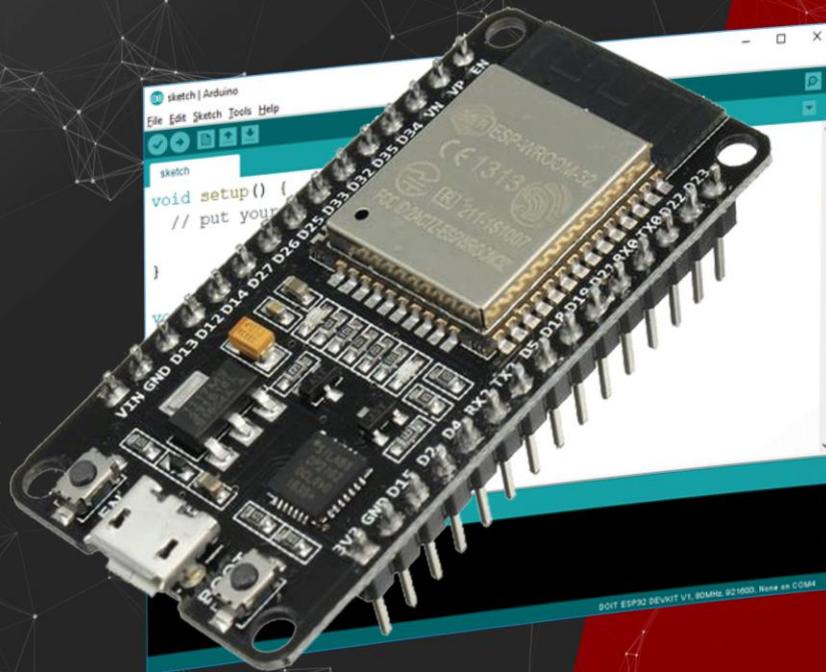


2nd Edition

LEARN ESP32

with Arduino IDE

LEARN ESP32 with Arduino IDE



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

Rui Santos and Sara Santos

2nd Edition

Security Notice

This is the kind of thing I hate having to write about, but the evidence is clear: piracy for digital products is over all the internet.

For that reason I've taken certain steps to protect my 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 probably won't see anything different, since those strings are hidden in this PDF. I apologize for having to do that – but it means if someone were to share this eBook I know exactly who shared it and I 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>

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

What I really want to say is thank you for purchasing this eBook and I hope you have fun with it!

Disclaimer

This eBook has been written for information purposes only. Every effort has been 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 of those products, and we recommend them because they are useful, not because of the small commissions we made if you decide to buy something. Please do not spend any money on these products unless you feel you need them.

Other Helpful Links:

- [Join Private Facebook Group](#)
- [Terms and Conditions](#)

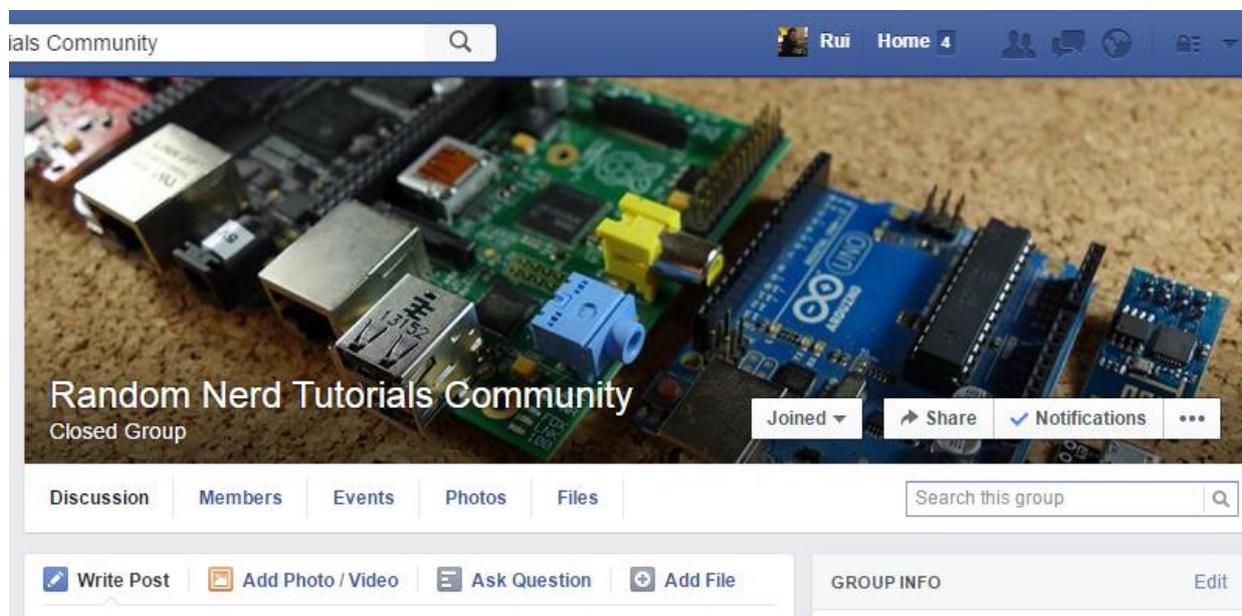
Join the Private Facebook Group

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

Inside the group you can ask questions and create discussions about everything related to ESP32, ESP8266, Arduino, Raspberry Pi, BeagleBone, etc.

See it for yourself!

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



About the Authors

This course was built, developed, and written by Rui Santos and Sara Santos. We both live in Porto, Portugal, and we know each other since 2009. If you want to learn more about us, feel free to 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. I'm the author of "BeagleBone For Dummies", and Technical reviewer of the book "Raspberry Pi For Kids For Dummies". I wrote a book with Sara Santos for the [NoStarchPress publisher](#) about projects with the Raspberry Pi: "[20 Easy Raspberry Pi Projects: Toys, Tools, Gadgets, and More!](#)"



Hi! I'm Sara Santos! I started working at Random Nerd Tutorials back in 2015 as a hobby: I helped Rui with some simple tasks when he had a lot of work to do. Back then, I knew nothing about electronics, programming, Arduino, etc... Over time I started learning everything I could about those subjects and I just loved it! At first, I helped Rui once a week on Saturdays, but then, I started working on the RNT blog alongside him, almost every day! Currently, I work full time at Random Nerd Tutorials and I love what I do!

Table of Contents

MODULE 0

Course Intro 10

Welcome to Learn ESP32 with Arduino IDE 11

MODULE 1

Getting Started with ESP32 20

Unit 1 - Introducing ESP32 21

Unit 2 - Installing ESP32 in Arduino IDE (Windows, Mac OS X, and Linux)..... 27

Unit 3 - How to Use Your ESP32 Board with this Course 35

Unit 4 - ESP32 Breadboard Friendly 45

MODULE 2

Exploring the ESP32 GPIOs..... 47

Unit 1 - ESP32 Digital Inputs and Outputs..... 48

Unit 2 - ESP32 Touch Sensor 53

Unit 3 - ESP32 Pulse-Width Modulation (PWM) 60

Unit 4 - ESP32 Reading Analog Inputs..... 67

Unit 5 - ESP32 Hall Effect Sensor 72

Unit 6 - ESP32 with PIR Motion Sensor: Interrupts and Timers 76

Unit 7 - ESP32 Flash Memory – Store Permanent Data (Write and Read) 86

Unit 8 - Other ESP32 Sketch Examples 94

MODULE 3

ESP32 Deep Sleep..... 95

Unit 1 - ESP32 Deep Sleep Mode 96

Unit 2 - Deep Sleep – Timer Wake Up 100

Unit 3 – Deep Sleep Touch Wake Up..... 106

Unit 4 - Deep Sleep External Wake Up..... 111

MODULE 4

ESP32 Web Servers..... 124

Unit 1 - Web Server Introduction..... 125

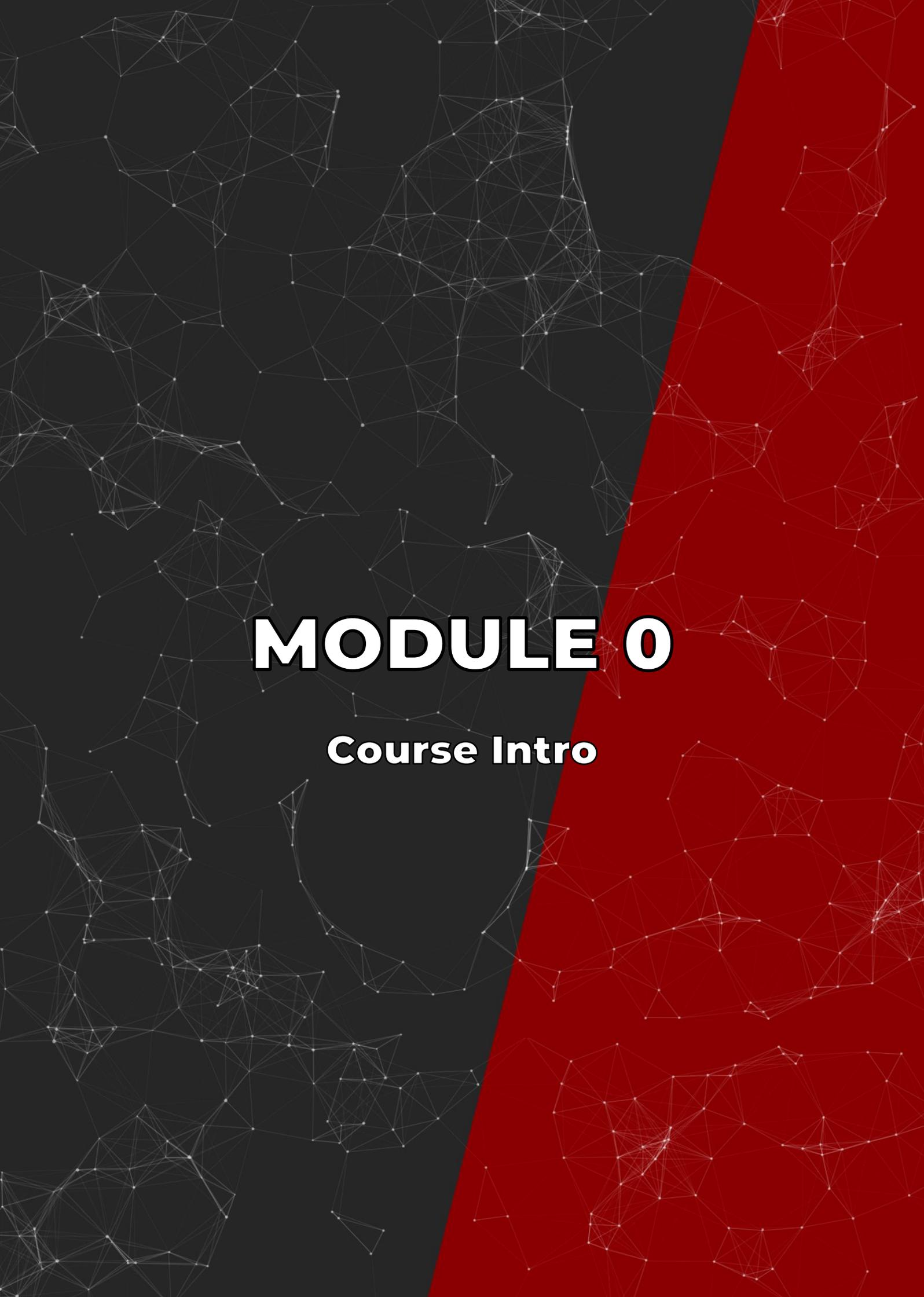
Unit 2 - Web Server – Control Outputs 130

Unit 3 - ESP32 Web Server – HTML and CSS Basics (Part 1/2) 142

Unit 4 - ESP32 Web Server – HTML in Arduino IDE (Part 2/2)	156
Unit 5 - ESP32 Web Server – Control Outputs (Relay).....	163
Unit 6 - Making Your ESP32 Web Server Password Protected	172
Unit 7 - Accessing the ESP32 Web Server from Anywhere.....	179
Unit 8 - ESP32 Web Server – Display Sensor Readings.....	188
Unit 9 - ESP32 Control Servo Motor Remotely (Web Server)	203
Unit 10 - Color Picker Web Server for RGB LED Strip.....	218
Unit 11 – Asynchronous Web Server: Temperature and Humidity Readings	228
Unit 12 – Asynchronous Web Server: Control Outputs	246
MODULE 5	
ESP32 Bluetooth.....	258
Unit 1 - ESP32 Bluetooth Low Energy (BLE) – Introduction.....	259
Unit 2 - Bluetooth Low Energy – Notify and Scan.....	267
Unit 3 - ESP32 BLE Server and Client (Part 1/2)	280
Unit 4 - ESP32 BLE Server and Client (Part 2/2)	292
Unit 5 - Bluetooth Classic.....	305
MODULE 6	
LoRa Technology with ESP32.....	319
Unit 1 - ESP32 with LoRa Introduction	320
Unit 2 - ESP32 – LoRa Sender and Receiver	328
Unit 3 - Further Reading about LoRa Gateways	348
Unit 4 - LoRa – Where to Go Next?	351
MODULE 7	
ESP32 with MQTT	352
Unit 1 - ESP32 with MQTT: Introduction	353
Unit 2 - Installing Mosquitto MQTT Broker on a Raspberry Pi	357
Unit 3 - MQTT Project: MQTT Client ESP32#1	360
Unit 4 - MQTT Project : MQTT Client ESP32 #2.....	375
Unit 5 - Installing Node-RED and Node-RED Dashboard on a RPi	387
Unit 6 - Connect ESP32 to Node-RED using MQTT	393
MODULE 8	
ESP-NOW Communication Protocol	409
Unit 1 – ESP-NOW: Getting Started.....	410

Unit 2 - ESP-NOW Two-Way Communication Between ESP32	424
Unit 3 - ESP-NOW Send Data to Multiple Boards (one-to-many)	436
Unit 4 - ESP-NOW Receive Data from Multiple Boards (many-to-one).....	448
Unit 5 - ESP-NOW Web Server Sensor Dashboard (ESP-NOW + Wi-Fi).....	459
PROJECT 1	
ESP32 Wi-Fi Multisensor: Temperature, Humidity, Motion, Luminosity, and Relay Control	483
Unit 1 - ESP32 Wi-Fi Multisensor: Temperature, Humidity, Motion, Luminosity, and Relay Control.....	484
Unit 2 - ESP32 Wi-Fi Multisensor: How the Code Works?.....	513
PROJECT 2	
Remote Controlled Wi-Fi Car Robot	531
Unit 1 - Remote Controlled Wi-Fi Car Robot (Part 1/2)	532
Unit 2 - Remote Controlled Wi-Fi Car Robot (Part 2/2)	543
Unit 3 - Assembling the Smart Robot Car Chassis Kit.....	557
Unit 4 - Access Point (AP) For Wi-Fi Car Robot	562
PROJECT 3	
ESP32 BLE Android Application	569
Unit 1 - ESP32 BLE Android Application: Control Outputs and Display Sensor Readings	570
Unit 2 - Bluetooth Low Energy (BLE) Android Application with MIT App Inventor 2: How the App Works?.....	584
PROJECT 4	
LoRa Long Range Sensor Monitoring and Data Logging.....	598
Unit 1 - LoRa Long Range Sensor Monitoring and Data Logging.....	599
Unit 2 - ESP32 LoRa Sender	606
Unit 3 - ESP32 LoRa Receiver	620
Unit 4 - LoRa Sender Solar Powered	638
Unit 5 - Final Tests, Demonstration, and Data Analysis.....	648
EXTRA UNITS	
ESP32 Static/Fixed IP Address	659
ESP32 Dual Core – Create Tasks	666
ESP32 SPIFFS (SPI Flash File System)	676
Build an ESP32 Web Server using Files from Filesystem (SPIFFS)	686

ESP32 Client-Server Wi-Fi Communication Between Two Boards.....	697
ESP32 HTTP GET (OpenWeatherMap and ThingSpeak).....	715
ESP32 HTTP POST (ThingSpeak and IFTTT.com).....	728
GUIDE FOR ESP32 GPIOs	
ESP32 Pinout Reference: Which GPIO pins should you use?.....	755
List of Parts Required	765

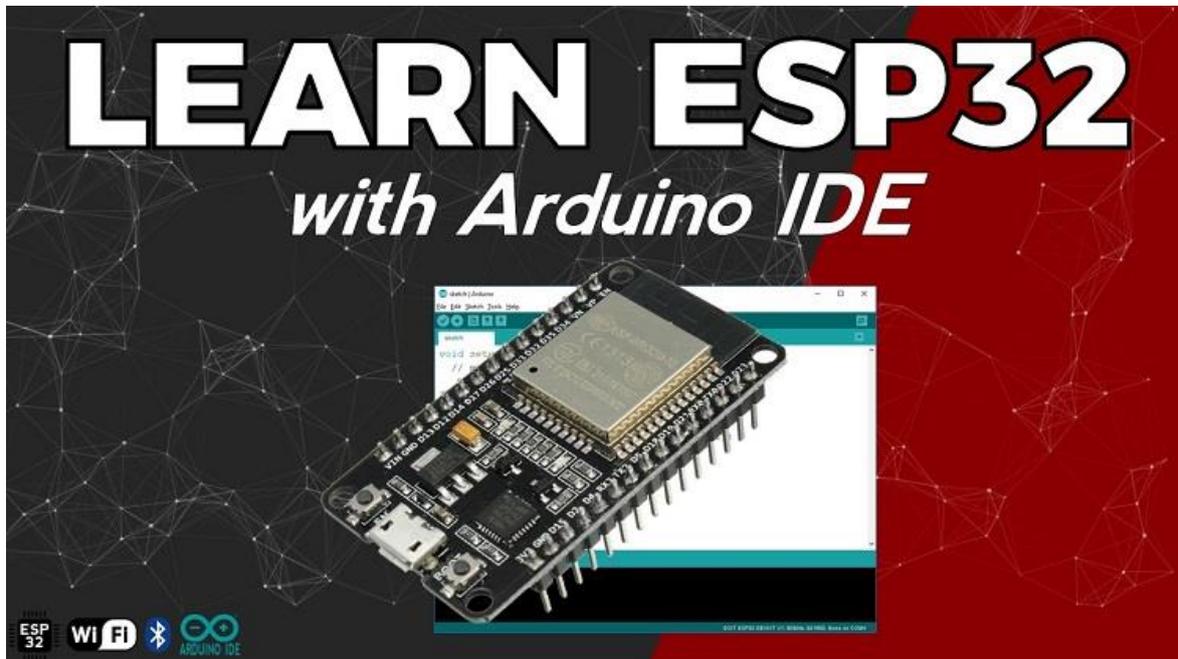


MODULE 0

Course Intro

Welcome to Learn ESP32 with Arduino IDE

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



How to Follow this Course?

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 watching/reading at any time. However, you **MUST** follow Module 1 first to properly setup the ESP32 in your Arduino IDE. Otherwise, the examples in the following Modules won't work.

Download Source Code and Resources

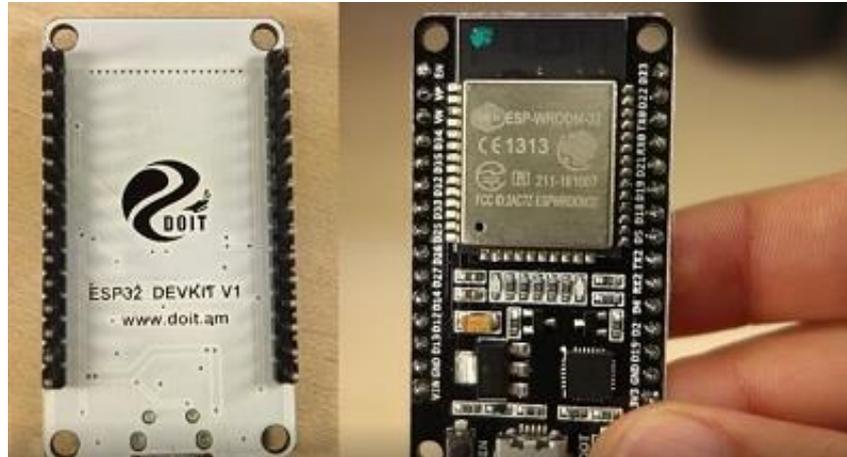


Each Unit contains the source code, schematics, and all the resources you need to follow the projects. You can download each resource at the Unit page, or you can download the [Learn ESP32 with Arduino IDE GitHub repository](#) and instantly download all the resources for this course.

Course Modules

This course contains 8 Modules and 4 Projects. Scroll down to take a look at the Modules and Projects covered in the course.

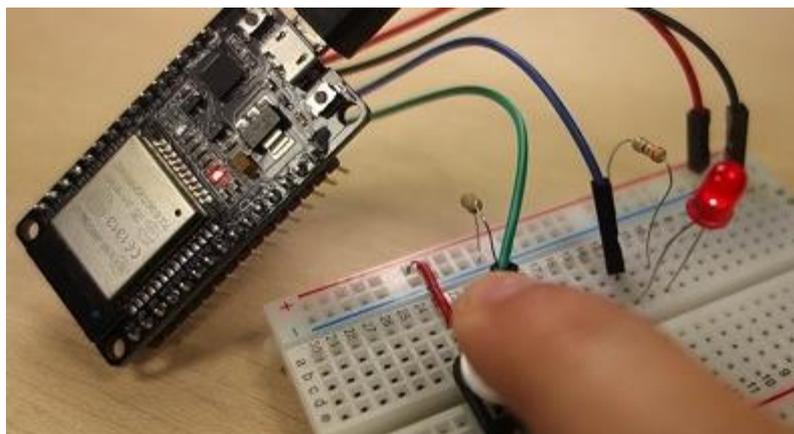
Module #1: Getting Started with ESP32



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! Units in this Module:

- Unit 1: Introducing ESP32
- Unit 2: Installing the ESP32 Board in Arduino IDE (Windows, Mac OS X, and Linux)
- Unit 3: How To Use Your ESP32 Board with this Course
- Unit 4: Make the ESP32 Breadboard Friendly

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, create PWM signals, and read digital and analog inputs. We'll also take a look at the ESP32 touch capacitive pins and the built-in hall effect sensor. Units in this Module:

- Unit 1: ESP32 Digital Inputs and Outputs
- Unit 2: ESP32 Touch Sensor
- Unit 3: ESP32 Pulse-Width Modulation (PWM)
- Unit 4: ESP32 Reading Analog Inputs
- Unit 5: ESP32 Hall Effect Sensor
- Unit 6: ESP32 with PIR Motion Sensor - Interrupts and Timers
- Unit 7: ESP32 Flash Memory - Store Permanent Data (Write and Read)
- Unit 8: Other ESP32 Sketch Examples

Module #3: ESP32 Deep Sleep Mode



Using deep sleep in your ESP32 is a great way to save power in battery-powered applications. In this Module we'll show you how to put your ESP32 into deep sleep mode and the different ways to wake it up. Units in this Module:

- Unit 1: ESP32 Deep Sleep Mode
- Unit 2: Deep Sleep - Timer Wake Up
- Unit 3: Deep Sleep - Touch Wake Up
- Unit 4: Deep Sleep - External Wake Up

Module #4: Building Web Servers with the ESP32



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

- Unit 1: ESP32 Web Server - Introduction
- Unit 2: ESP32 Web Server - Control Outputs
- Unit 3: ESP32 Web Server - HTML and CSS Basics (Part 1/2)
- Unit 4: ESP32 Web Server - HTML in Arduino IDE (Part 2/2)
- Unit 5: ESP32 Web Server – Control Outputs (Relay)
- Unit 6: Making Your ESP32 Web Server Password Protected
- Unit 7: Accessing the ESP32 Web Server From Anywhere
- Unit 8: ESP32 Web Server – Display Sensor Readings
- Unit 9: ESP32 Control Servo Motor Remotely (Web Server)
- Unit 10: ESP32 Color Picker Web Server for RGB LED Strip
- Unit 11: Asynchronous Temperature and Humidity Web Server with Auto Update
- Unit 12: Asynchronous Web Server: Control Outputs

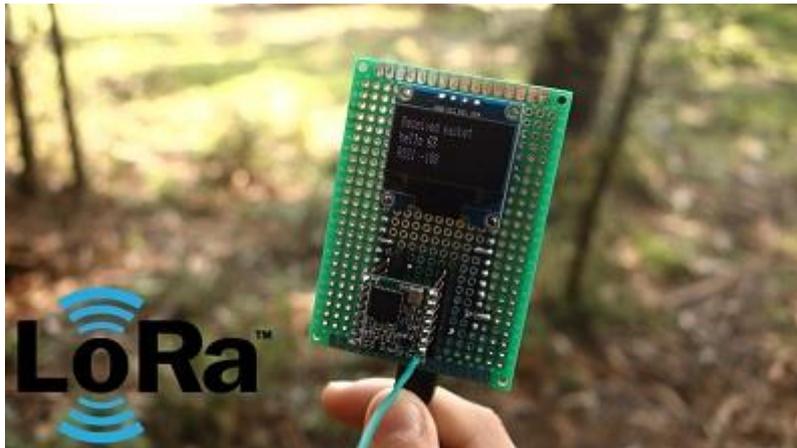
Module #5: ESP32 Bluetooth Low Energy and Bluetooth Classic



The ESP32 comes not only with Wi-Fi, but it also has 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). Units in this Module:

- Unit 1: ESP32 Bluetooth Low Energy (BLE) - Introduction
- Unit 2: Bluetooth Low Energy - Notify and Scan
- Unit 3: ESP32 BLE Server and Client (Part 1/2)
- Unit 4: ESP32 BLE Server and Client (Part 2/2)
- ESP32 with Bluetooth Classic and Android Smartphone

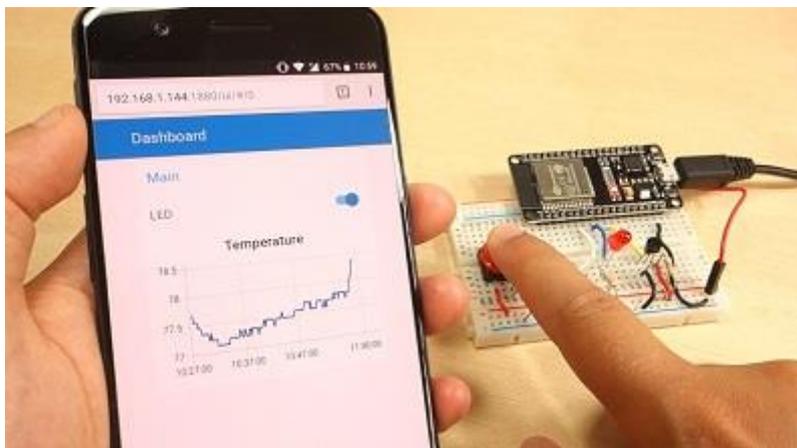
Module #6: LoRa Technology with the ESP32



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

- Unit 1: ESP32 with LoRa - Introduction
- Unit 2: ESP32 LoRa Sender and Receiver
- Unit 3: Further Reading about LoRa Gateways
- Unit 4: LoRa - Where to Go Next?

