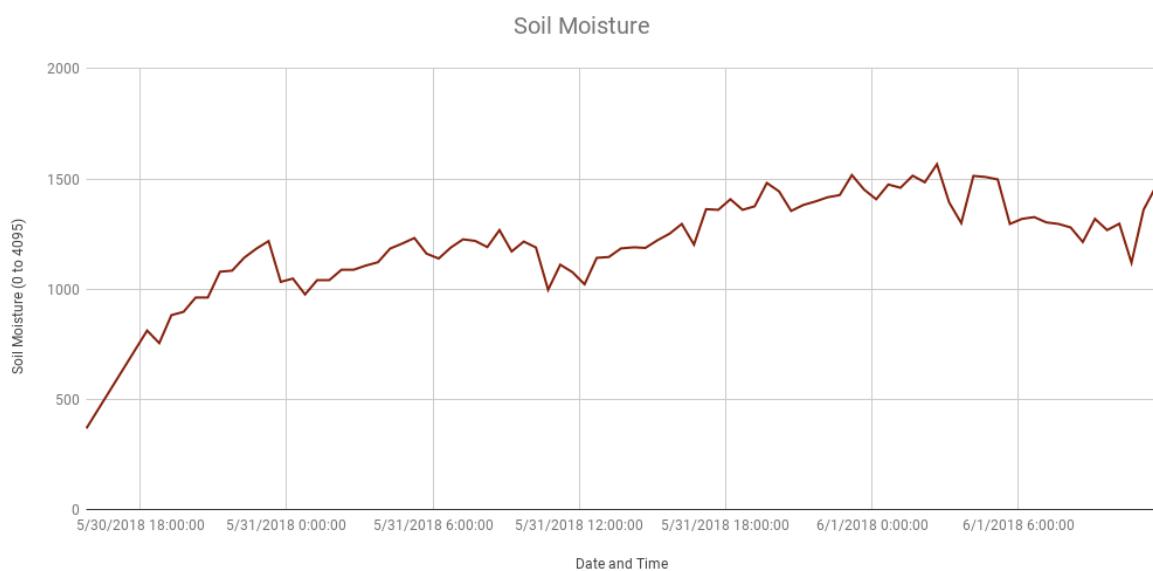
	A	B	C	D	E	F	G
1	Reading ID	Date	Hour	Temperature	Soil Moisture (0-4095)	RSSI	Battery Level (0-100)
2	0	5/30/2018	3:18:02 PM	22.13	4095	-60	100
3	1	5/30/2018	3:47:55 PM	21.56	368	-109	100
4	6	5/30/2018	6:17:15 PM	20.19	812	-109	100
5	7	5/30/2018	6:47:05 PM	19.69	755	-108	100
6	8	5/30/2018	7:16:57 PM	19.25	882	-108	100
7	9	5/30/2018	7:46:50 PM	18.87	897	-113	100
8	10	5/30/2018	8:16:41 PM	18.44	962	-108	100
9	11	5/30/2018	8:46:32 PM	18	962	-108	100
10	12	5/30/2018	9:16:24 PM	17.62	1079	-108	100
11	13	5/30/2018	9:46:16 PM	17.31	1084	-109	99
12	14	5/30/2018	10:16:10 PM	17.06	1143	-111	99
13	15	5/30/2018	10:46:03 PM	16.81	1184	-114	98
14	16	5/30/2018	11:15:56 PM	16.62	1218	-112	98
15	17	5/30/2018	11:45:48 PM	14.25	1033	-109	94
16	18	5/31/2018	12:15:41 AM	14.31	1048	-109	94
17	19	5/31/2018	12:45:35 AM	14.31	976	-114	94
18	20	5/31/2018	1:15:26 AM	14.31	1041	-112	93
19	21	5/31/2018	1:45:19 AM	14.31	1041	-110	93
20	22	5/31/2018	2:15:12 AM	14.31	1088	-110	93
21	23	5/31/2018	2:45:05 AM	14.31	1088	-112	93

Here you have all the data collected: reading ID, date, hour, temperature, soil moisture, RSSI, and battery level.