Module #7: 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. Units in this Module:

- Unit 1: ESP32 with MQTT – Introduction
- Unit 2: Installing Mosquitto MQTT Broker on a Raspberry Pi
- Unit 3: MQTT Project – MQTT Client ESP32 #1
- Unit 4: MQTT Project – MQTT Client ESP32 #2
- Unit 5: Installing Node-RED and Node-RED Dashboard on a Raspberry Pi
- Unit 6: Connect ESP32 to Node-RED using MQTT

Module #8: 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 in an easy way. Units included:

- Unit 1: ESP-NOW: Getting Started
- Unit 2: ESP-NOW Two-Way Communication Between ESP32
- Unit 3: ESP-NOW Send Data to Multiple Boards (one-to-many)
- Unit 4: ESP-NOW Receive Data from Multiple Boards (many-to-one)
- Unit 5: ESP-NOW Web Server Sensor Dashboard (ESP-NOW + Wi-Fi)

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. Units in this project:

- Unit 1: ESP32 Wi-Fi Multisensor - Temperature, Humidity, Motion, Luminosity, and Relay Control
- Unit 2: ESP32 Wi-Fi Multisensor - How the Code Works?

Project #2: Remote Controlled Wi-Fi Car Robot



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

- Unit 1: Remote Controlled Wi-Fi Car Robot - Part 1/2
- Unit 2: Remote Controlled Wi-Fi Car Robot - Part 2/2
- Unit 3: Assembling the Smart Robot Car Chassis Kit
- Unit 4: Extra - Access Point (AP) For Wi-Fi Car Robot

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



In this project you're going to create an Android application to interact with the ESP32 using Bluetooth Low Energy (BLE). Units in this project:

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

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



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 be using LoRa communication protocol. Units in this project:

- Unit 1: LoRa Long Range Sensor Monitoring and Data Logging
- Unit 2: ESP32 LoRa Sender
- Unit 3: ESP32 LoRa Receiver
- Unit 4: ESP32 LoRa Sender Solar Powered
- Unit 5: Final Tests, Demonstration, and Data Analysis

Extra Units

The course also comes with a special module with some extra units:

- ESP32 Static/Fixed IP Address
- ESP32 Dual Core – Create Tasks
- ESP32 SPIFFS (SPI Flash File System)
- Build an ESP32 Web Server using Files from Filesystem (SPIFFS)
- ESP32 Over-the-air (OTA) Programming - Web Updater
- ESP32 Client-Server Wi-Fi Communication Between Two Boards
- ESP32 HTTP GET (OpenWeatherMap and Thingspeak)
- ESP32 HTTP POST (ThingSpeak and IFTTT.com)
- ESP32 Pinout Reference: Which GPIO pins should you use?

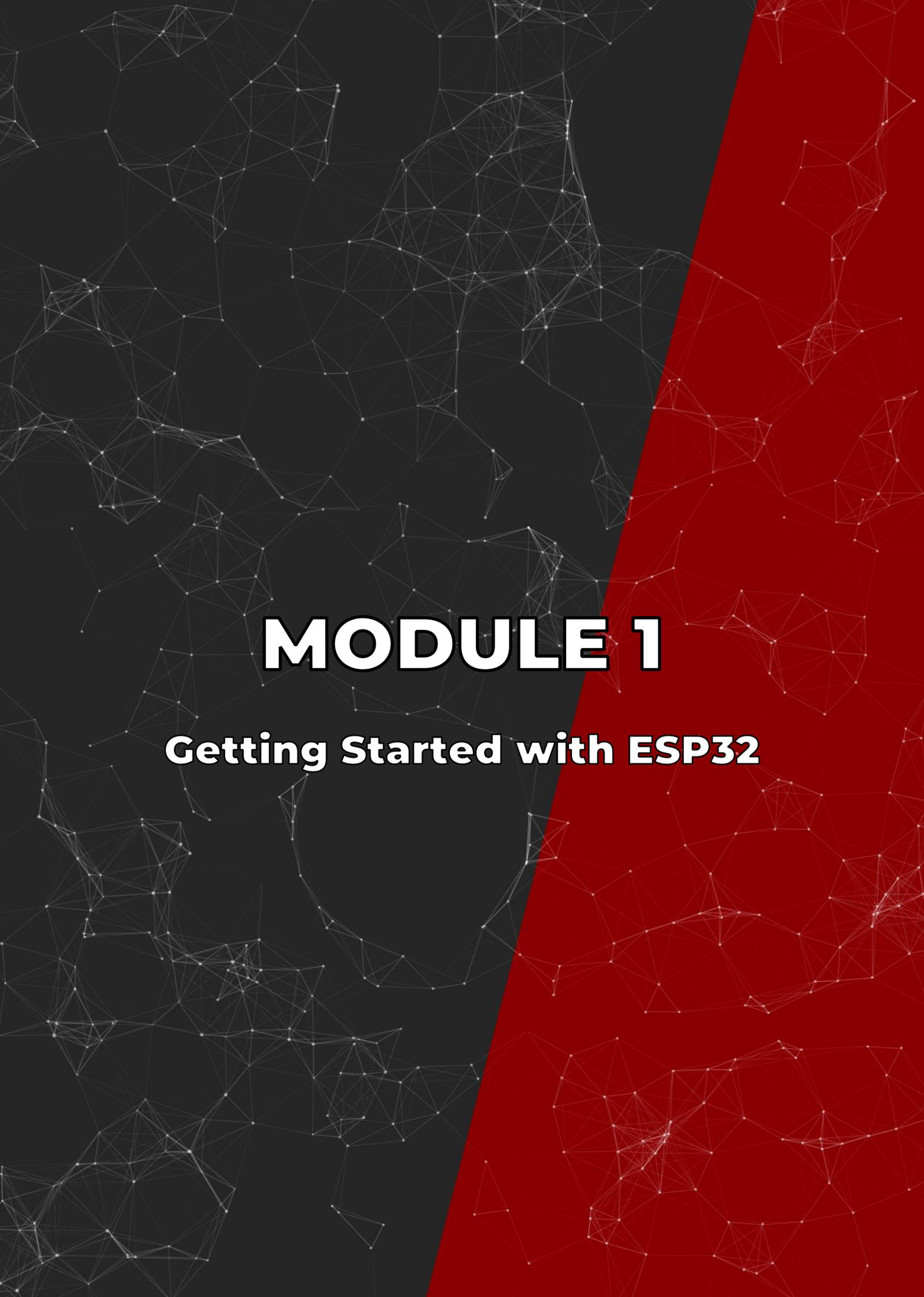
Getting Parts for the Course

To properly follow this course, you need some electronics components. In each Module we provide a complete list of the needed parts and links to [Maker Advisor](#), so that you can find the part you're looking for on 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 take a look at [our favorite tools and parts here](#).

Note: for the full parts list, consult the Appendix at the end of this eBook.



MODULE 1

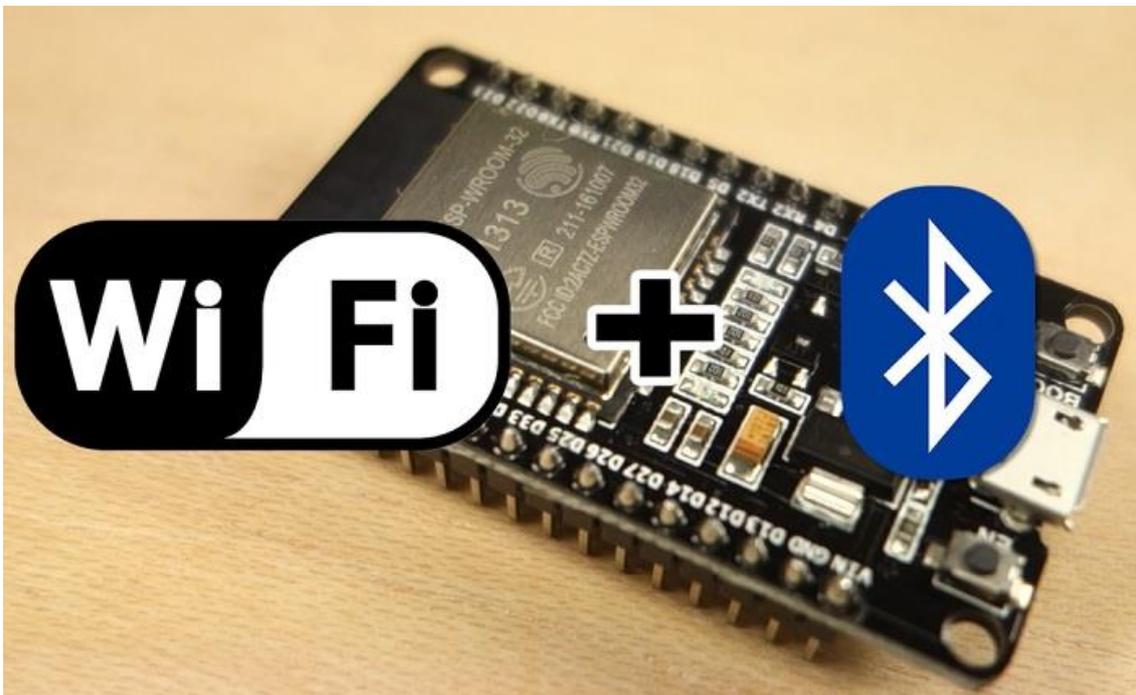
Getting Started with ESP32

Unit 1 - Introducing ESP32



Introducing the ESP32 Board

This unit is an introduction to the ESP32 which is the ESP8266 successor. The ESP32 is loaded with lots of new features. It now combines Wi-Fi and Bluetooth wireless capabilities.

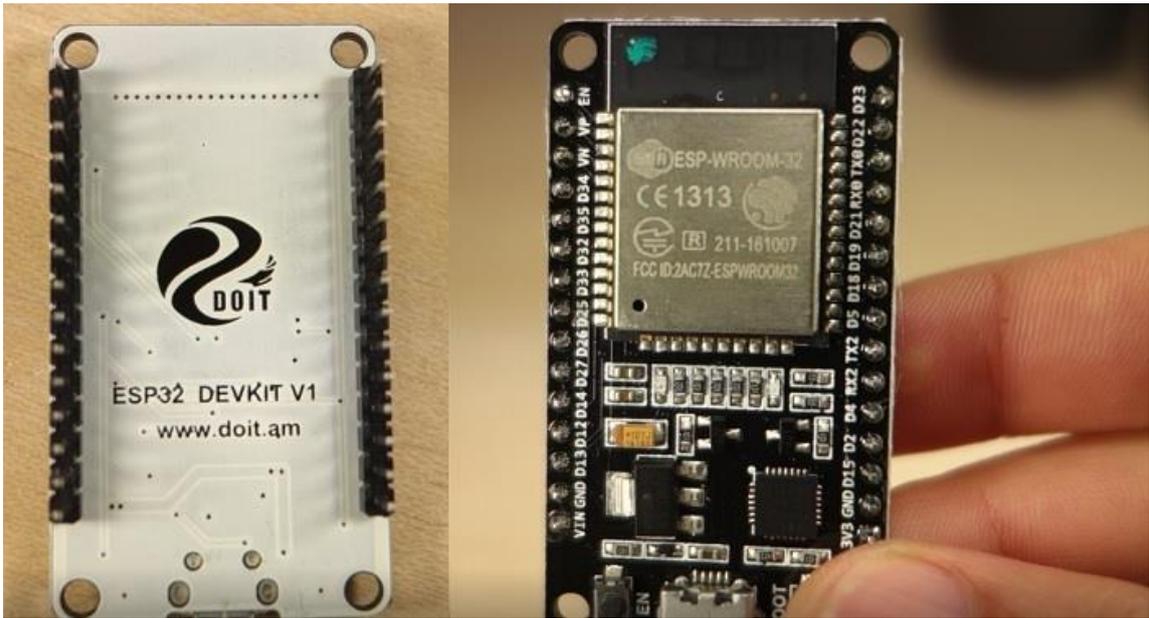


There are a lot of ESP32 development boards. I encourage you to visit the [ESP32.net website](https://www.ESP32.net) where each ESP32 chip and development board are listed. You can compare their differences and features.



Introducing the ESP32 DOIT DEVKIT V1 Board

For this course, we'll be using the [ESP32 DEVKIT V1 DOIT board](#), but any other ESP32 with the ESP-WROOM-32 chip will work just fine.



Here's just a few examples of boards that are very similar and compatible with the projects that will be presented throughout this course.

DOIT DEVKIT V1	ESP32 DevKit	ESP-32S NodeMCU	ESP32 Thing
			
WEMOS LOLIN32	"WeMos" OLED	HUZZAH32	Others
			(...)

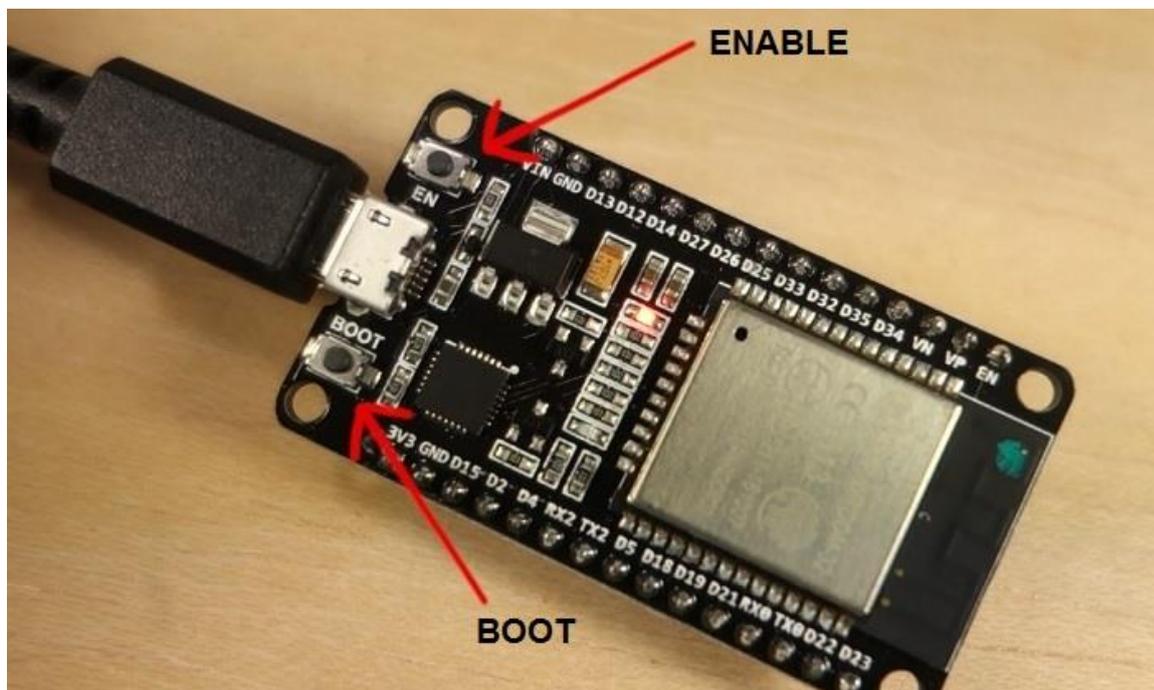
We'll provide a more in depth guide that compares the differences between the main ESP32 development boards and how to identify them. In summary, any ESP32 board should work with this course with small changes in the wiring.

Features

Let's take a closer look at the board. It comes with the ESP-WROOM-32 chip. It has a 3.3V voltage regulator that drops the input voltage to power the ESP32 chip. And it also comes with a CP2102 chip that allows you to plug the ESP32 to your computer to program it without the need for an FTDI programmer.



The board has two on-board buttons: the ENABLE and the BOOT button.



If you press the ENABLE button, it reboots your ESP32. If you hold down the BOOT button and then press the enable, the ESP32 reboots in programming mode. You don't need to worry about these details at the moment, because we'll explore them in the next sections.

If you don't know where to get the ESP32, you can check this page on [Maker Advisor](#).

Specifications

When it comes to the ESP32 chip specifications, you'll find that:

- The ESP32 is dual core, this means it has 2 processors.
- It has Wi-Fi and bluetooth built-in.
- It runs 32 bit programs.
- The clock frequency can go up to 240MHz and it has a 512 kB RAM.
- This particular board has 30 pins, 15 in each row. (There's a new version of this board with 36 pins.)
- It also has wide variety of peripherals available, like: capacitive touch, ADCs, DACs, UART, SPI, I2C and much more. We'll explore these functionalities later in the course.

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
Peripherals	Capacitive touch, ADCs (analog-to-digital converter), DACs (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), and more.

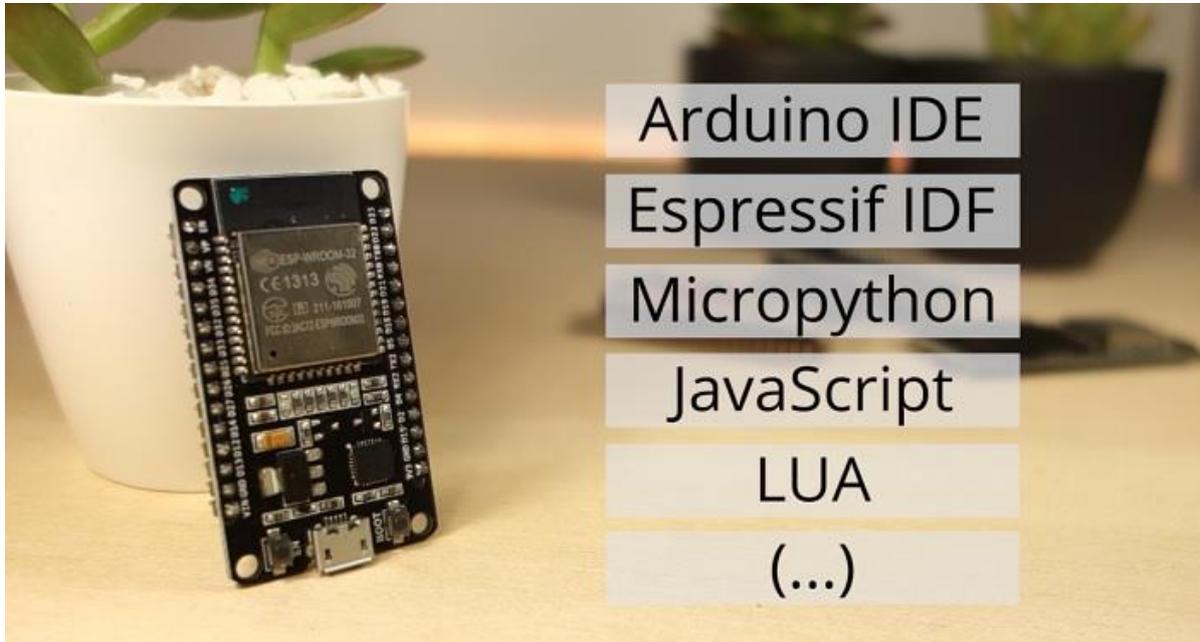
ESP32 Pinout

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

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

Programming Environments

The ESP32 can be programmed in different programming environments. You can use the Arduino IDE, Espressif IDF (IoT Development Framework), Micropython, JavaScript, LUA, etc. Throughout this course will be focusing mainly on programming the ESP32 with the Arduino IDE.



Next

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

Unit 2 - Installing ESP32 in Arduino IDE (Windows, Mac OS X, and Linux)

Important: before starting this installation procedure, make sure you have the latest version of the Arduino IDE installed in your computer.

The ESP32 is currently being integrated with the Arduino IDE just like it was done for the ESP8266. This add-on for the Arduino IDE allows you to program the ESP32 using the Arduino IDE and its programming language. You can find the latest Windows instructions at the official [GitHub repository](#).

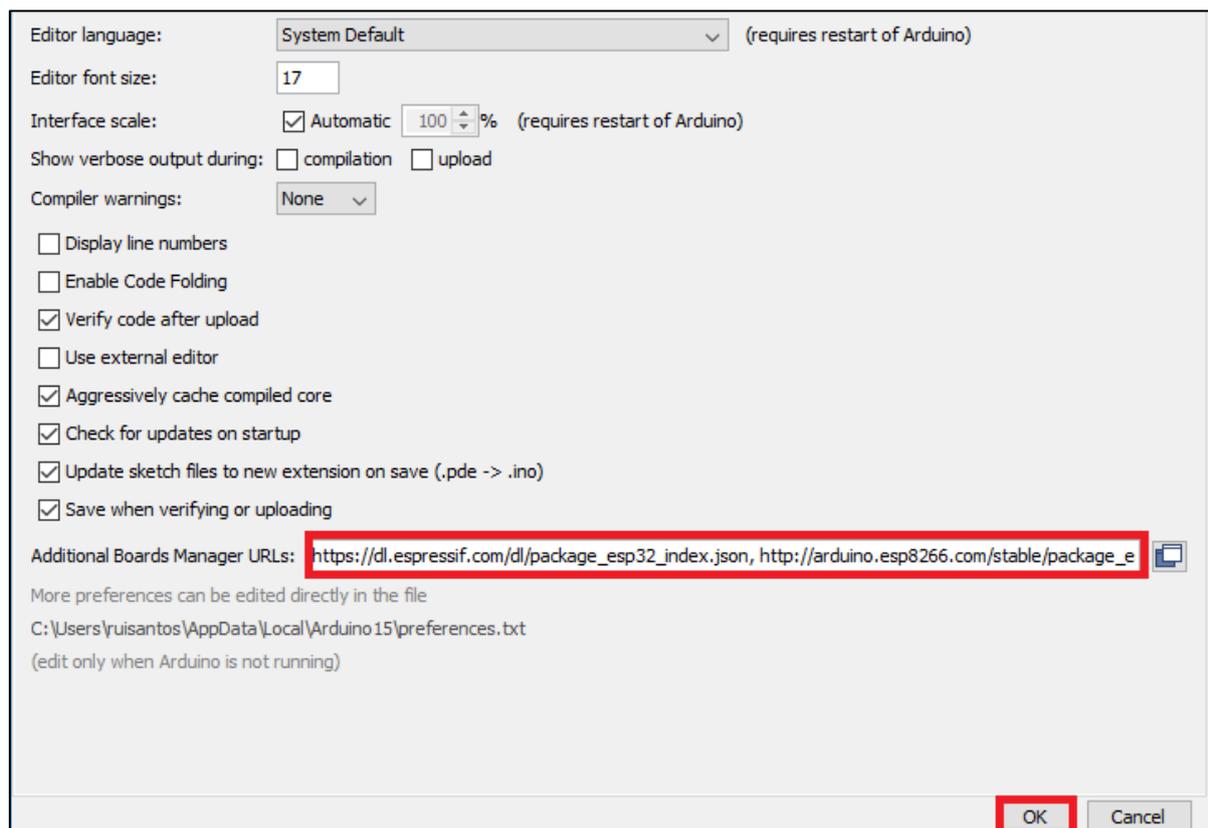
Installing the ESP32 Board

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

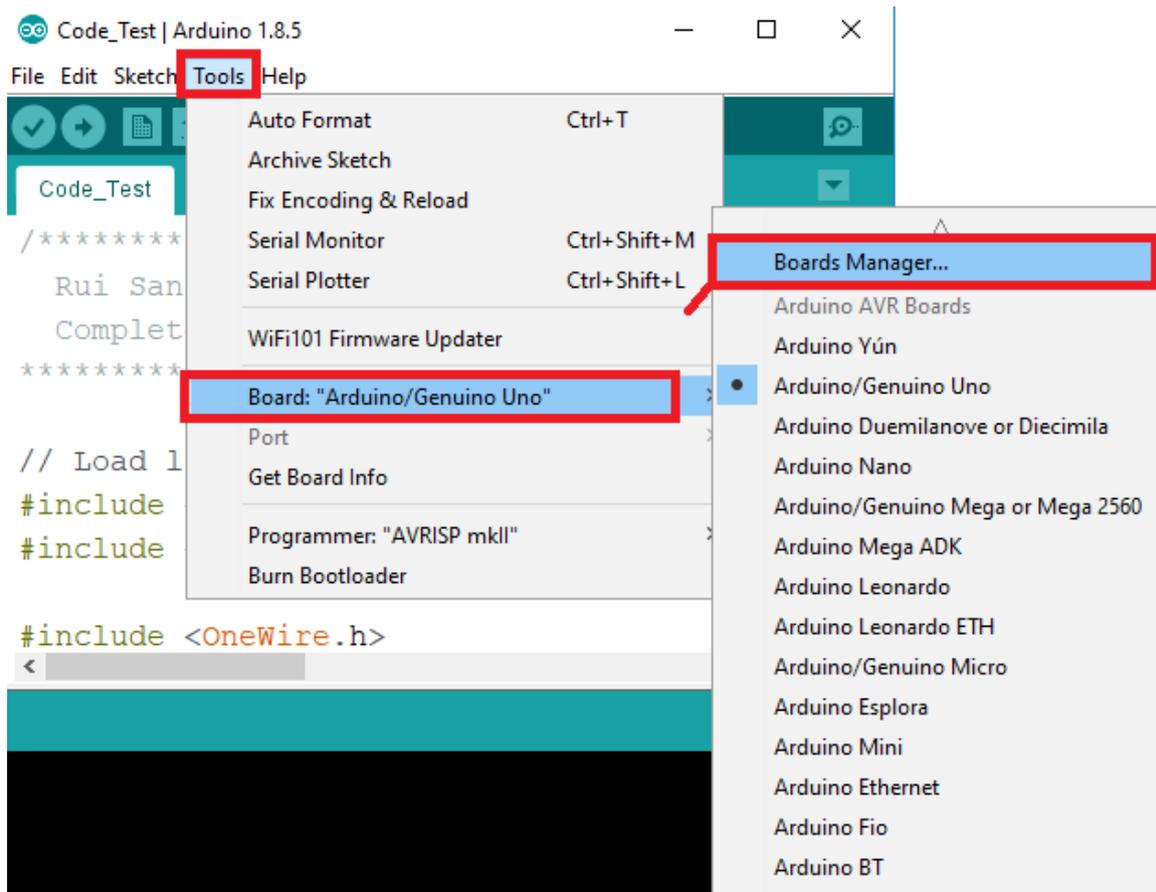
- 1) Open the preferences window from the Arduino IDE. Go to **File ▶ Preferences**
- 2) Enter **`https://dl.espressif.com/dl/package_esp32_index.json`** into the **“Additional Board Manager URLs”** field as shown in the figure below. Then, click the **“OK”** button.

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

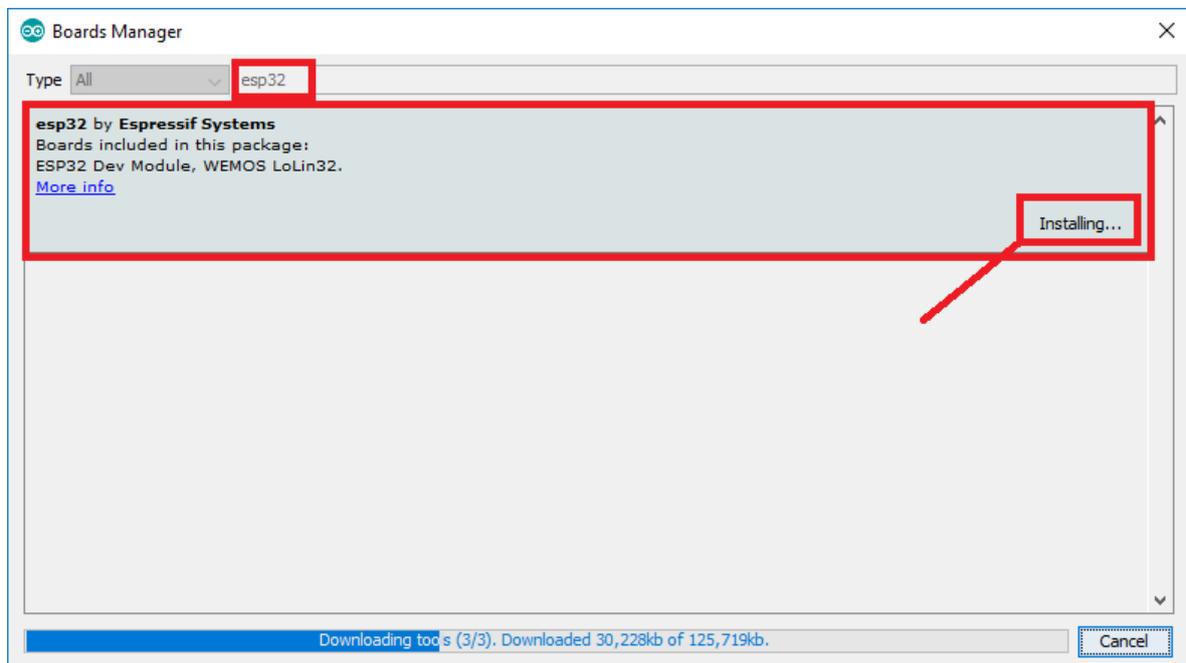
```
https://dl.espressif.com/dl/package_esp32_index.json,  
http://arduino.esp8266.com/stable/package_esp8266com_index.json
```



3) Open boards manager. Go to **Tools** ▶ **Board** ▶ **Boards Manager...**



4) Search for ESP32 and press install button for the “ESP32 by Espressif Systems”:

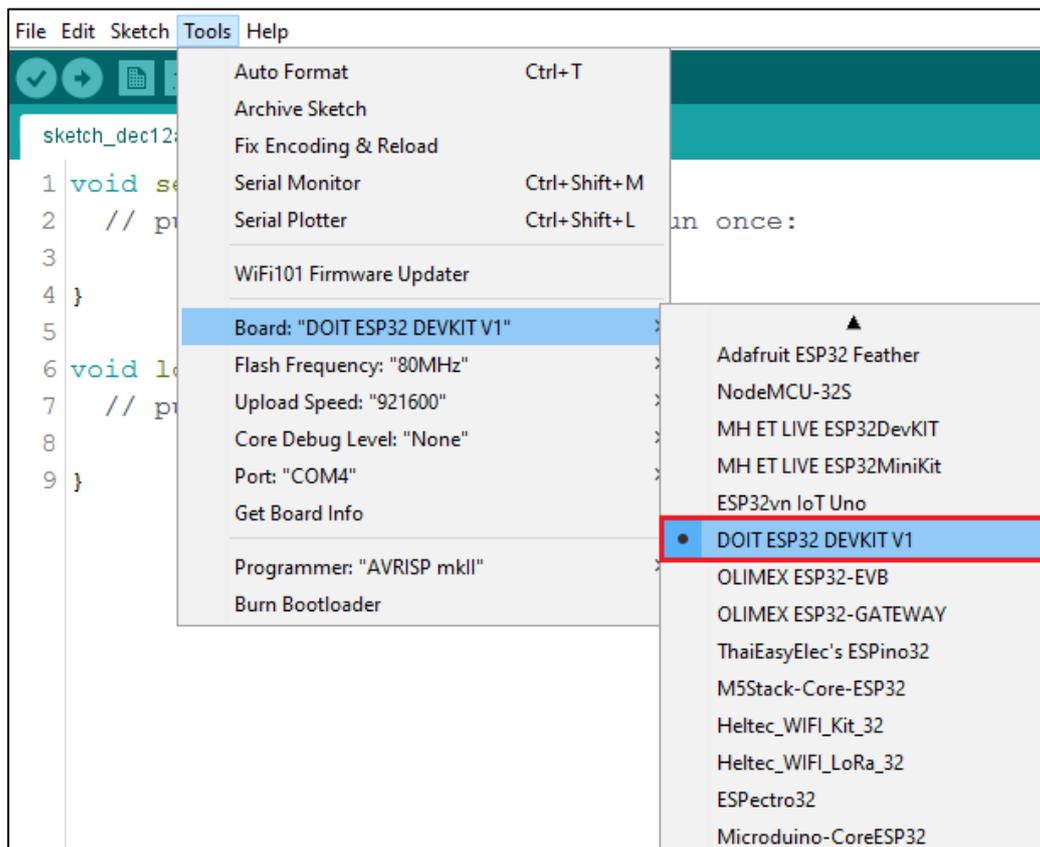


Testing the Installation

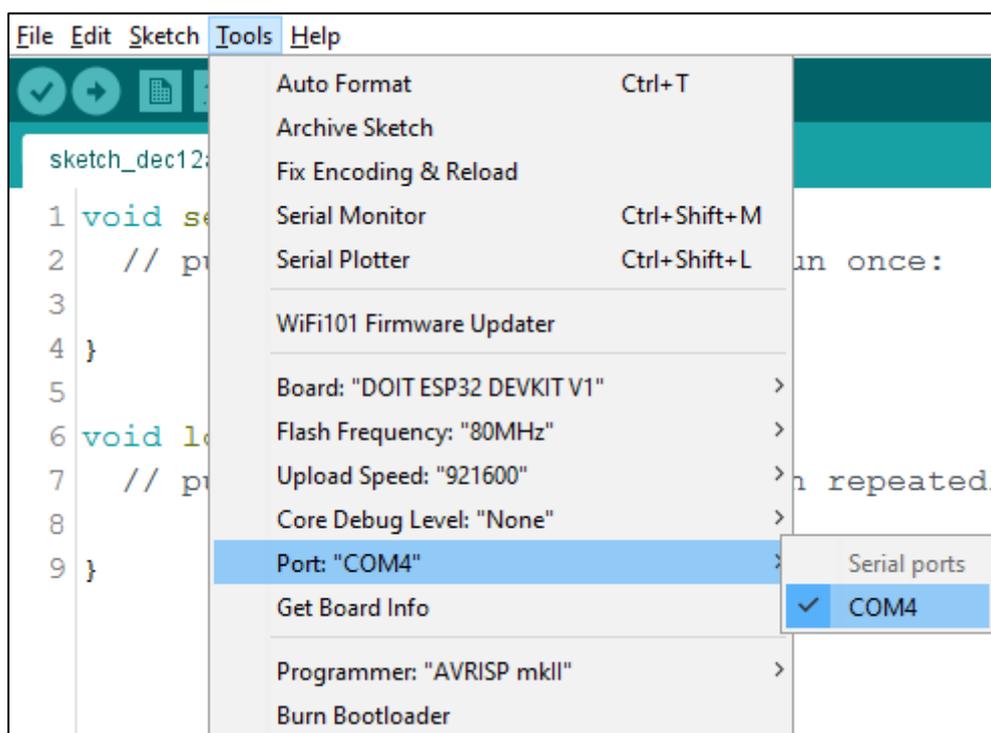
Plug your [ESP32 DOIT DEVKIT V1 Board](#) to your computer. Then, follow these steps:

1) Open Arduino IDE

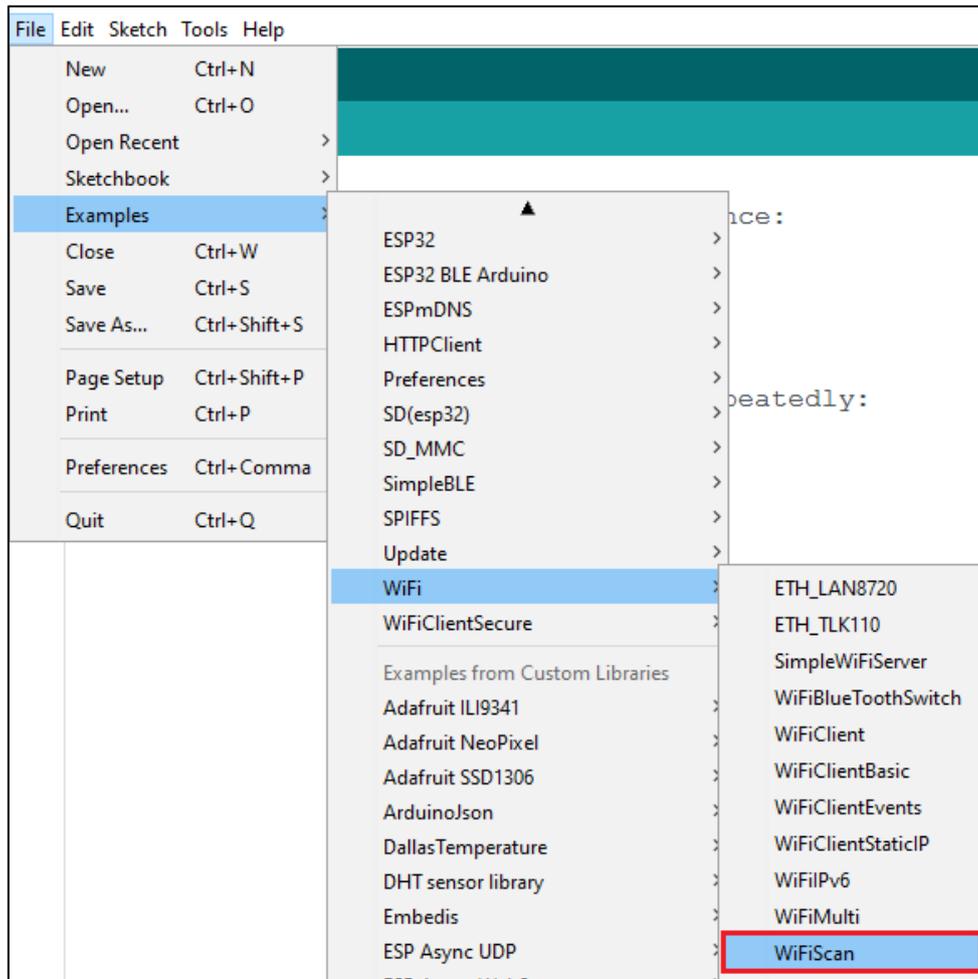
2) Select your **Board** in **Tools** ▶ **Board** menu (in our case it's the **DOIT ESP32 DEVKIT V1**)



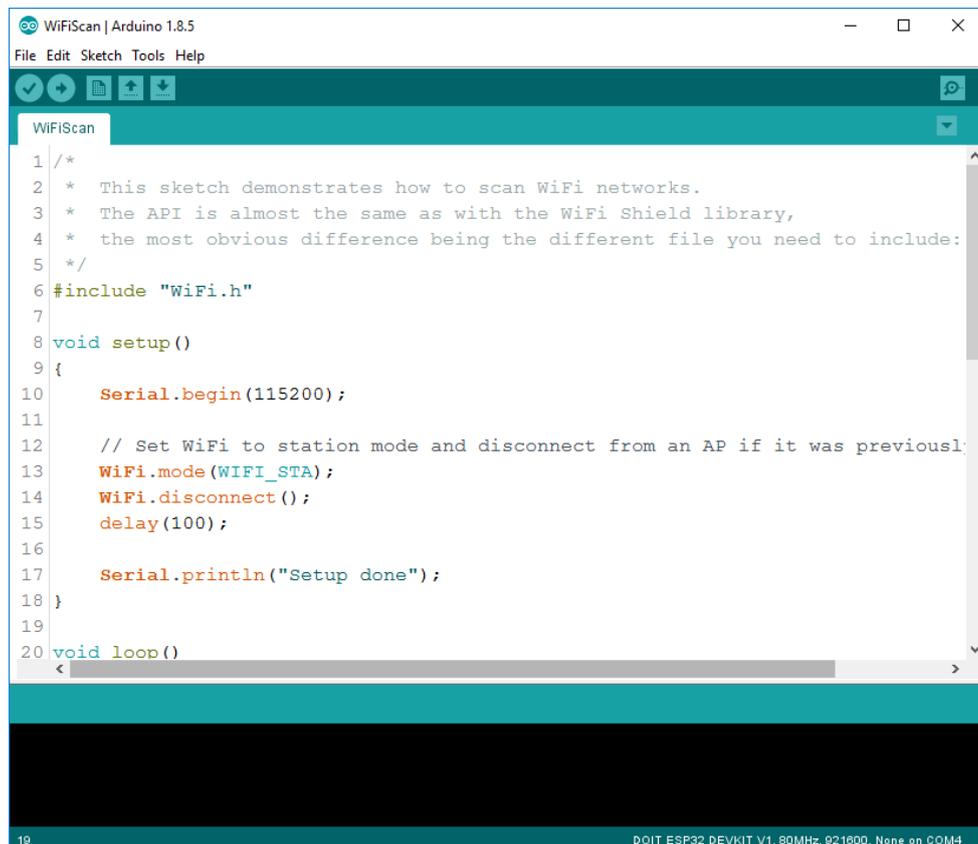
3) Select the Port (if you don't see the COM Port in your Arduino IDE, you need to install the [ESP32 CP210x USB to UART Bridge VCP Drivers](#)):



4) Open the following example under **File** ▶ **Examples** ▶ **WiFi (ESP32)** ▶ **WiFi Scan**



5) A new sketch opens:



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



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

A screenshot of the Arduino IDE Serial Monitor window. The title bar reads 'COM4'. The text in the monitor shows the upload progress: 'writing at 0x00042000... (84 %)', 'Writing at 0x00050000... (89 %)', 'Writing at 0x00054000... (94 %)', 'Writing at 0x00058000... (100 %)', 'Wrote 481440 bytes (299651 compressed) at 0x00010000 in 4.7 seconds', 'Hash of data verified.', 'Compressed 3072 bytes to 122...', 'Writing at 0x00008000... (100 %)', 'Wrote 3072 bytes (122 compressed) at 0x00008000 in 0.0 seconds', 'Hash of data verified.', 'Leaving...', and 'Hard resetting...'. At the bottom, the hardware information is displayed: 'DOIT ESP32 DEVKIT V1, 80MHz, 921600, None on COM4'. The 'Done uploading.' message is highlighted with a red box.

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



9) Press the ESP32 on-board Enable button and you should see the networks available near your ESP32:

A screenshot of the Arduino IDE Serial Monitor window. The title bar reads 'COM4'. The text in the monitor shows the results of a network scan: 'scan done', '2 networks found', '1: MEO-620B4B (-49)*', '2: MEO-WiFi (-50)', 'scan start', 'scan done', '2 networks found', '1: MEO-620B4B (-48)*', '2: MEO-WiFi (-49)'. At the bottom, the 'Autoscroll' checkbox is checked, the 'Both NL & CR' dropdown is set to 'Both NL & CR', the '115200 baud' dropdown is highlighted with a red box, and the 'Clear output' button is visible.

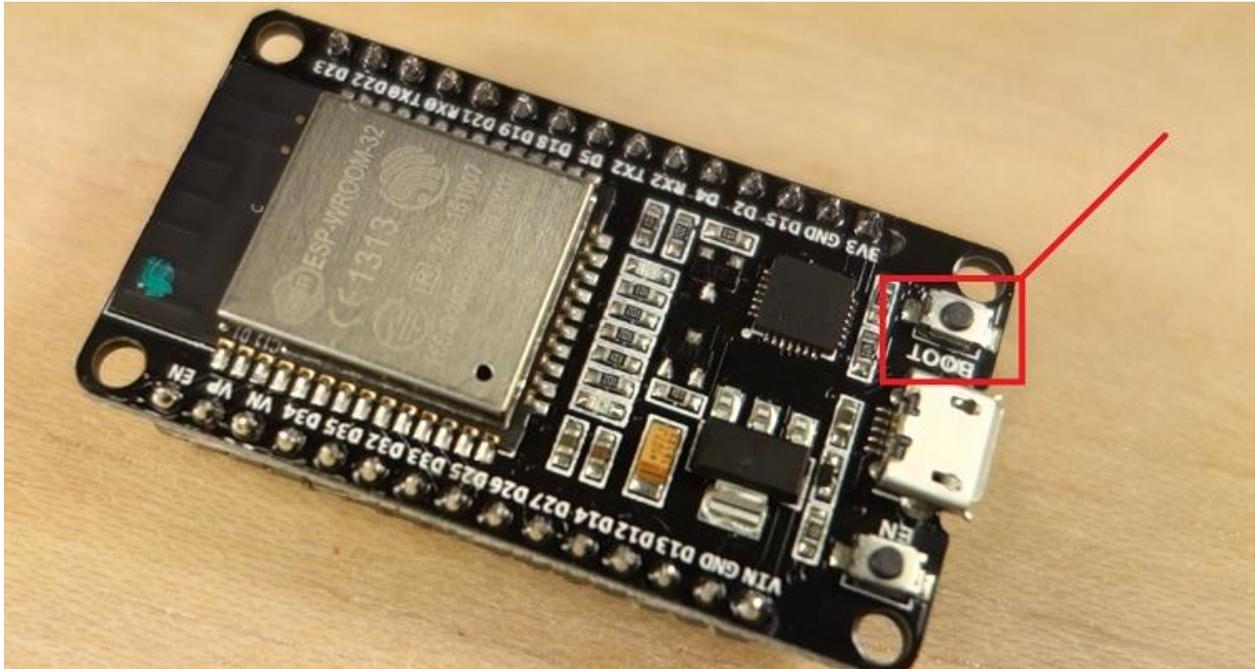
This is a very basic tutorial that illustrates how to prepare your Arduino IDE for the ESP32 on your computer.