You can use this data to create charts. Here's our temperature data throughout the experimenting period.



And here are the soil moisture results.



Wrapping Up

That's it for the LoRa Long Range Sensor Monitoring project.



Here are some of the new concepts learned:

- Making your ESP32 solar-powered;
- Powering the ESP32 with lithium batteries;
- Data logging to an SD card;
- Requesting time from an NTP server;
- And much more...

We hope you've found this project useful and you can apply it to monitor your own field.

Final Thoughts

Congratulations on completing this course!

If you followed all the Modules presented in this course, now you master the ESP32. Let's see the most important concepts that you've learned. You know how to:

- Use the ESP32 GPIOs: analog inputs, PWM, digital outputs, touch pins, interrupts, and much more;
- Take advantage of the ESP32 deep sleep capabilities to build low power consumption circuits and projects;
- Create a password-protected web server to control any output and display sensor readings;
- Customize your web server using HTML and CSS;
- Make your web server accessible from anywhere in the world;
- Use the BLE capabilities of the ESP32, notify and scan, and create BLE servers and clients;
- Use LoRa technology to extend the communication range between two ESP32;
- Use MQTT to communicate between different devices and how to use Node-RED to control the ESP32;
- Communicate between multiple ESP32 boards using ESP-NOW communication protocol;
- Build more advanced projects with the concepts learned throughout the course.

We hope you had fun following this course! If you have something that you would like to share, let us know in the Facebook group ([Join the Facebook group here](#)).

Good luck with all your projects,

Rui Santos and Sara Santos

List of Parts Required

Parts Required for Module 1 to Module 10

Here's a quick overview of the components and tools used throughout the "Learn ESP32 with Arduino IDE" course. We don't recommend purchasing all the items at once. Instead, we recommend going through each project, checking exactly what you want to build, and getting the components accordingly.

Important: all components and parts are linked to our website [MakerAdvisor.com/tools](https://www.MakerAdvisor.com/tools) where you can compare prices in more than eight different stores, so you can always find the best price.

- [3x ESP32 DOIT DEVKIT V1 Board](#)
- [3x 5mm LED](#)
- [3x 330 Ohm resistor](#)
- [2x pushbutton](#)
- [2x 10k Ohm resistor](#)
- [10K Ohm potentiometer](#)
- [Mini PIR motion sensor \(AM312\)](#) or [PIR motion sensor \(HC-SR501\)](#)
- [BME280 sensor module](#)
- [DHT11/DHT22](#) temperature and humidity sensor
- [4.7k Ohm resistor](#)
- [OLED display](#)
- [2x LoRa Transceiver modules \(RFM95\)](#)
- 2x RFM95 LoRa breakout board (optional)
- [Jumper wires](#)
- [2x Breadboard](#)
- [Stripboard](#)
- Smartphone with Bluetooth Low Energy (BLE)
- [DS18B20 temperature sensor](#)
- [I2C 16x2 LCD](#)
- [Raspberry Pi board](#)
- [MicroSD Card – 32GB Class10](#)
- [Raspberry Pi Power Supply \(5V 2.5A\)](#)

Parts Required for Projects

Project #1 - ESP32 Wi-Fi Multisensor:

- [ESP32 DOIT DEVKIT V1 board](#)
- [DHT22 temperature and humidity sensor](#)
- [Mini PIR motion sensor](#)
- [Light dependent resistor \(LDR\)](#)
- [RGB LED common anode](#)
- [2x 10K Ohm resistor](#)
- [3x 220 Ohm resistor](#)
- [1x LED holder](#)
- [Relay module](#)
- 12V lamp
- [12V power source](#)
- [Breadboard](#)
- [Prototyping circuit board](#)
- [Jumper wires](#)
- [Project box enclosure](#) (or 3D print your own case)
- Useful tools:
 - [Soldering iron](#)
 - [Hot glue gun](#)
 - [3D printer](#)