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 in 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 finger from the “**BOOT**” button:

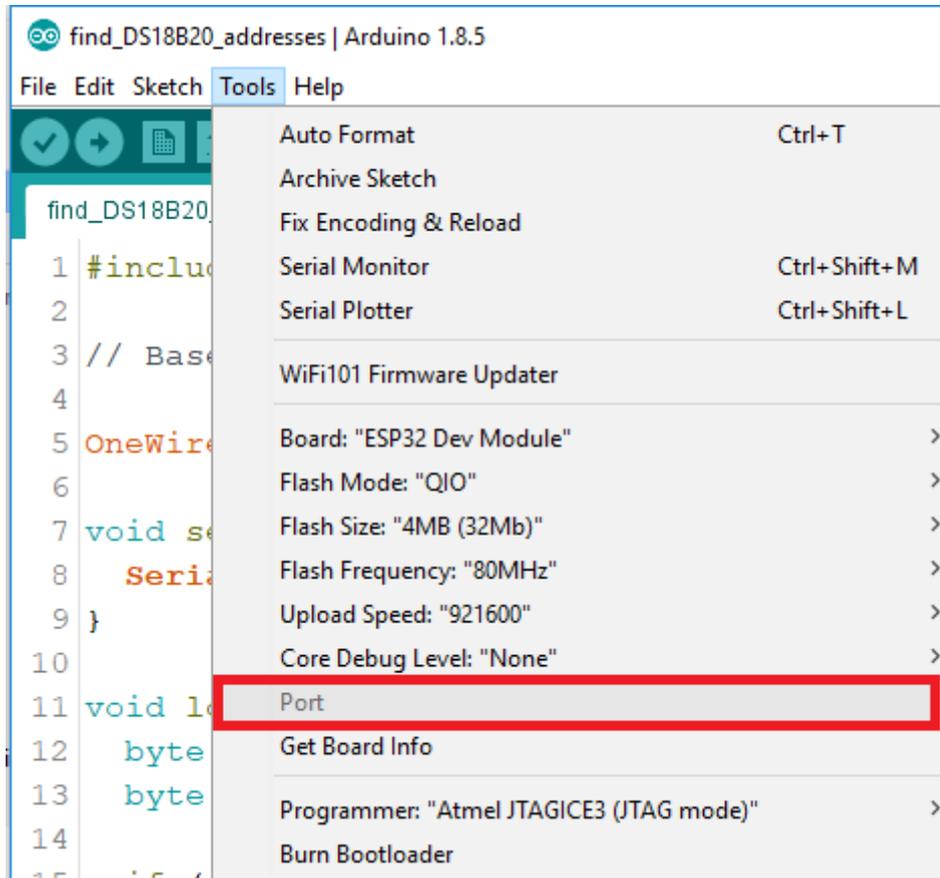
```
Uploading...
Archiving built core (caching) in: C:\Users\RUISAN~1\AppData\Local\Temp\arduino_cache_959883\c
Sketch uses 501366 bytes (38%) of program storage space. Maximum is 1310720 bytes.
Global variables use 37320 bytes (12%) of dynamic memory, leaving 257592 bytes for local varia
esptool.py v2.1
Connecting.....
Chip is ESP32D0WDQ6 (revision (unknown 0xa))
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 921600
Changed.
Configuring flash size...
Auto-detected Flash size: 4MB
Compressed 8192 bytes to 47...
```

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

Troubleshooting Tip #2:

COM Port not found/not available

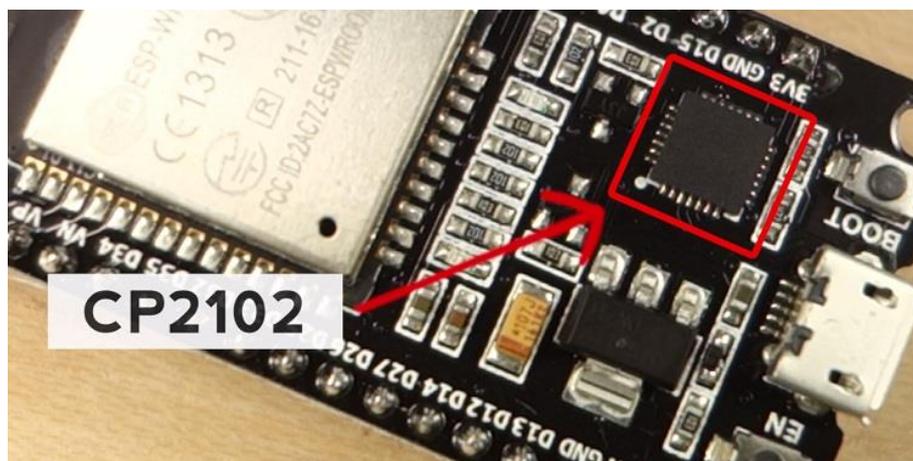
If you plug your ESP32 board to your computer, but you can't find the ESP32 Port available in your Arduino IDE (it's grayed out):



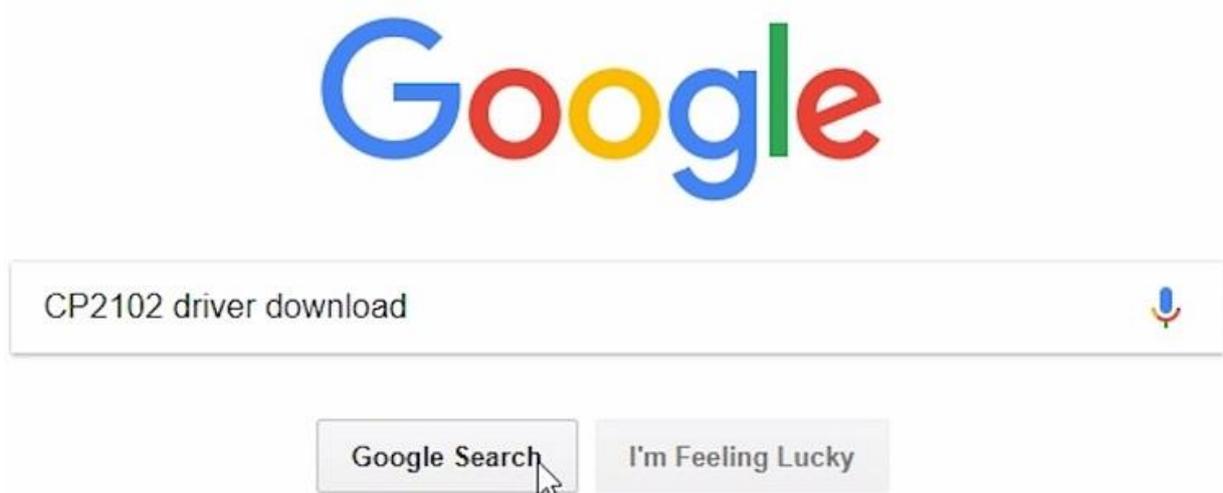
It might be one of these two problems: **1. USB drivers missing** or **2. USB cable without data wires**.

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.



Go to Google and search for your particular chip to find the drivers and install them in your operating system.



You can download the CP2102 drivers on the [Silicon Labs](#) website.

SILICON LABS

简体中文 日本語 Log In | Register

Parametric Search | Cross-Reference Search

Search silabs.com GO

About Products Solutions Community & Support

Silicon Labs » Products » Development Tools » Software » USB to UART Bridge VCP Drivers

CP210x USB to UART Bridge VCP Drivers

The CP210x USB to UART Bridge Virtual COM Port (VCP) drivers are required for device operation as a Virtual COM Port to facilitate host communication with CP210x products. These devices can also interface to a host using the direct access driver. These drivers are static examples detailed in application note 197: The Serial Communications Guide for the CP210x, download an example below:

AN197: The Serial Communications Guide for the CP210x

Download Software

The CP210x Manufacturing DLL and Runtime DLL have been updated and must be used with v6.0 and later of the CP210x Windows VCP Driver. Application Note Software downloads affected are AN144SW.zip, AN205SW.zip and AN223SW.zip. If you are using a 5x driver and need support you can download archived Application Note Software.

[Legacy OS software and driver package download links and support information >](#)

Download for Windows 10 Universal (v10.1.1)

Platform	Software	Release Notes
Windows 10 Universal	Download VCP (2.3 MB)	Download VCP Revision History

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

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.

Unit 3 - How to Use Your ESP32 Board with this Course

For this course, we'll be using the ESP32 DEVKIT V1 DOIT board, but any other ESP32 with the ESP-WROOM-32 chip should work just fine.



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



There are many ESP32 development boards available, and if you have a different board than these, you are also able to follow the course.

How to Use Your ESP32 Board with this Course

To make it easier to follow along, regardless of the ESP32 board you're using, we've created this section to show you which changes you need to do to make your ESP32 work.

You just need to keep in mind two things:

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

Visiting the Product Page

In order to do that, first you need to identify your ESP32 board. A good place to start is opening the product page where you purchased your ESP32. I've ordered mine from [Banggood](#) and here's the product page.



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 actually bought an ESP32 DOT IT board. Even though they are very similar, they are different.

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 with silk screen. In my case, you can clearly see this is the ESP32 DEVKIT V1 DOIT board.

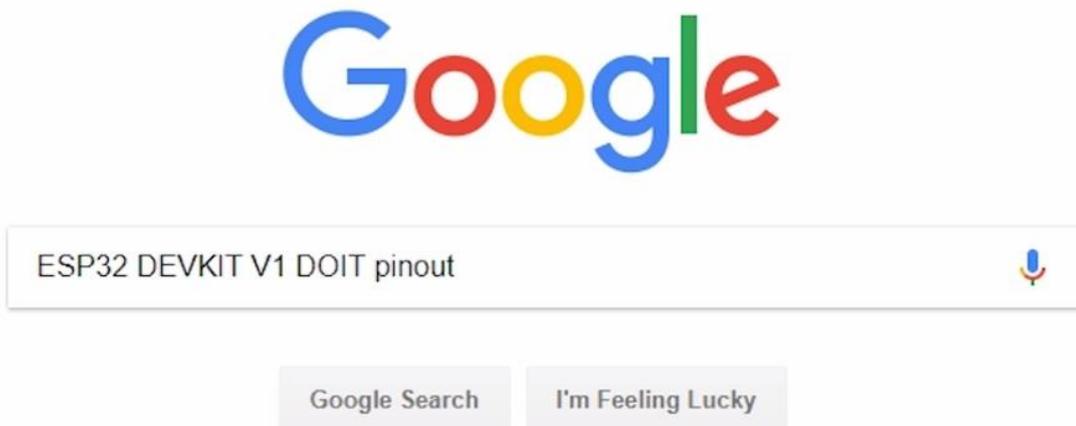


If you have an ESP32 NodeMCU, the following figure shows how it looks like (it says NodeMCU ESP-32S).



ESP32 Pinout

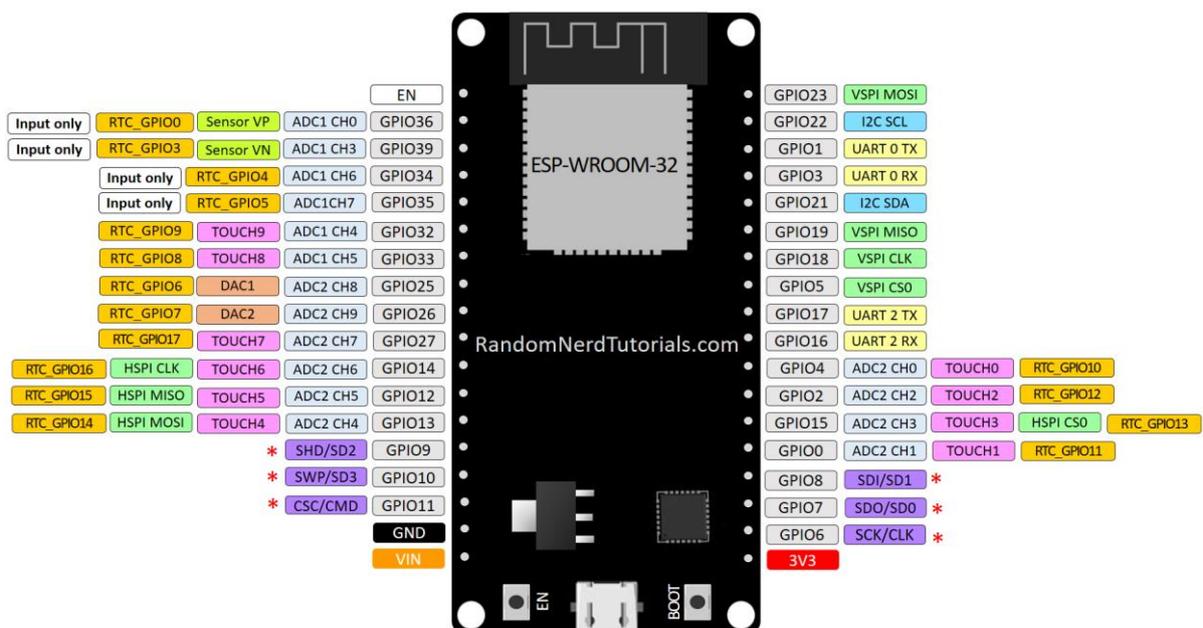
Knowing the name of your board is very important so that you can search for its pinout. Now you can go to Google and search for your development board pinout. Search for your ESP32 board name and add the "pinout" keyword in the end.



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

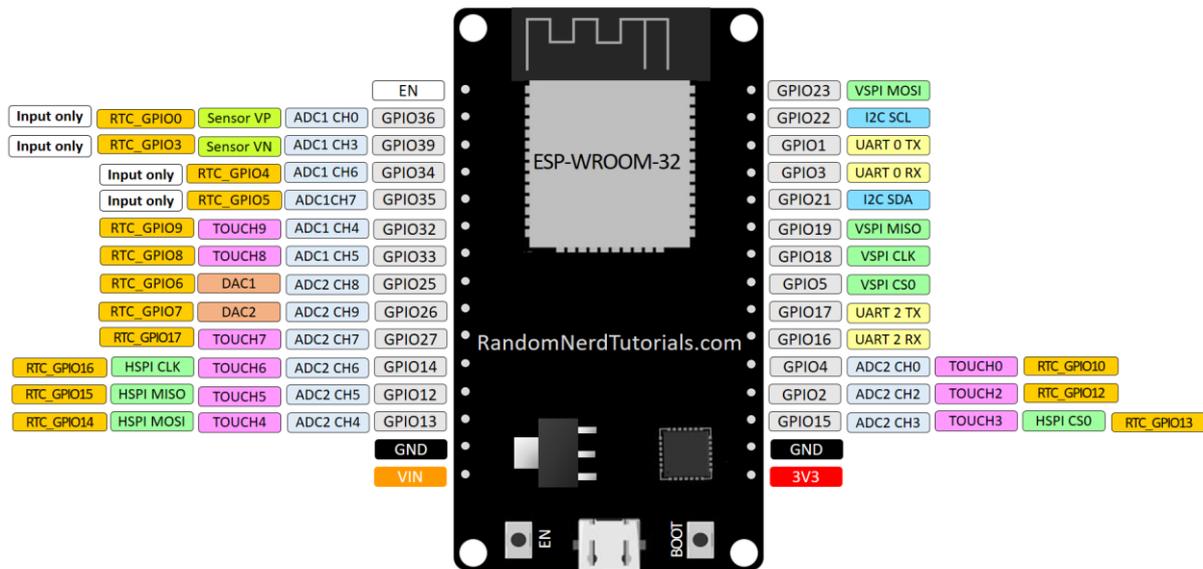
- [Printable version](#)
- [Image version 30 pins](#)
- [Image version 36 pins](#)

ESP32 DEVKIT V1 – DOIT version with 36 GPIOs



ESP32 DEVKIT V1 – DOIT

version with 30 GPIOs



I also recommend visiting the [ESP32.net](https://www.esp32.net) website as it provides an extensive list with names and figures for all known ESP32 development boards.

Blink – Example Sketch

Let's take a look at an example sketch to show you what you need to worry about, if you want to build a simple circuit that blinks an LED with the ESP32. Copy the following code to the Arduino IDE:

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Blink_LED/Blink_LED.ino

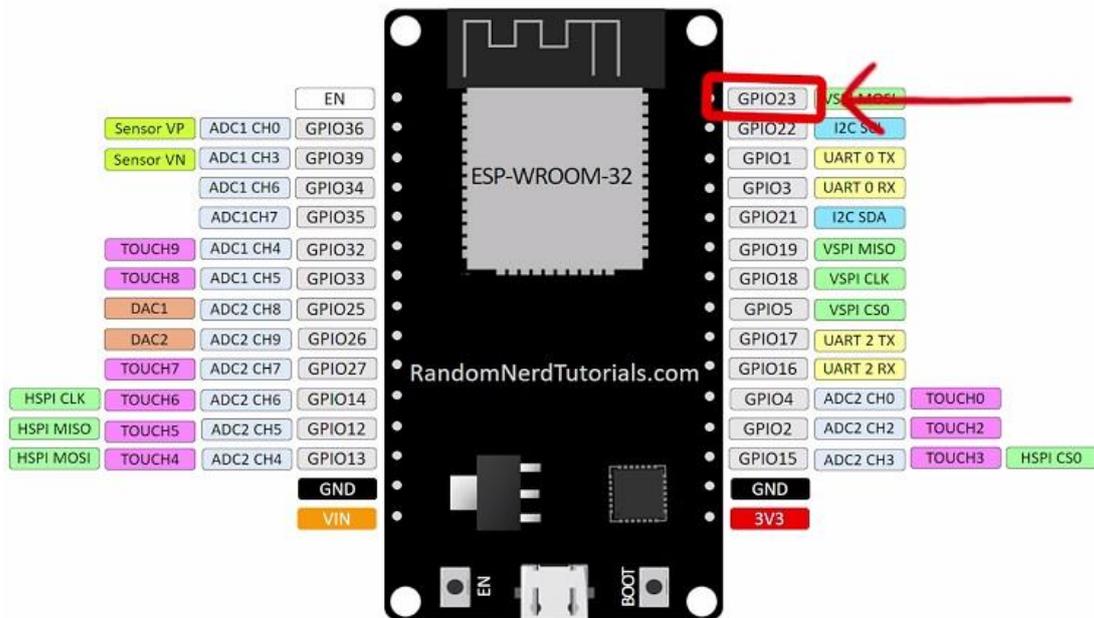
```
/*  
  Blink  
*/  
  
// ledPin refers to ESP32 GPIO 23  
const int ledPin = 23;  
  
// the setup function runs once when you press reset or power the board  
void setup() {  
  // initialize digital pin ledPin as an output.  
  pinMode(ledPin, OUTPUT);  
}  
  
// the loop function runs over and 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;
```

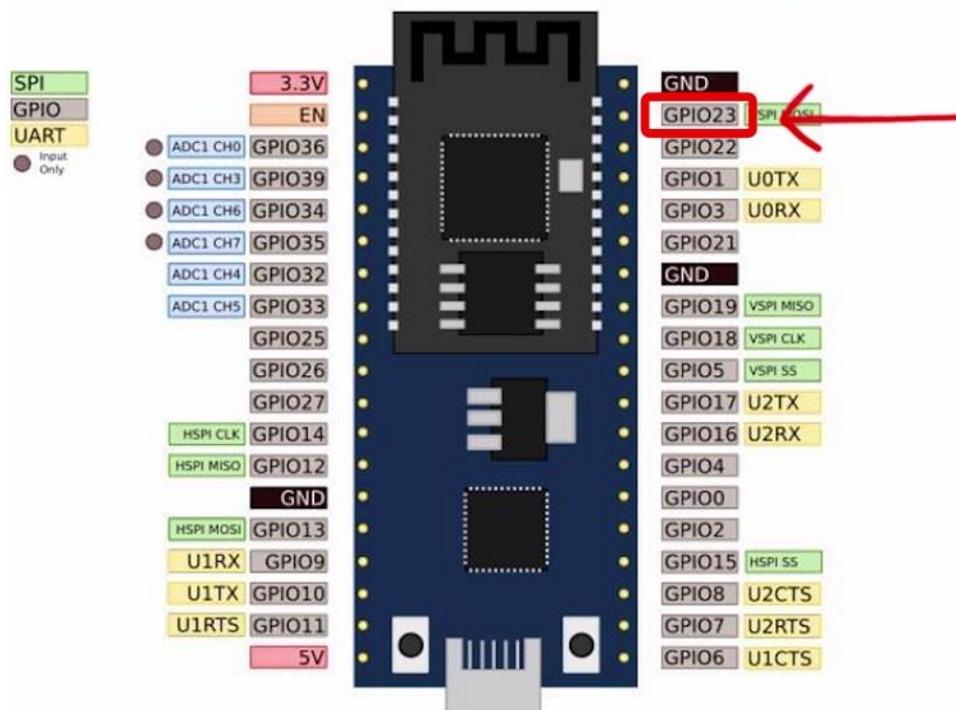
If you're using the ESP32 DEVKIT V1 DOIT board, you need to connect your LED to the first pin on the top right corner.

ESP32 DEVKIT V1 - DOIT



But if you're using the NodeMCU ESP-32S board, GPIO 23 is located in the 2nd pin of the top right corner, as shown in the figure below.

NodeMCU ESP-32S



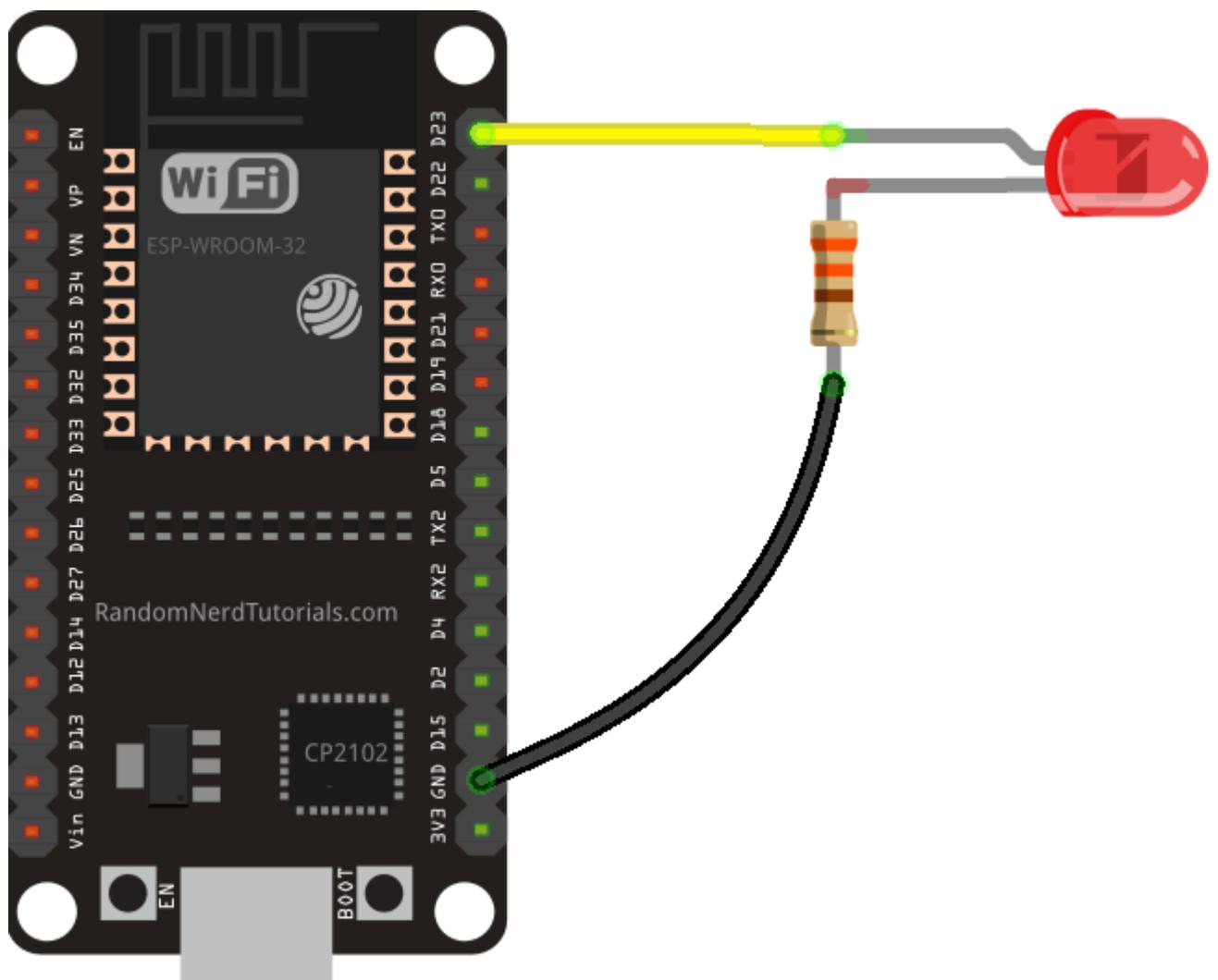
Important: always check the pinout for your specific board, before building any circuit.

Schematic

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [330 Ohm resistor](#)
- [Jumper wires](#)
- [Breadboard](#) (optional)

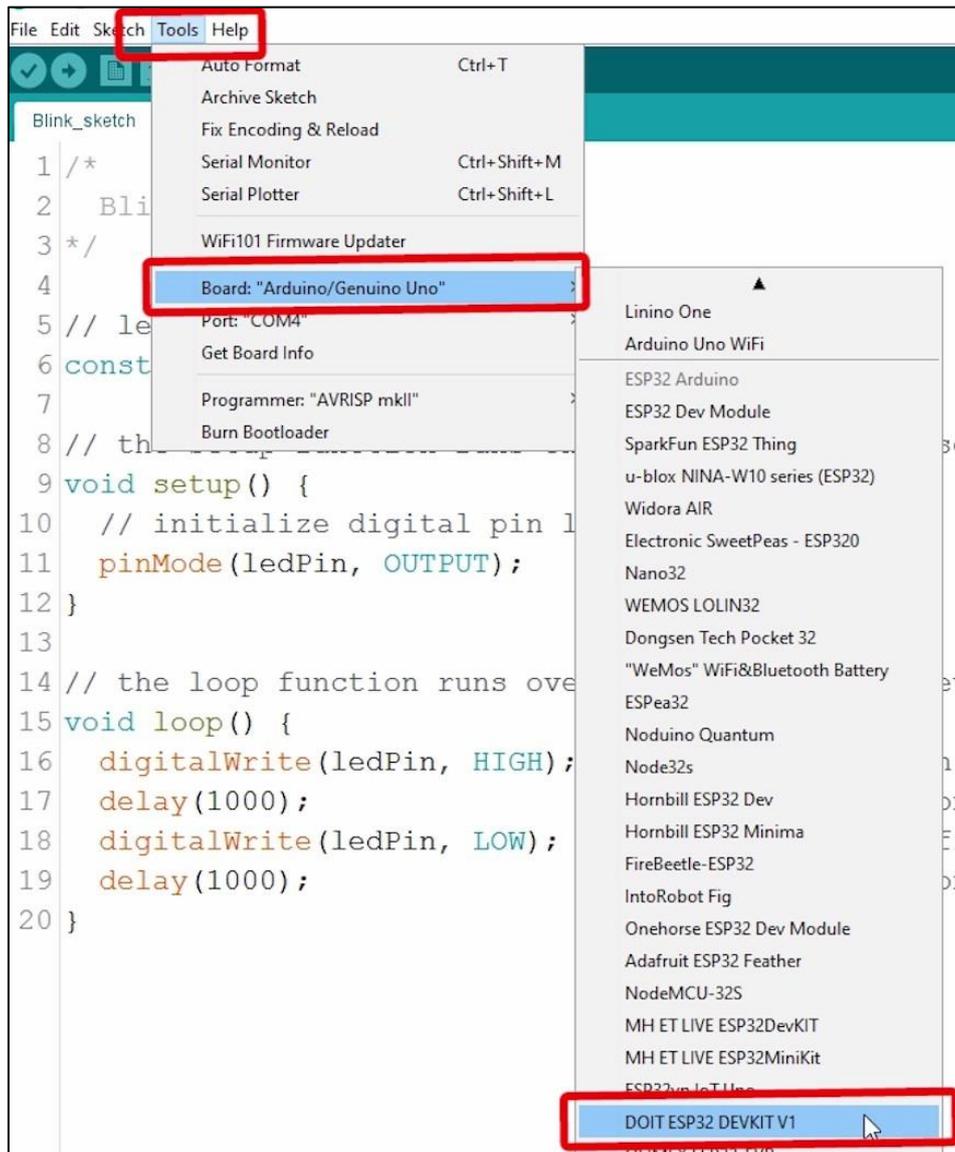
Follow the next schematic to wire the LED to the ESP32 DEVKit DOIT board (also check where GND is located in your board).



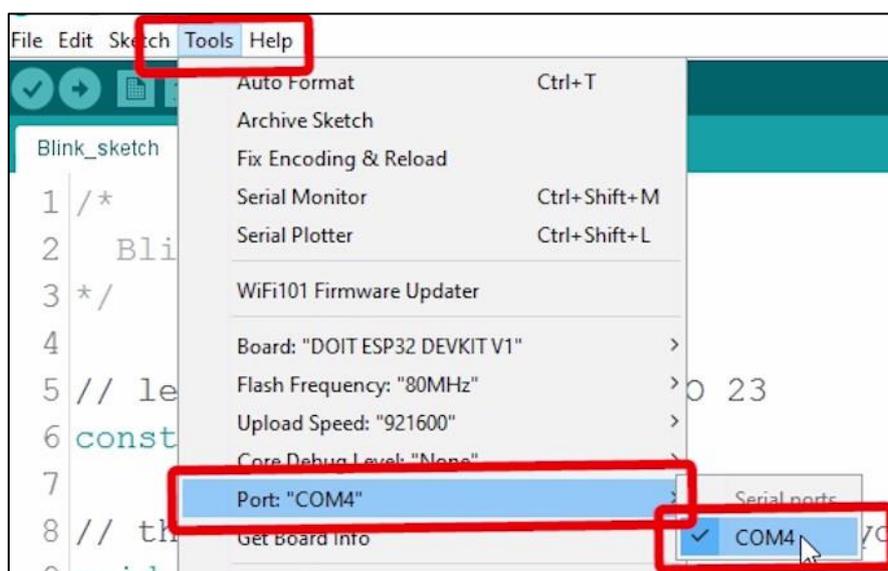
(This schematic uses the ESP32 DEVKIT V1 module version with 30 GPIOs – if you're using another model, please check the pinout for the board you're using.)

Preparing the Arduino IDE

After connecting an LED to the ESP32 board GPIO 23, you need to go to **Tools** ▶ **Board**, scroll down to the ESP32 section and select the name of your ESP32 board that you found earlier. In my case, it's the **DOIT ESP32 DEVKIT V1** board.



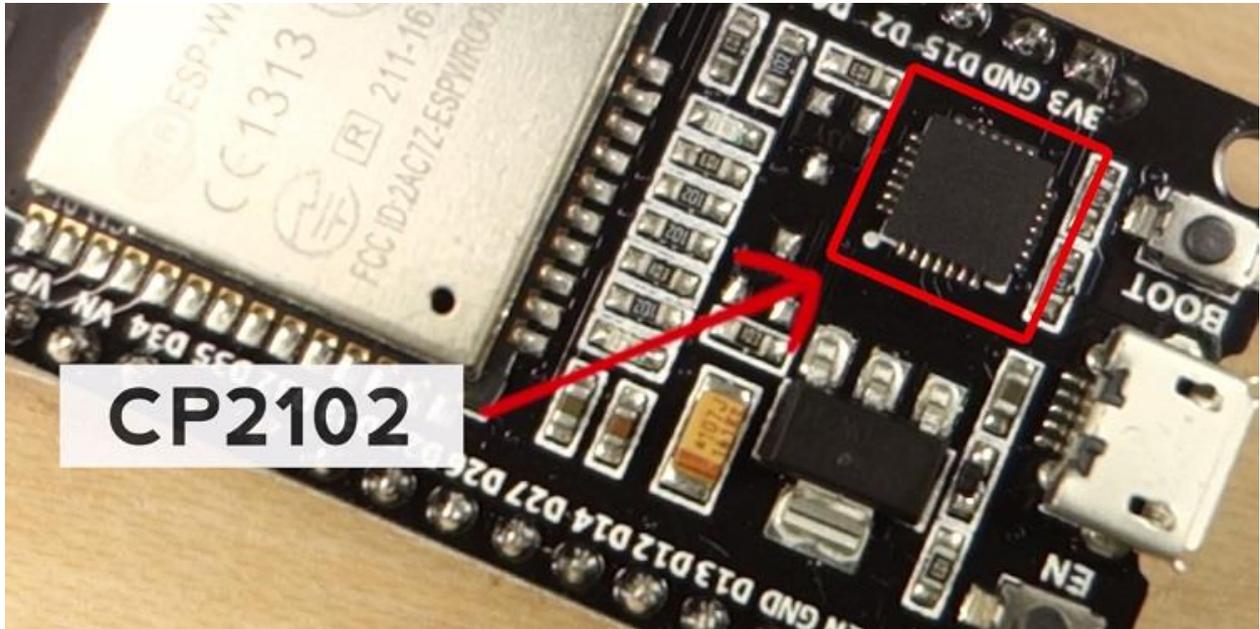
While having the ESP32 plugged to your computer. Go to **Tools** ► **Port** and select a COM port available.



Installing Drivers

If you don't see your ESP's COM port available, this means you don't have the 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.



Go to Google and search for your particular chip to find the drivers and install them in your operating system.

A search bar containing the text 'CP2102 driver download' and a microphone icon on the right side. Two buttons: 'Google Search' and 'I'm Feeling Lucky'. A mouse cursor is hovering over the 'Google Search' button.

Note: You can download the CP2102 drivers on the [Silicon Labs](#) website.

SILICON LABS

简体中文 日本語 Log In | Register

Parametric Search | Cross-Reference Search

Search silabs.com GO

About Products Solutions Community & Support

Silicon Labs » Products » Development Tools » Software » USB to UART Bridge VCP Drivers

CP210x USB to UART Bridge VCP Drivers

The CP210x USB to UART Bridge Virtual COM Port (VCP) drivers are required for device operation as a Virtual COM Port to facilitate host communication with CP210x products. These devices can also interface to a host using the direct access driver. These drivers are static examples detailed in application note 197: The Serial Communications Guide for the CP210x, download an example below:

AN197: The Serial Communications Guide for the CP210x

Download Software

The CP210x Manufacturing DLL and Runtime DLL have been updated and must be used with v6.0 and later of the CP210x Windows VCP Driver. Application Note Software downloads affected are AN144SW.zip, AN205SW.zip and AN223SW.zip. If you are using a 5.x driver and need support you can download archived Application Note Software.

[Legacy OS software and driver package download links and support information >](#)

Download for Windows 10 Universal (v10.1.1)

Platform	Software	Release Notes
Windows 10 Universal	Download VCP (2.3 MB)	Download VCP Revision History

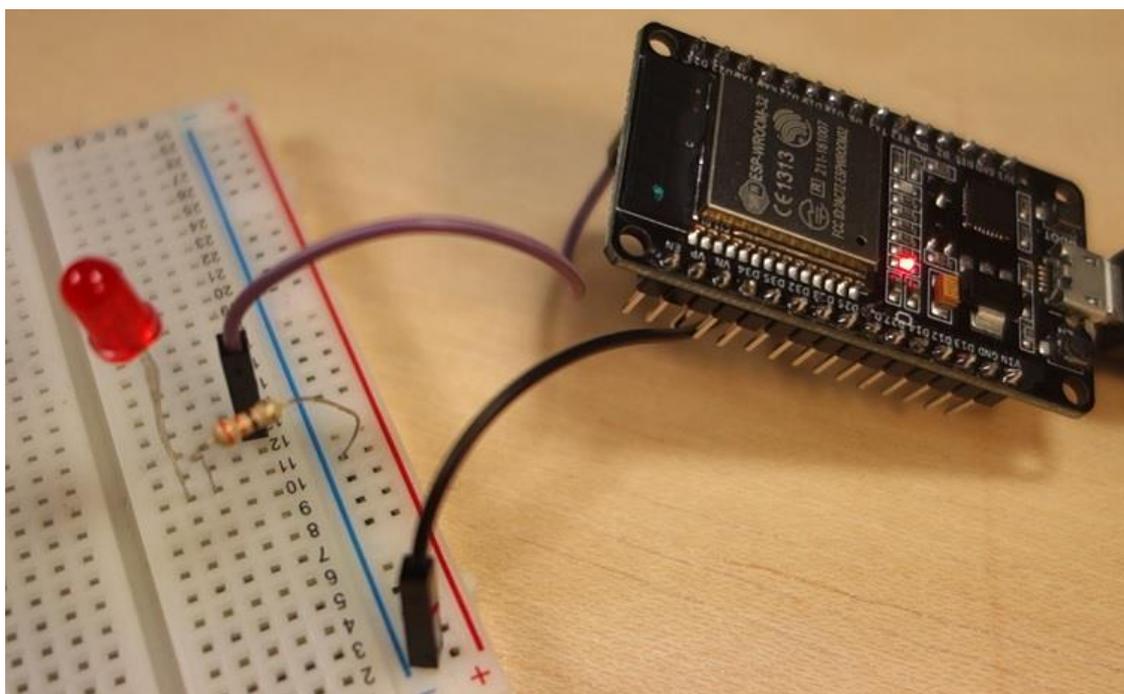
After they are installed, restart the Arduino IDE, and you should see the COM port in the **Tools** menu.

Uploading the Sketch

Go back to the Arduino IDE, press the Arduino IDE upload button and wait a few seconds while it compiles and uploads your sketch.



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



Wrapping Up

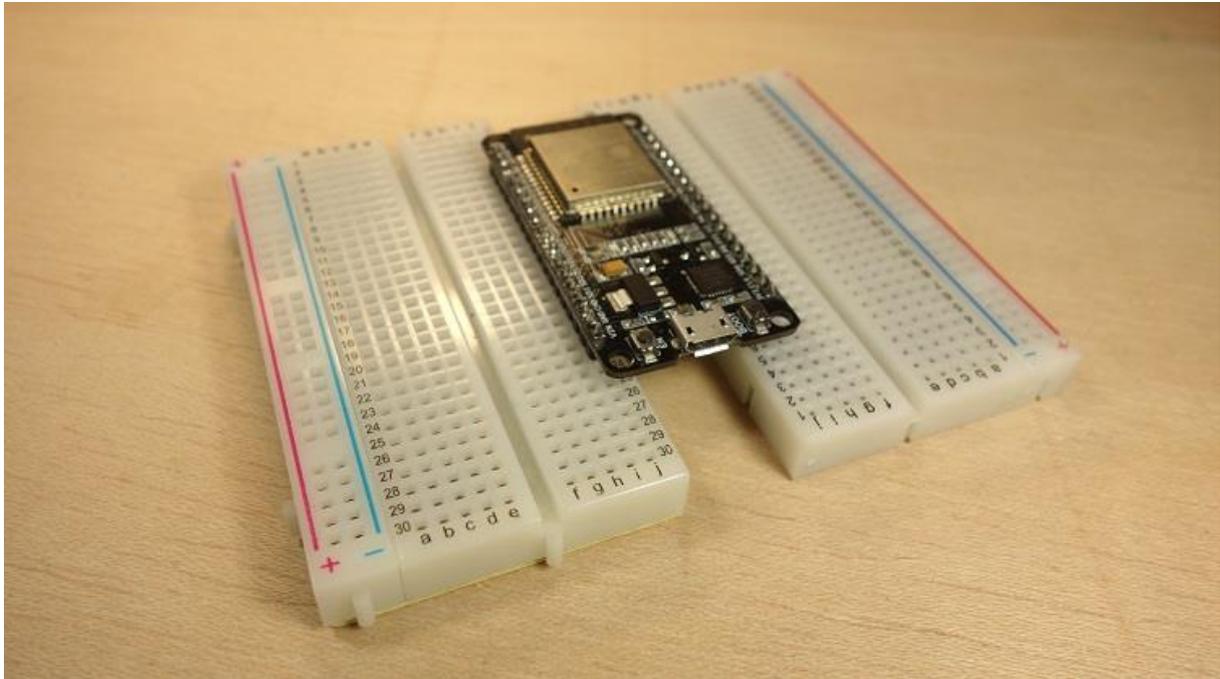
Here's the steps that you should follow:

- Go to the product page to see if you find any relevant information that allows you to identify your board;
- Check the back of your board, to see the ESP32 name;
- Use 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 as a reference;
- While wiring any circuit, always check that you're connecting the components to the right GPIO;
- To upload code, select the right board in the Arduino IDE and COM port.

That's it. I hope this was helpful and makes it easier for you to use this course, regardless of your ESP32 development board.

Unit 4 - ESP32 Breadboard Friendly

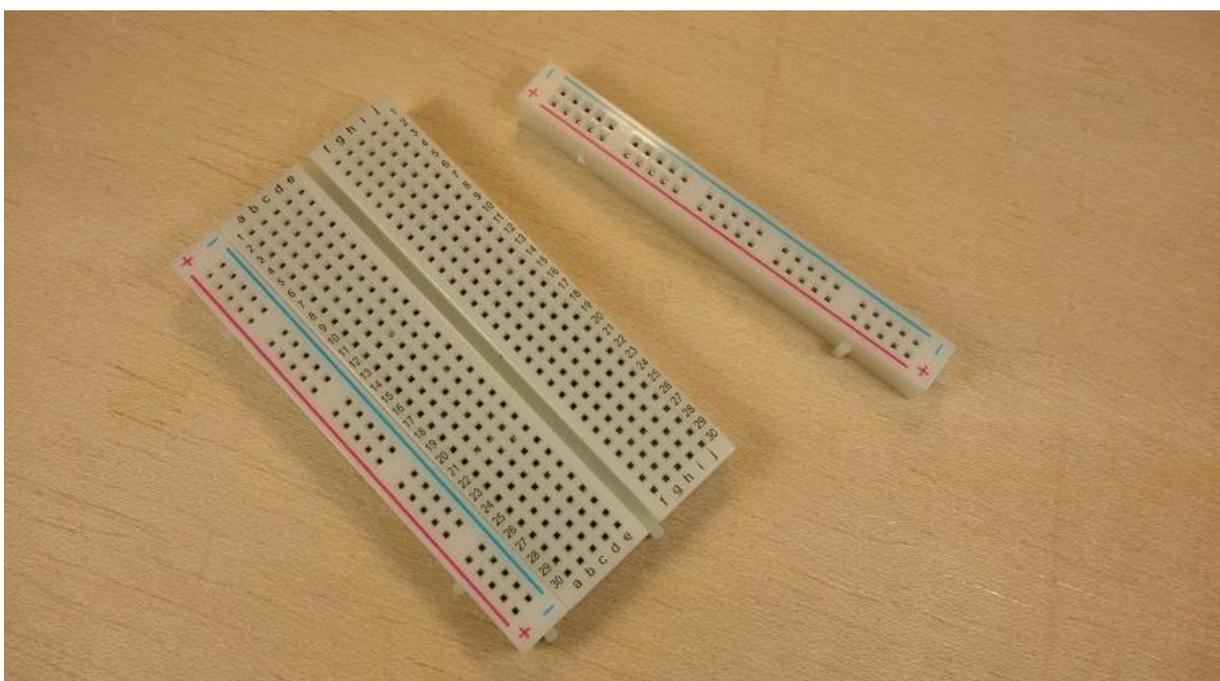
The ESP32 is not breadboard friendly. However, you can make your breadboards ESP32-friendly. In this Unit, we'll show you how.



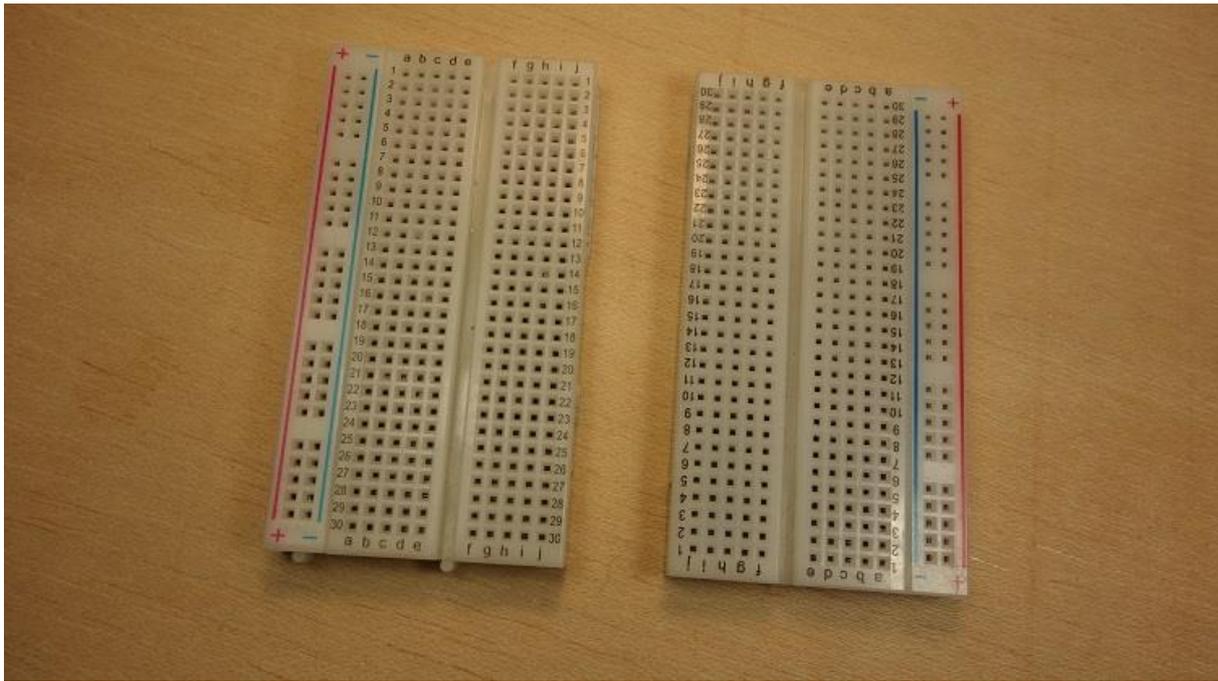
Note: this tip was shared by Geo Massar, one of our readers. Thank you!

ESP32 on Two Breadboards

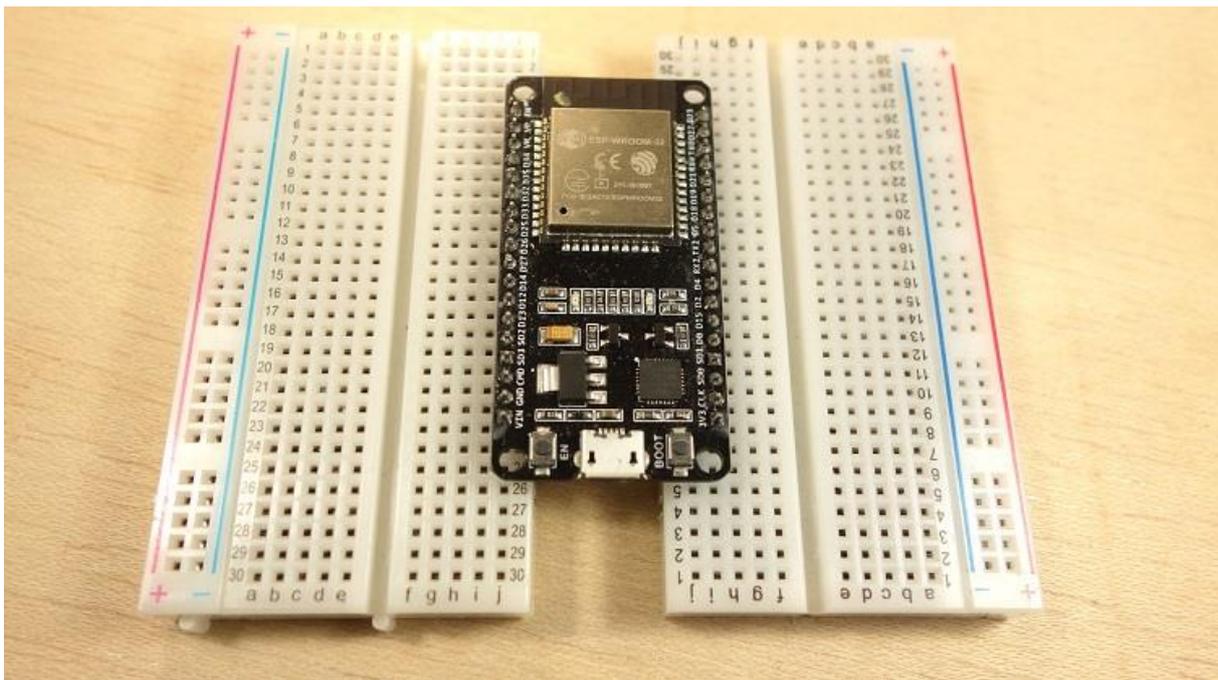
Usually, breadboards come with two power rail on each side. Detach one power rail from your breadboard, as show in the figure below.