Project #2 - ESP32 Wi-Fi Car Robot:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [Smart Robot Chassis Kit](#) (or your own DIY robot chassis + 2x [DC motors](#))
- [L298N motor driver](#)
- [1x Power bank – portable charger](#)
- [4x 1.5 AA batteries](#)
- [2x 100nF ceramic capacitors](#)
- [1x SPDT Slide Switch](#)
- [Jumper wires](#)
- [Breadboard](#) or [stripboard](#)
- [Velcro tape](#)

Project #3 - ESP32 BLE Android Application:

- [ESP32 DOIT DEVKIT V1 board](#)
- [DS18B20 temperature sensor](#)
- [10K Ohm resistor](#)
- [5mm LED](#)
- [220 Ohm resistor](#)

- [Jumper wires](#)
- [Breadboard](#)
- Smartphone with Bluetooth Low Energy (BLE)

Project #4 - ESP32 LoRa Long Range Sensor Monitoring:

LoRa Sender

- [ESP32 DOIT DEVKIT V1 board](#)
- [RFM95 LoRa transceiver module](#)
- RFM95 LoRa breakout board (optional)
- Temperature sensor:
 - [DS18B20 temperature sensor \(waterproof version\)](#)
 - [10K Ohm resistor](#)
 - [Resistive Soil Moisture sensor](#)
- Power source and charger:
 - [Lithium Li-ion battery \(at least 3800mAh capacity\)](#)
 - Battery holder
 - [Battery charger \(optional\)](#)
 - [TP4056 Lithium Battery Charger](#)
 - [2x Mini Solar Panel \(5V 1.2W\)](#)
- Battery voltage level monitor: [27K Ohm resistor + 100K Ohm resistor](#)
- Voltage regulator:
 - [Low-dropout or LDO regulator \(MCP1700-3320E\)](#)
 - [100uF electrolytic capacitor](#)
 - [100nF ceramic capacitor](#)
- [2x Breadboards](#)
- [Jumper Wires](#)
- [Project box enclosure \(IP65/IP67\)](#)

LoRa Receiver

- [ESP32 DOIT DEVKIT V1 board](#)
- [RFM95 LoRa transceiver module](#)
- RFM95 LoRa breakout board (optional)
- [MicroSD card module](#)
- [MicroSD card](#)
- [2x Breadboards](#)
- [Jumper Wires](#)

Useful tools:

- [Soldering Iron](#)
- [Hot glue gun](#)
- [Multimeter](#)