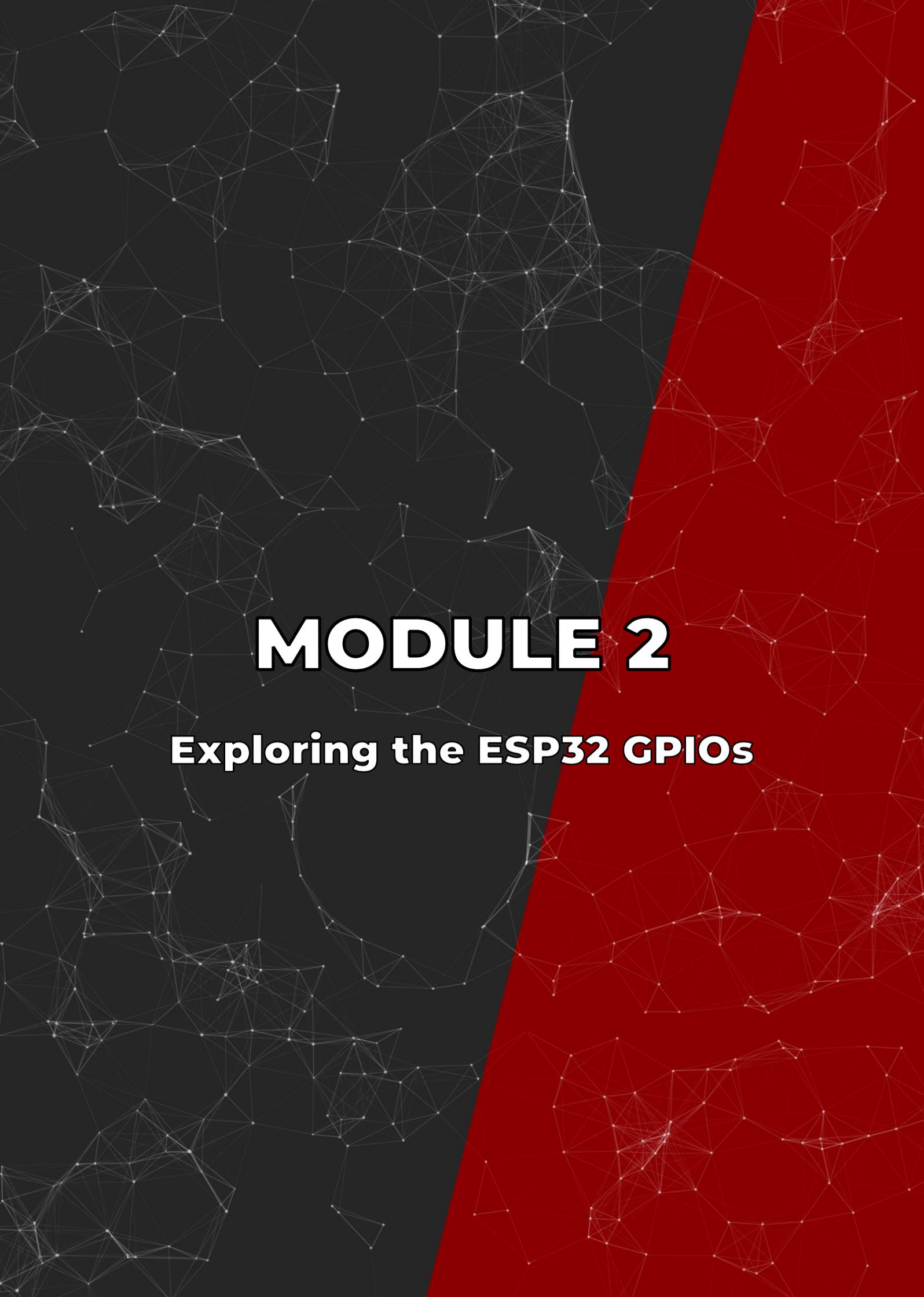
Repeat this process for another breadboard:



Then, insert the ESP32 into the two breadboards.



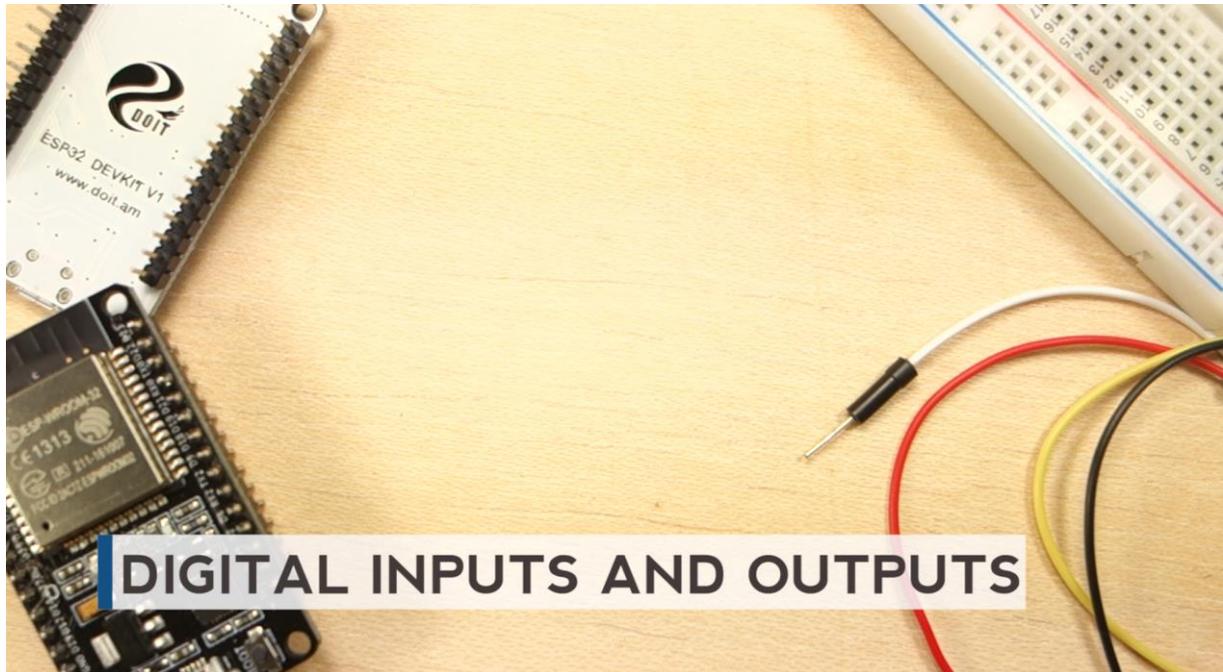
Now, it is easier to prototype circuits with the ESP32 using breadboards.



MODULE 2

Exploring the ESP32 GPIOs

Unit 1 - ESP32 Digital Inputs and Outputs



In this section, we're going to show you how to read digital inputs like a button switch, and how to control a digital output, like an LED. If you've programmed the Arduino or ESP8266 before with Arduino IDE, this is not new for you.

digitalWrite()

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

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

digitalRead()

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

```
digitalRead(GPIO)
```

Project Example

Let's just make a simple example to see how these 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.


```

// 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);
  }
}

```

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 GPIO4 and the LED is connected to GPIO16. When using the Arduino IDE with the ESP 32, 4 corresponds to GPIO4 and 16 corresponds to GPIO16.

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 state. Either INPUT or OUTPUT.

```

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

```

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

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

```

buttonState = digitalRead(buttonPin);

```

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

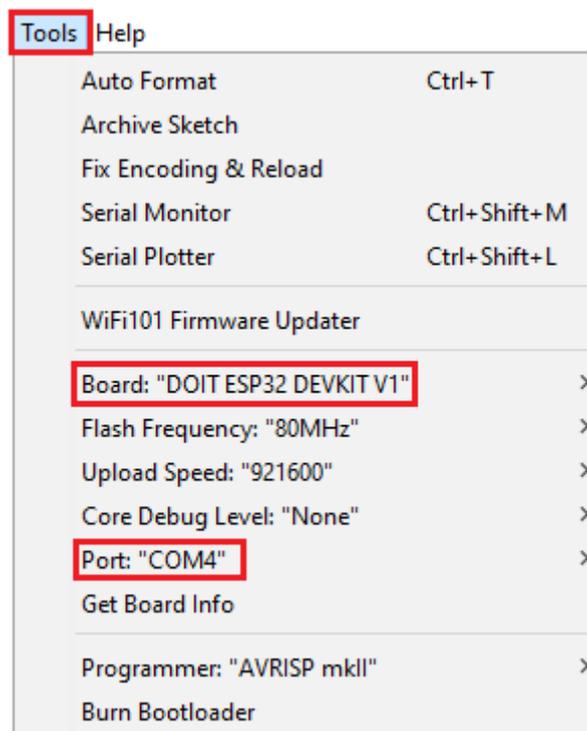
```
if (buttonState == HIGH) {  
  // turn LED on  
  digitalWrite(ledPin, HIGH);  
}
```

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

Before clicking the upload button, go to **Tools** ▶ **Board**, and select the board you're using. In my case. It's the **DOIT ESP32 DEVKIT V1 board**. Also don't forget to select your ESP32's COM port.



Now, press the upload button.

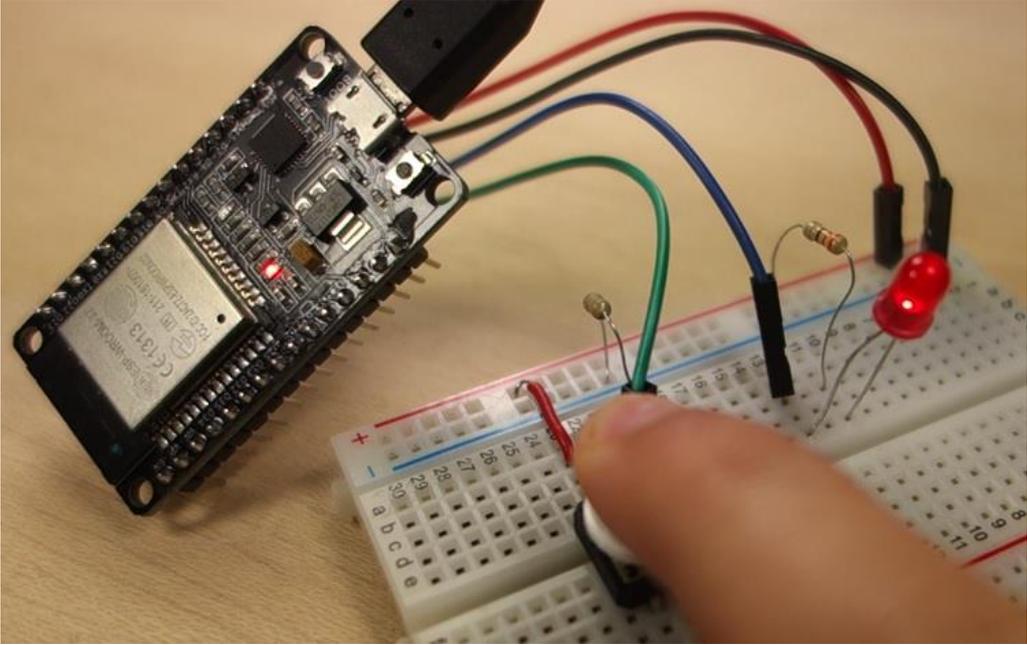


Then, wait for the "Done uploading." message:

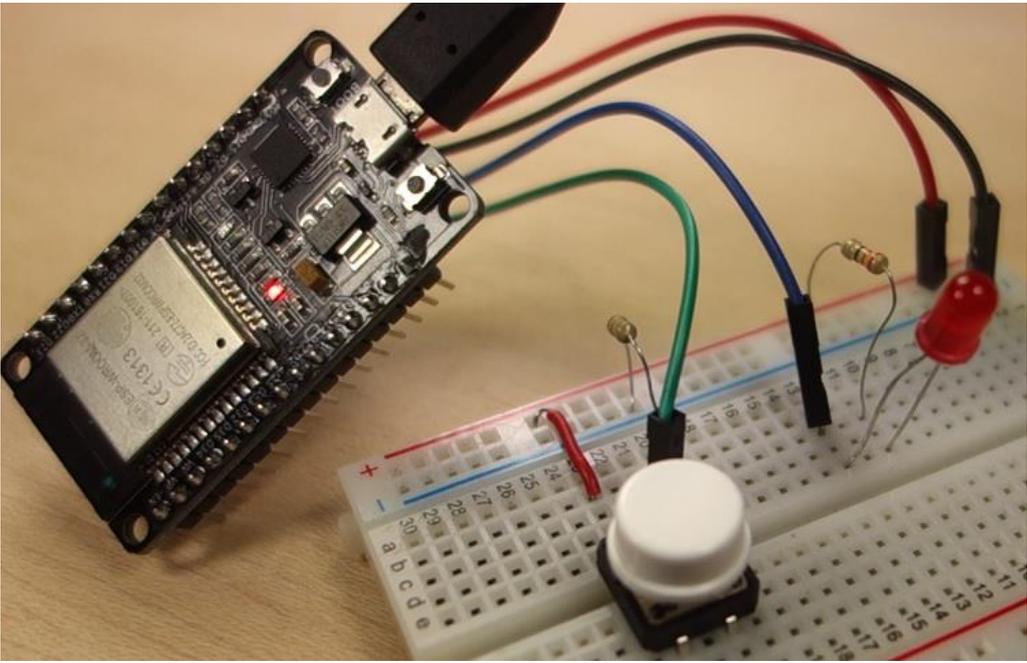
```
Done uploading
writing at 0x00042000... (84 %)
Writing at 0x00050000... (89 %)
Writing at 0x00054000... (94 %)
```

Testing Your Project

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



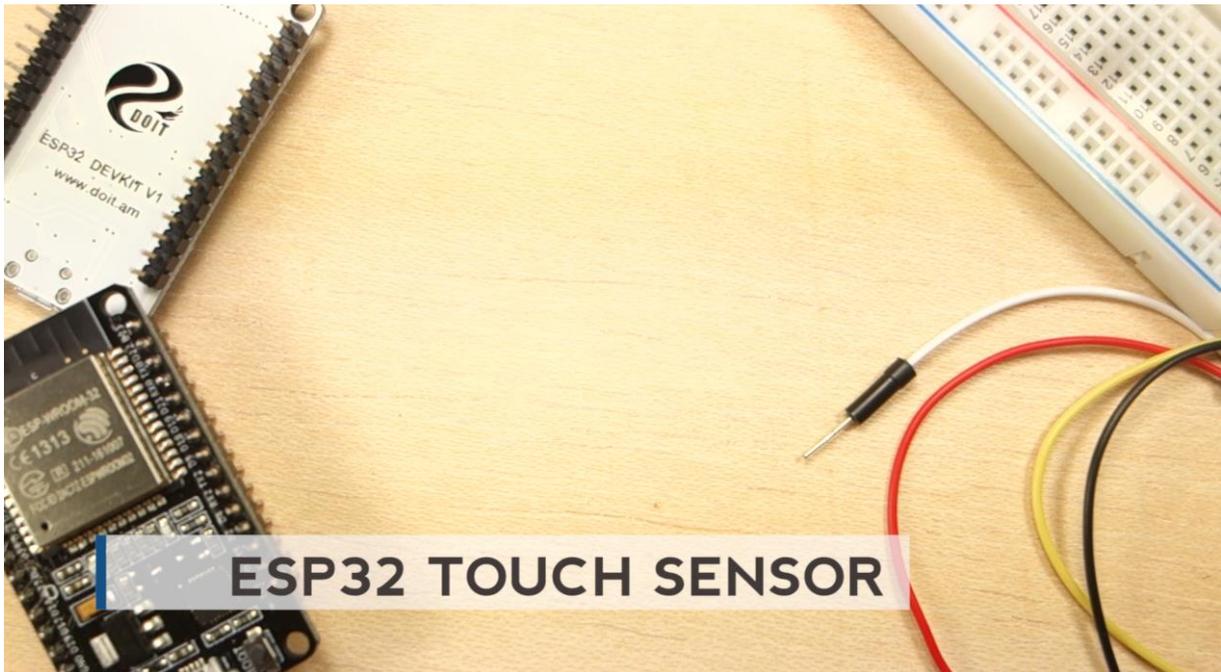
And turn off when you release it:



Wrapping Up

In summary, in this section you've learned how to read digital inputs and how to control digital outputs with the ESP32 using the Arduino IDE.

Unit 2 - ESP32 Touch Sensor

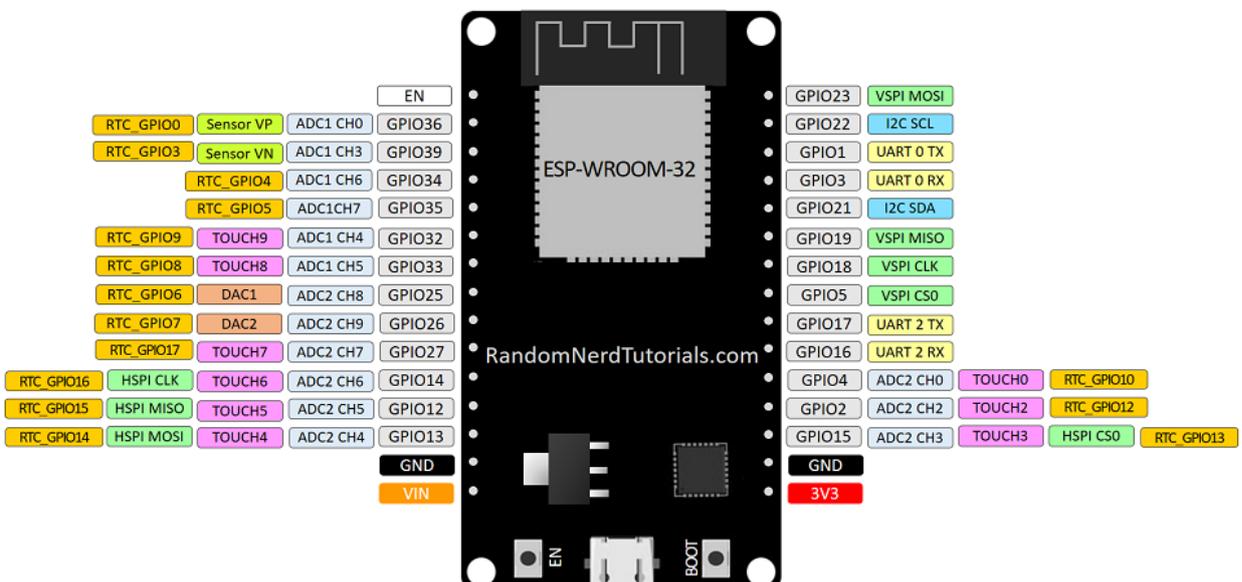


In this section you're going to learn about the ESP32 touch sensor.

Introducing the ESP32 Touch Sensor

The ESP32 has 10 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. Take a look at your board pinout to locate the 10 different touch sensors – the touch sensitive pins are highlighted in pink color.



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

Note: Touch sensor 1 is GPIO 0. However, it's not available as a pin in this particular ESP32 development board (version with 30 GPIOs). GPIO 0 is available on the new version of this board with 36 pins.

Note: at the time of writing this unit, there is an issue with touch pin assignment in Arduino IDE. GPIO 33 is swapped with GPIO 32 in the assignment. This means that if you want to refer to GPIO 32 you should use T8 in the code. If you want to refer to GPIO33 you should use T9. If you don't have this issue, please ignore this note.

touchRead()

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

```
touchRead(GPIO)
```

Code – Reading the Touch Sensor

Let's see how that function works by using an example from the library. In the Arduino IDE, go to **File** ▶ **Examples** ▶ **ESP32** ▶ **Touch** and open the **TouchRead** sketch.

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/TouchRead/TouchRead.ino>

```
// 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(4)); // get value of Touch 0 pin = GPIO 4
  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 sensor.

```
Serial.println(touchRead(4));
```

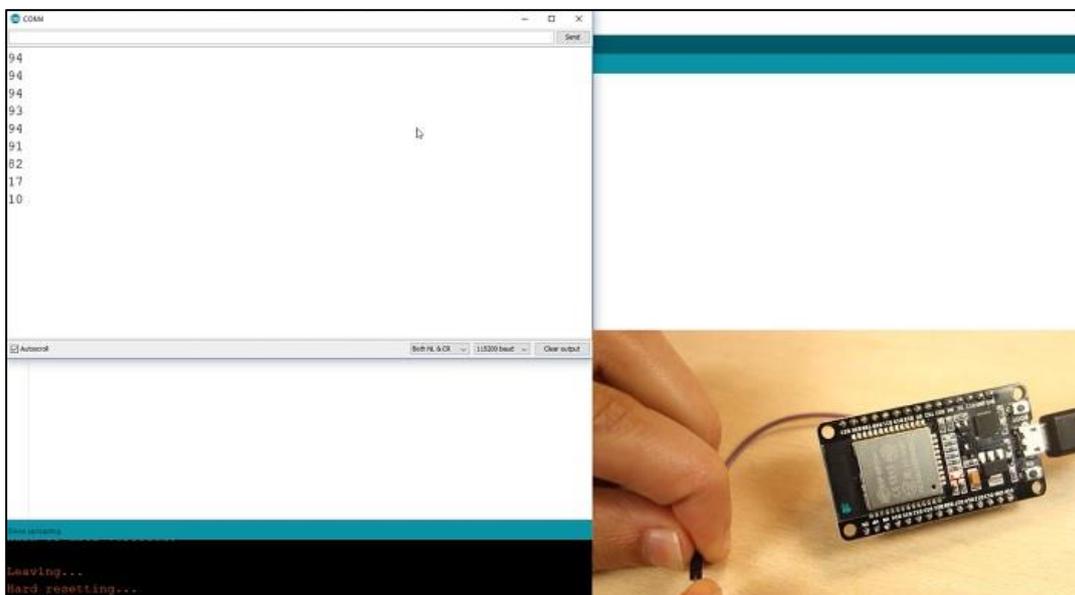
It uses the `touchRead()` function, and you pass as an argument the pin you want to read. In this case the example uses T0, which is the touch sensor 0, in GPIO 4.

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

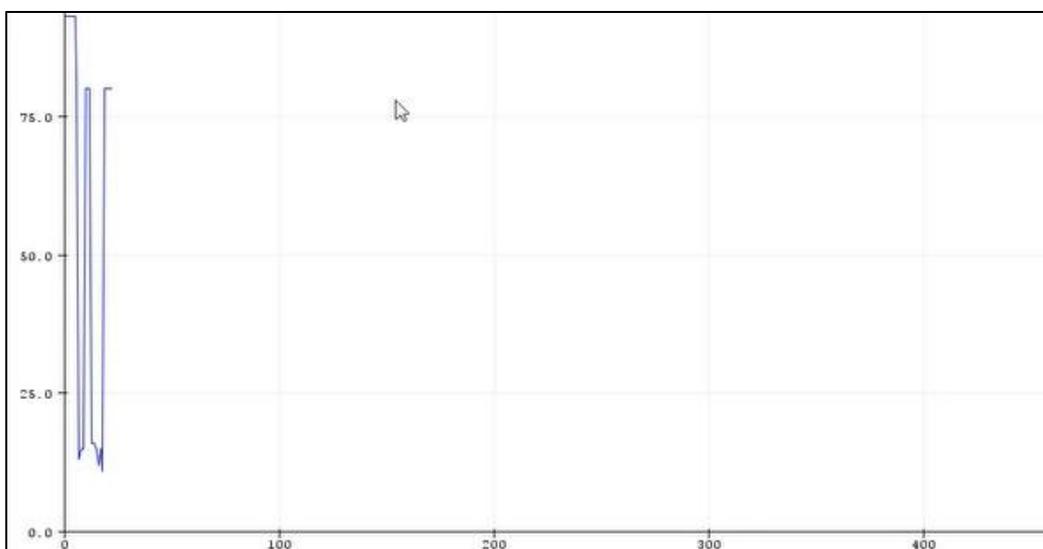
Testing the Example

Connect a jumper wire to GPIO 4. You will touch the metal part of this wire so that it senses the touch. In the Arduino IDE window, go to **Tools** and open the **Serial Monitor** at a baud rate of 115200. You'll see the new values being displayed every second.

Touch the wire connected to GPIO 4 and you'll see the values decreasing.



You can also use the serial plotter to better see the values. Close the serial monitor, go to **Tools** ▶ **SerialPloter**.



Touch Sensitive LED

How can you use this feature to control outputs? Let's build a simple touch controlled LED circuit.

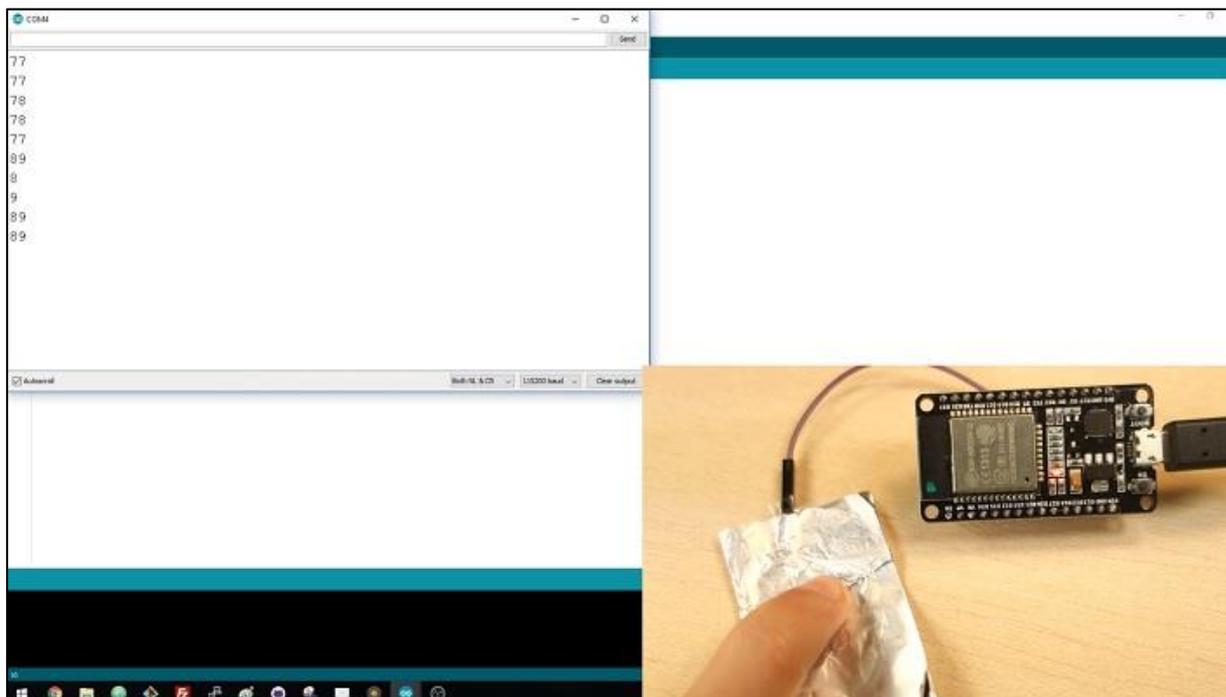
Finding the Threshold

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



With the same code running, go back to the serial monitor.

Now, touch the aluminum foil, and you'll see the values changing again.



In our case, when we aren't touching the pin, the normal value is above 70. And when we touch the aluminum foil it drops to some value below 10.

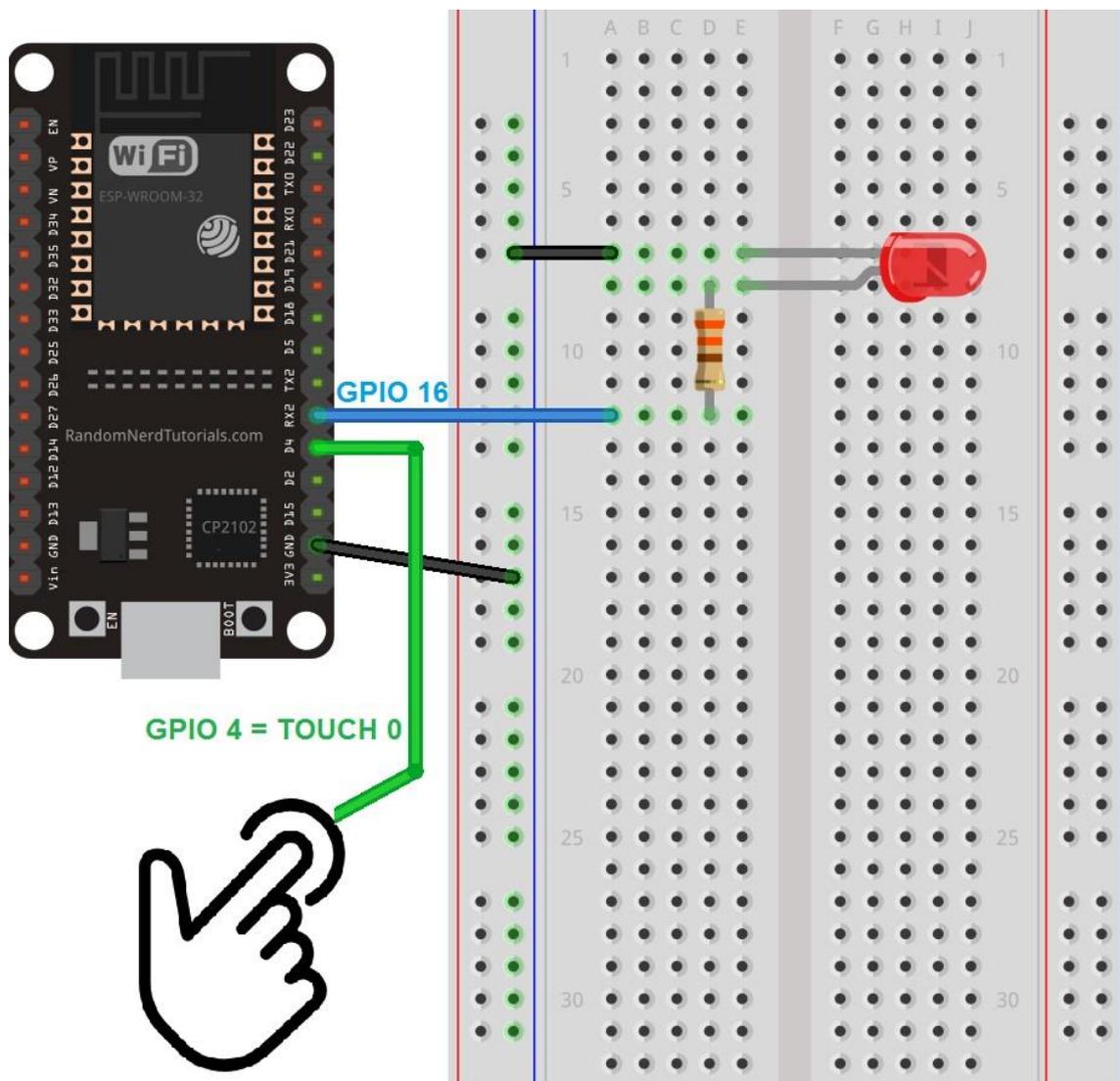
So, we can set a threshold value, and when the reading goes below that value, an LED lights up. A good threshold value in this case is 20, for example.

Schematic

Add an LED to your circuit by following this schematic. 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](#)
- [330 Ohm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)



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

Code

Copy the following code to your Arduino IDE.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Touch_Sensitive_LED/Touch_Sensitive_LED.ino

```
// 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;

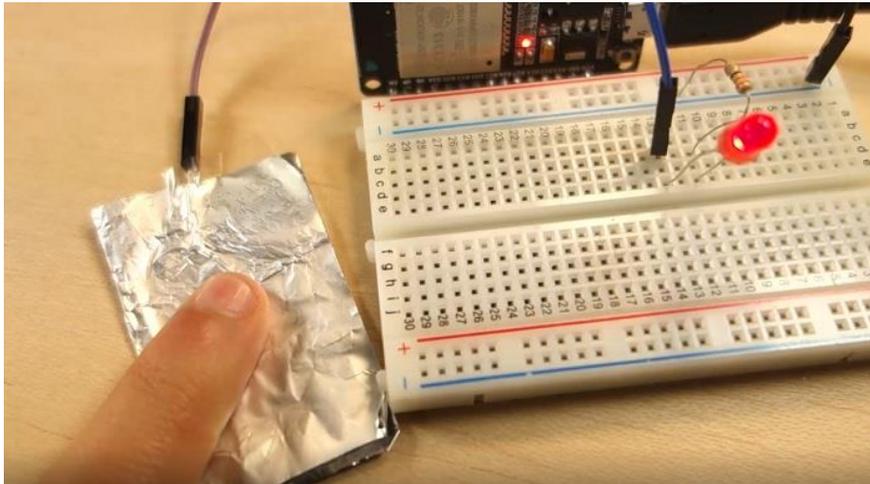
void setup(){
  Serial.begin(115200);
  delay(1000); // give me time to bring up serial monitor
  // initialize the LED pin as an output:
  pinMode (ledPin, OUTPUT);
}

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.

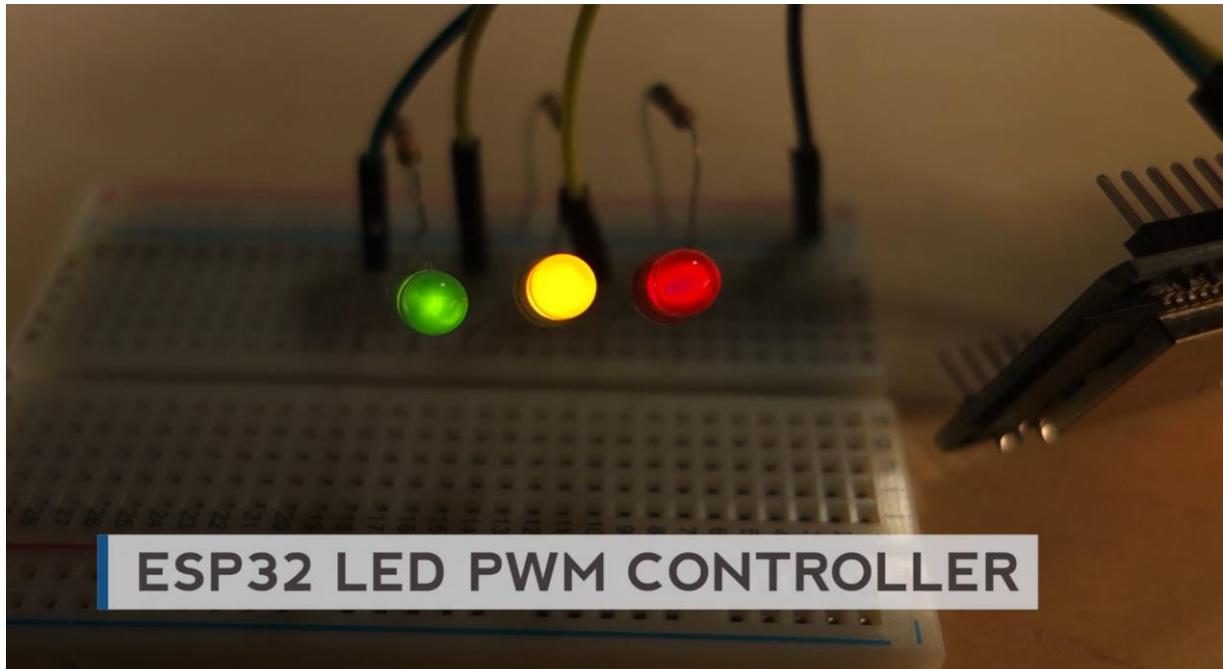


Wrapping Up

In summary, in this section you've learned:

- The ESP32 has 10 capacitive touch GPIOs, but only 9 are available with the ESP32 DEVKIT V1 DOIT board (with 30 pins).
- 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.

Unit 3 - ESP32 Pulse-Width Modulation (PWM)



In this section you'll learn how to dim an LED using the LED PWM controller of the ESP32 with the Arduino IDE.

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

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

- 1) First, you need to choose a PWM channel. There are 16 channels from 0 to 15.
- 2) Then, you need to set the PWM signal frequency. For an LED, a frequency of 5000 Hz is fine to use.
- 3) You also need to set the signal's duty cycle resolution: you have resolutions from 1 to 16 bits. We'll use 8-bit resolution, which means you can control the LED brightness using a value from 0 to 255.
- 4) Next, you need to specify to which GPIO or GPIOs the signal will appear upon. For that you'll use the following function:

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

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

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

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

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

Dimming an LED

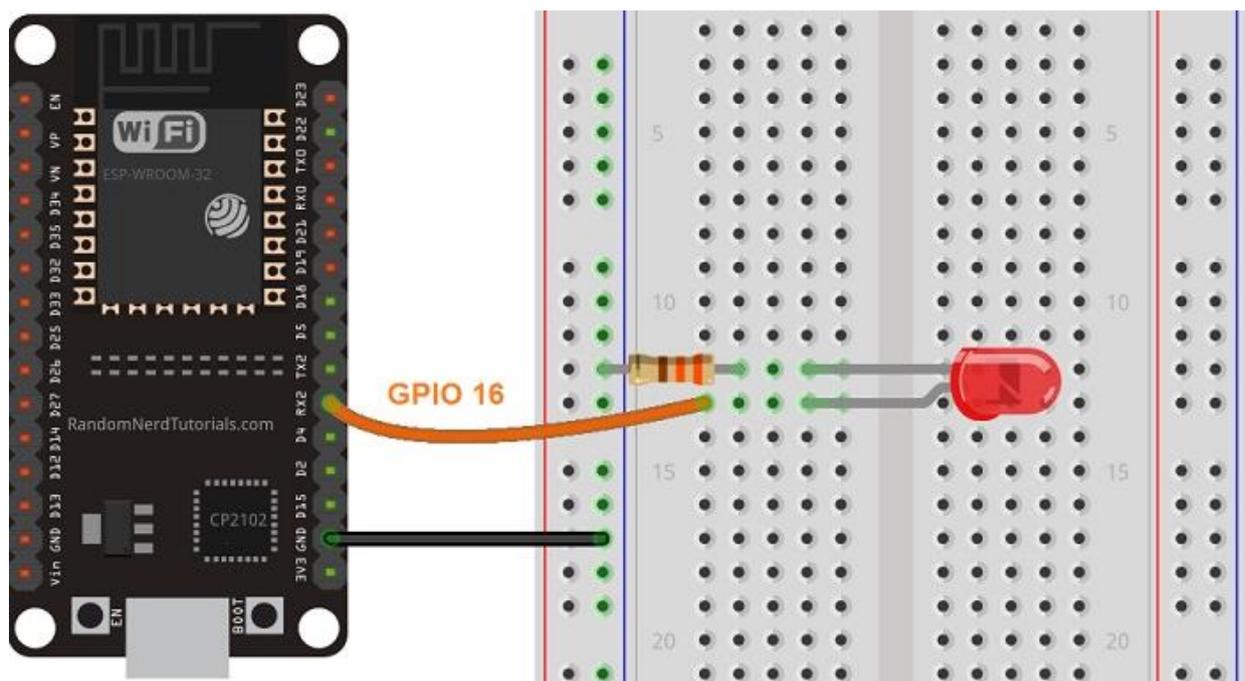
Let's go through a simple example to see how to use the ESP32 LED PWM controller using the Arduino IDE.

Schematic

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [330 Ohm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)

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



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

Note: You can use any pin you want, as long as it can act as an output. All pins that can act as outputs can be used as PWM pins.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/LED_PWM_Example_1/LED_PWM_Example_1.ino

Code

Open your Arduino IDE and copy the following code.

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

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

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

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

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

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

You start by defining the pin the LED is attached to. In this case the LED is attached to GPIO16.

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

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

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

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

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

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

signal in the `ledPin` GPIO that corresponds to GPIO 16. The channel that generates the signal is the `ledChannel` that corresponds to channel 0.

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

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

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

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

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

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

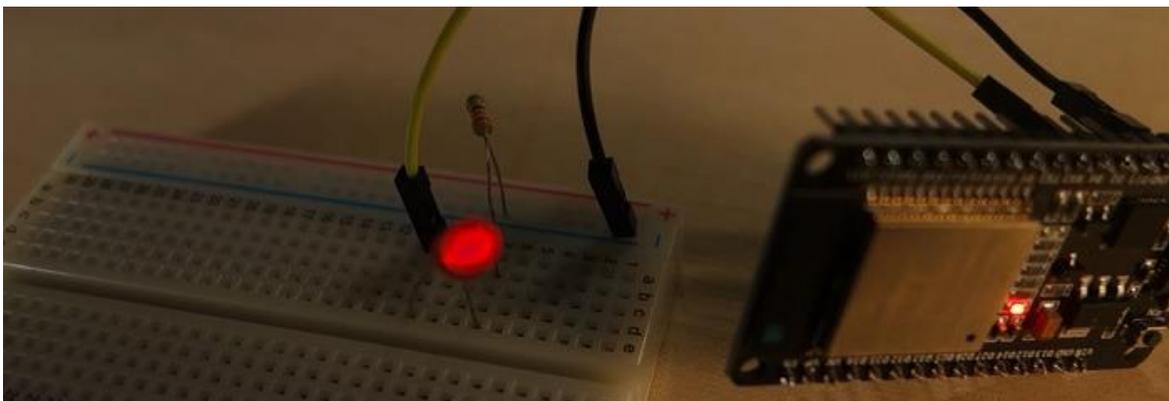
```
ledcWrite(ledChannel, dutyCycle);
```

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

Testing the Example

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

Look at your circuit. You should have a dimmer LED that increases and decreases brightness.



Getting the Same Signal on Different GPIOs

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

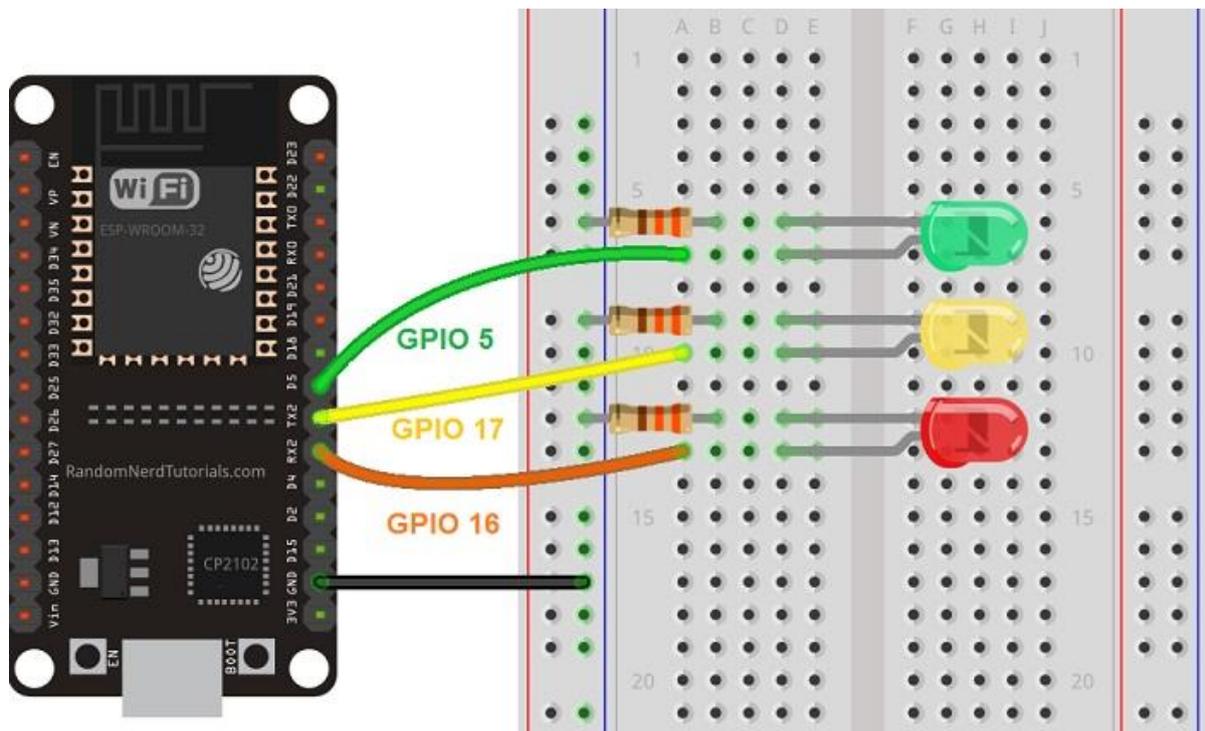
Let's modify the previous example to dim 3 LEDs using the same PWM signal from the same channel.

Schematic

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [3x 5mm LED](#)
- [3x 330 Ohm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)

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



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

Code

Copy the following code to your Arduino IDE.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/LED_PWM_Example_2/LED_PWM_Example_2.ino

```

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

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

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

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

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

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

```

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

```

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

```

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

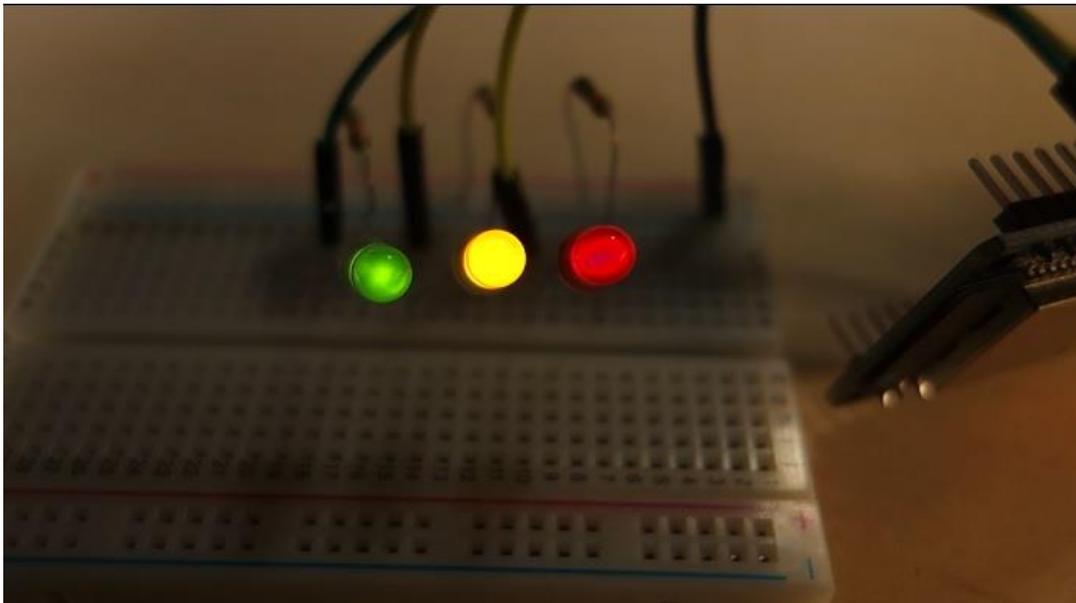
```

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

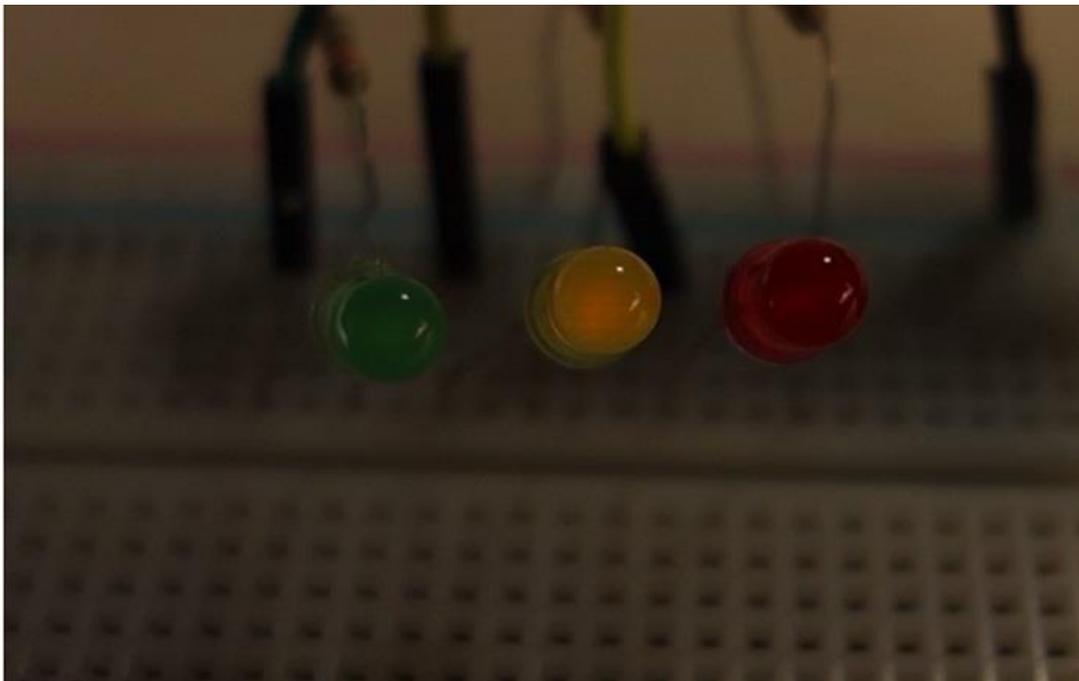
```

Testing the Project

Upload the new sketch to your ESP32. Make sure you have the right board and COM port selected. Now, take a look at your circuit:



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

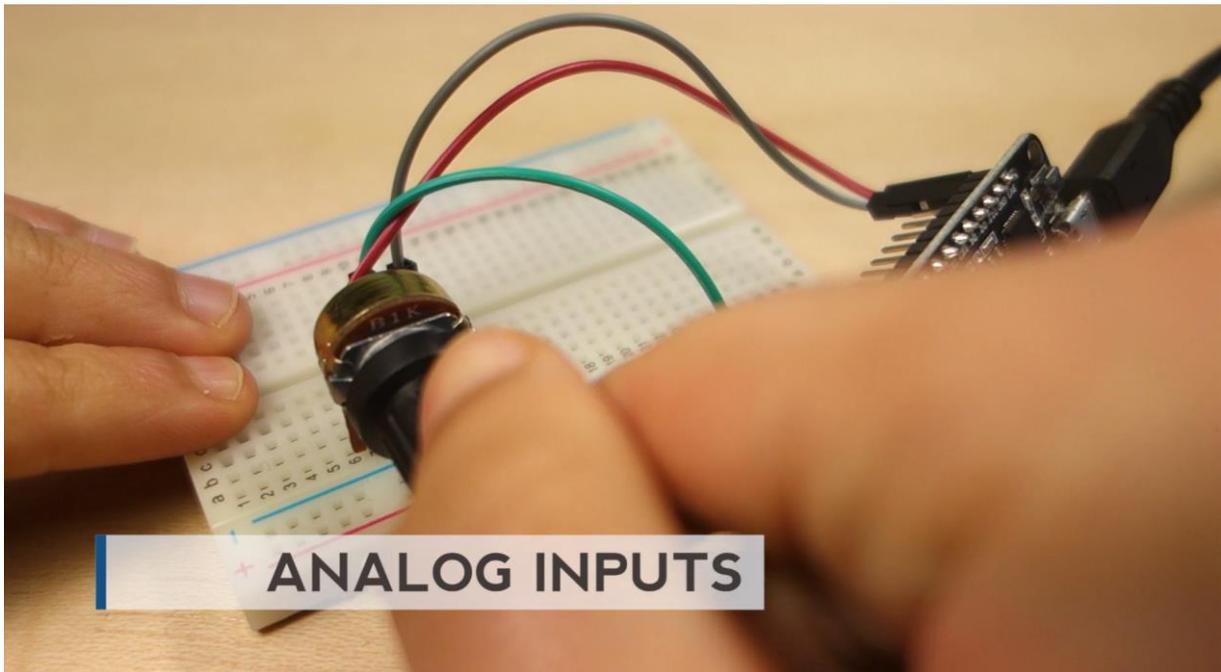


Wrapping Up

In summary, in this section you've learned how to use the LED PWM controller of the ESP32 with the Arduino IDE to dim an LED.