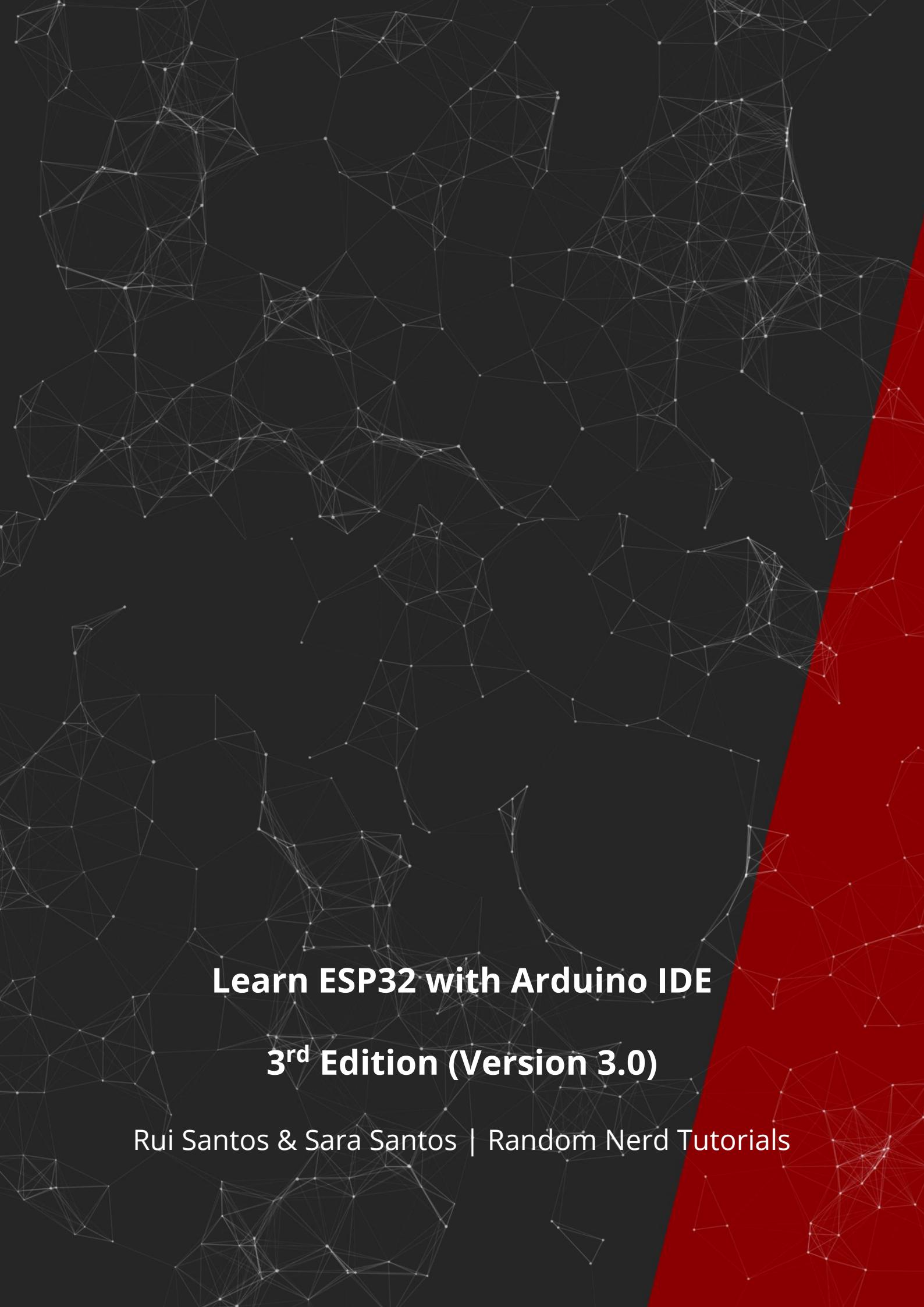
Other RNT Courses/eBooks

[Random Nerd Tutorials](#) is an online resource with electronics projects, tutorials, and reviews. Currently, Random Nerd Tutorials has more than [700 free blog posts](#) with complete tutorials using open-source hardware that anyone can read, remix, and apply to their projects.

To keep free tutorials coming, there is also paid content or what we like to call "premium content". To support Random Nerd Tutorials, you can [download premium content here](#). If you enjoyed this eBook, make sure you [check all our courses and resources](#).

Here's a list of all the eBooks available to download at Random Nerd Tutorials:

- [SMART HOME with Raspberry Pi, ESP32, and ESP8266](#) (Bestseller)
- [Learn LVGL: Build GUIs for ESP32 Projects](#)
- [Build Web Servers using ESP32 and ESP8266](#)
- [MicroPython Programming with ESP32/ESP8266](#)
- [Learn Raspberry Pi Pico with MicroPython](#)
- [Firebase Web App with the ESP32 and ESP8266](#)
- [Build ESP32-CAM Projects using Arduino IDE](#)
- [Home Automation Using ESP8266](#)
- [Arduino Step-by-step Projects Course](#)
- [Android Apps for Arduino with MIT App Inventor 2](#)
- [Electronics For Beginners eBook](#)



Learn ESP32 with Arduino IDE

3rd Edition (Version 3.0)

Rui Santos & Sara Santos | Random Nerd Tutorials