The concepts learned can be used to control other outputs with PWM by setting the right properties to the signal.

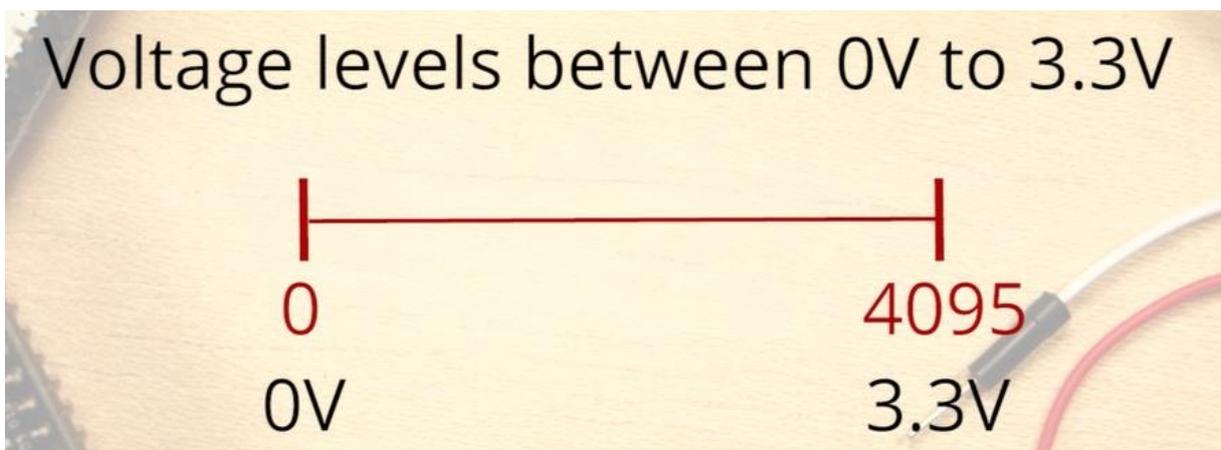
Unit 4 - ESP32 Reading Analog Inputs



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.

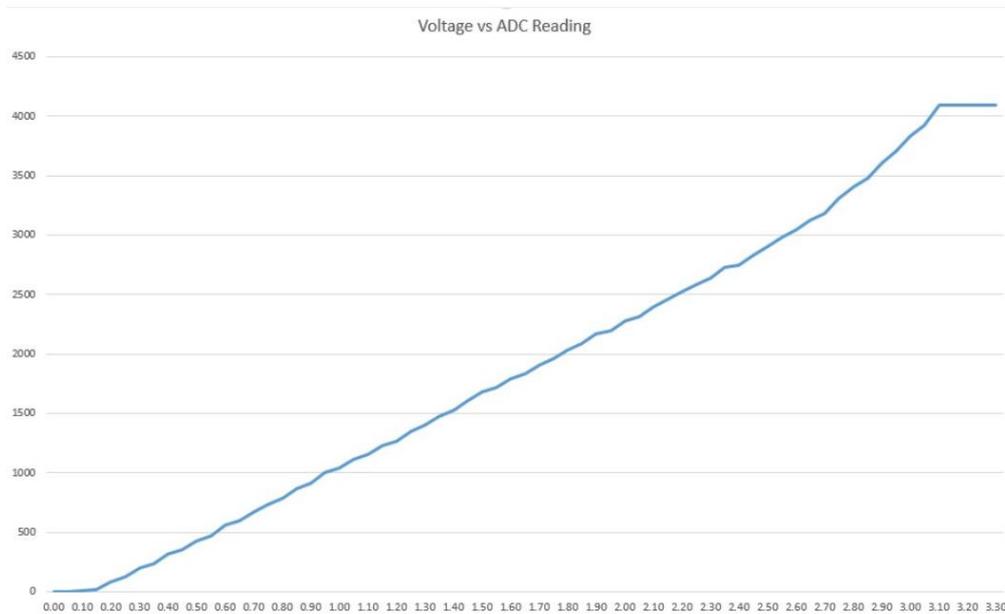
Analog Inputs

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.



ADC 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 low 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](#).

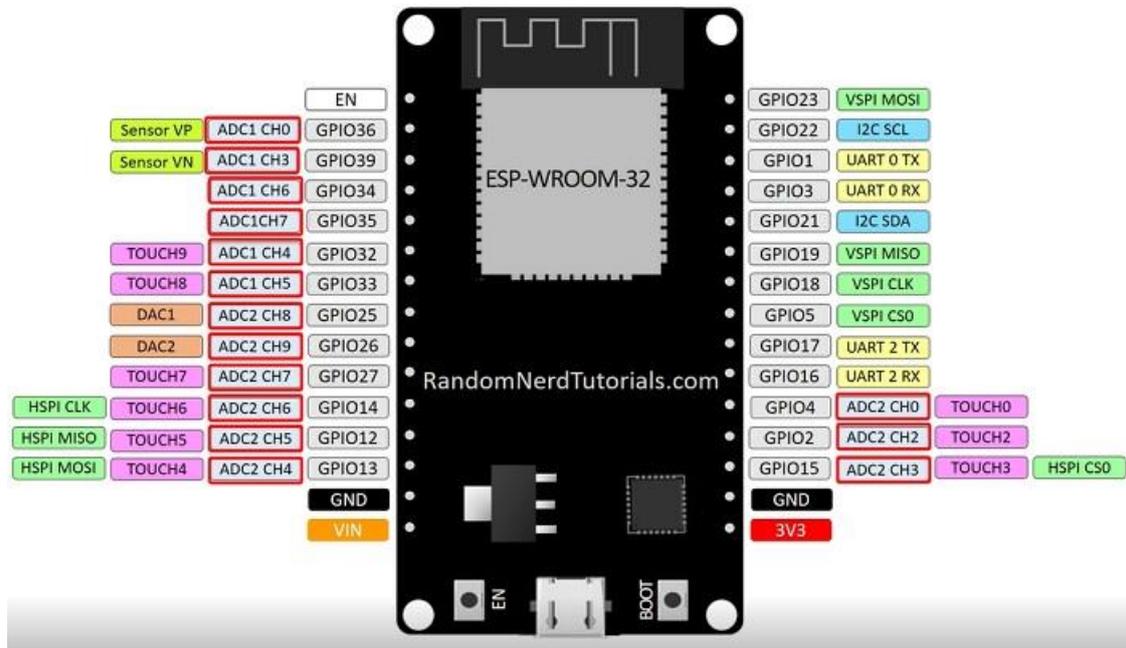
analogRead()

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

```
analogRead(GPIO)
```

The ESP32 supports measurements in 18 different channels. But only 15 are available in the DEVKIT V1 DOIT board (version with 30 GPIOs). These are the ADC pins. Grab your ESP32 board pinout and locate the ADC pins. These are highlighted with a red border in the figure below.

Note: the ADC pins marked as ADC2 don't work well when Wi-Fi is used.



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

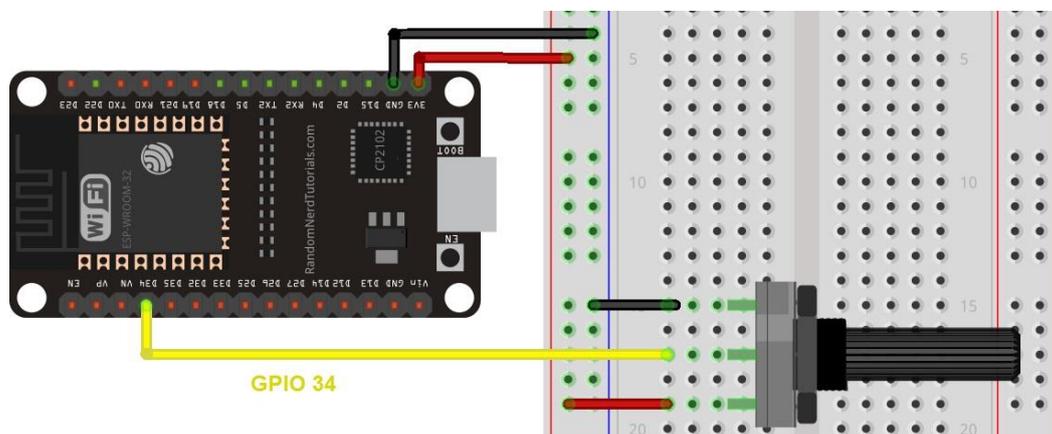
To see how this works let's make a simple example to read an analog value from a potentiometer.

Schematic

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.



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

Code

Copy the following code to your Arduino IDE.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Analog_Input_Pot/Analog_Input_Pot.ino

```
// 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.

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);
```

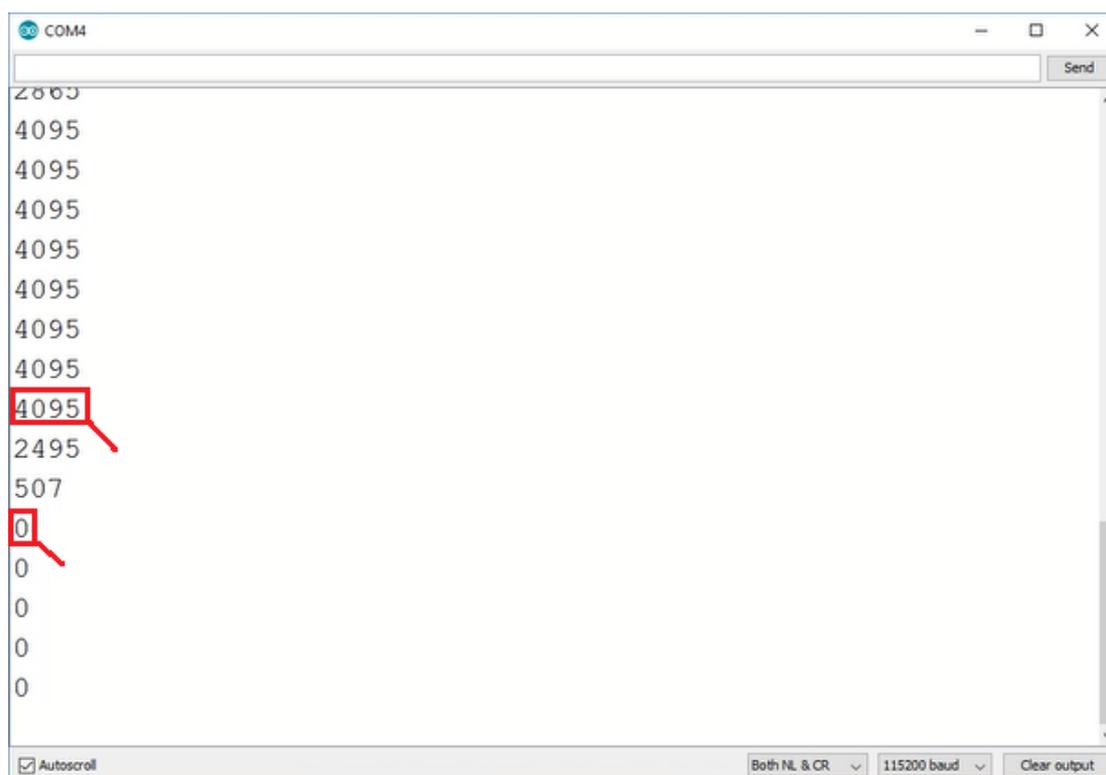
Upload the code provided to your ESP32. Before uploading any code to your ESP, always check you have the right board and COM port selected.

Testing the Example

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.

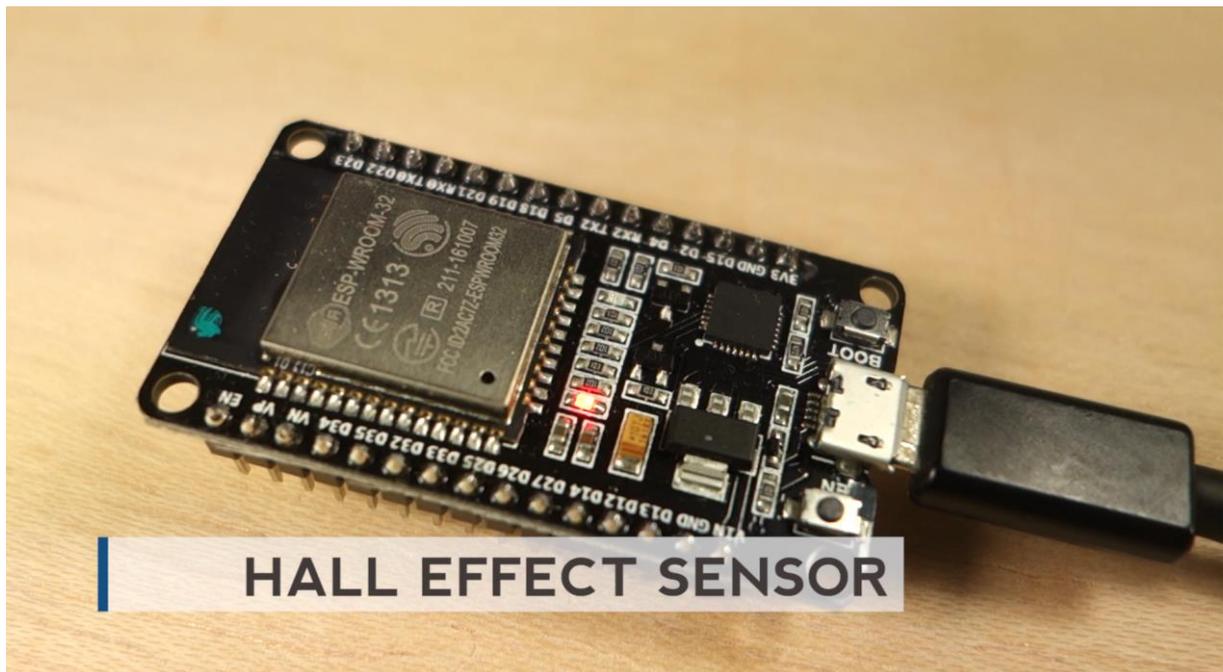


Wrapping Up

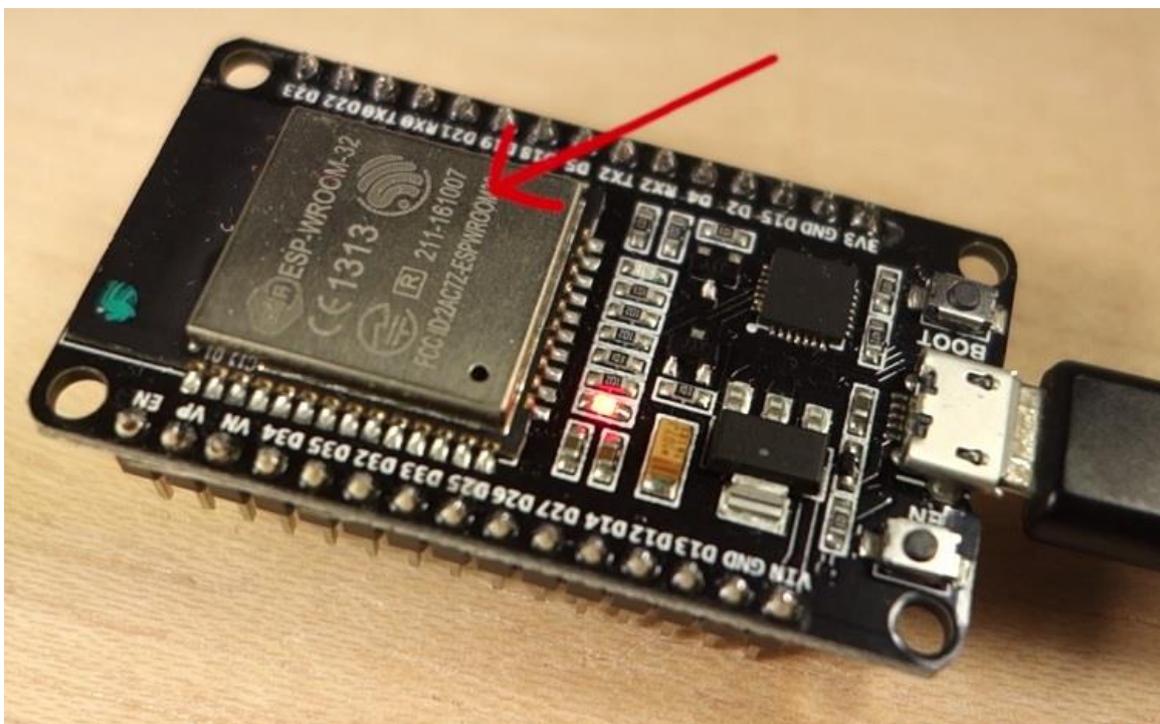
In summary:

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

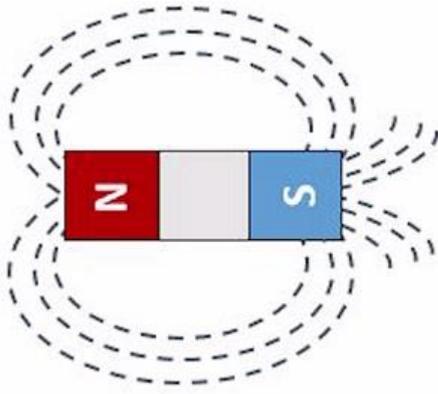
Unit 5 - ESP32 Hall Effect Sensor



The ESP32 features a built-in hall effect sensor which is located as shown in the figure below (behind that metal cap):



A hall effect sensor can detect variations in the magnetic field in its surroundings. The greater the magnetic field, the greater the output voltage.



Hall effect sensor

A hall effect sensor can be combined with a threshold detection to act as a switch. Hall effect sensors are mainly used to:

- Detect proximity;
- Calculate positioning;
- Count the number of revolutions of a wheel;
- Detect a door closing;
- And much more.

Read Hall Effect Sensor

Reading the hall effect sensor measurements with the ESP32 using the Arduino IDE is as simple as using the `hallRead()` function. In your Arduino IDE, go to **File** ▶ **Examples** ▶ **ESP32** ▶ **HallSensor**:

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Hall_Effect_Sensor/Hall_Effect_Sensor.ino

```
// Simple sketch to access the internal hall effect detector on the esp32.
// values can be quite low.
// Brian Degger / @sctv

int val = 0;

void setup() {
  Serial.begin(9600);
}

// put your main code here, to run repeatedly
void loop() {
  // read hall effect sensor value
  val = hallRead();
  // print the results to the serial monitor
  Serial.println(val);
  delay(1000);
}
```

This example simply reads the hall sensor measurements and displays them on the Serial monitor.

```
val = hallRead();  
// print the results to the serial monitor  
Serial.println(val);
```

Add a delay of one second in the loop, so that you can actually read the values.

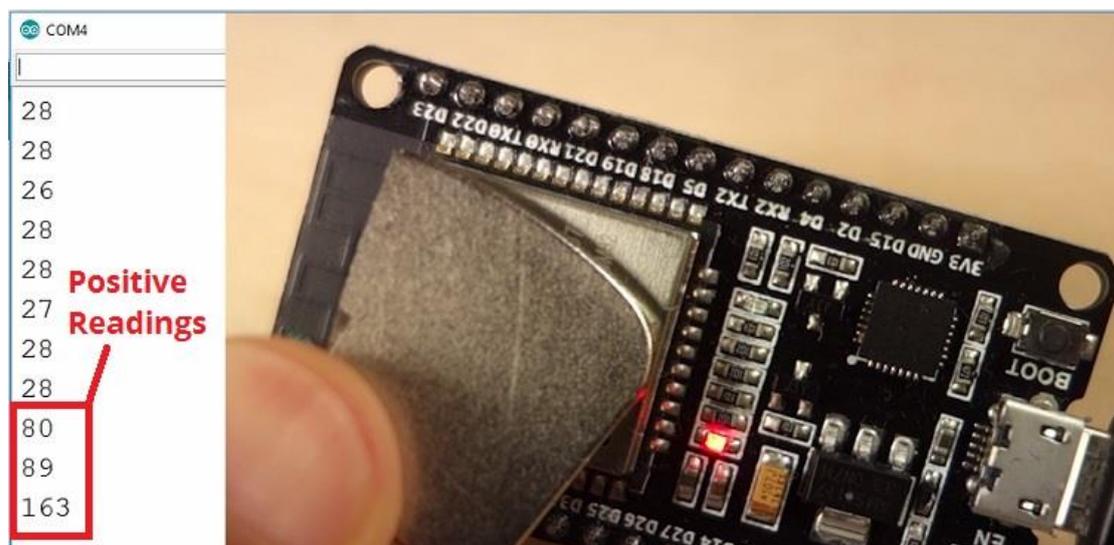
```
delay(1000);
```

Upload the code to your ESP32 board:

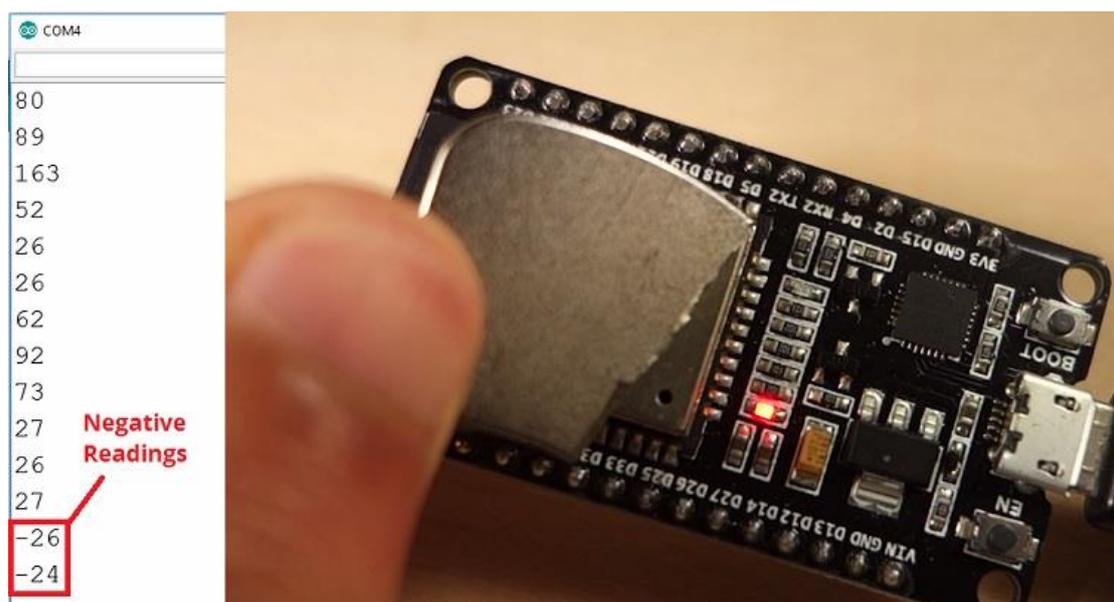


Demonstration

Once the upload is finished, open the Serial Monitor at a baud rate of 9600. Approximate a magnet to the ESP32 hall sensor and see the values increasing...



Or decreasing depending on the magnet pole that is facing the sensor:



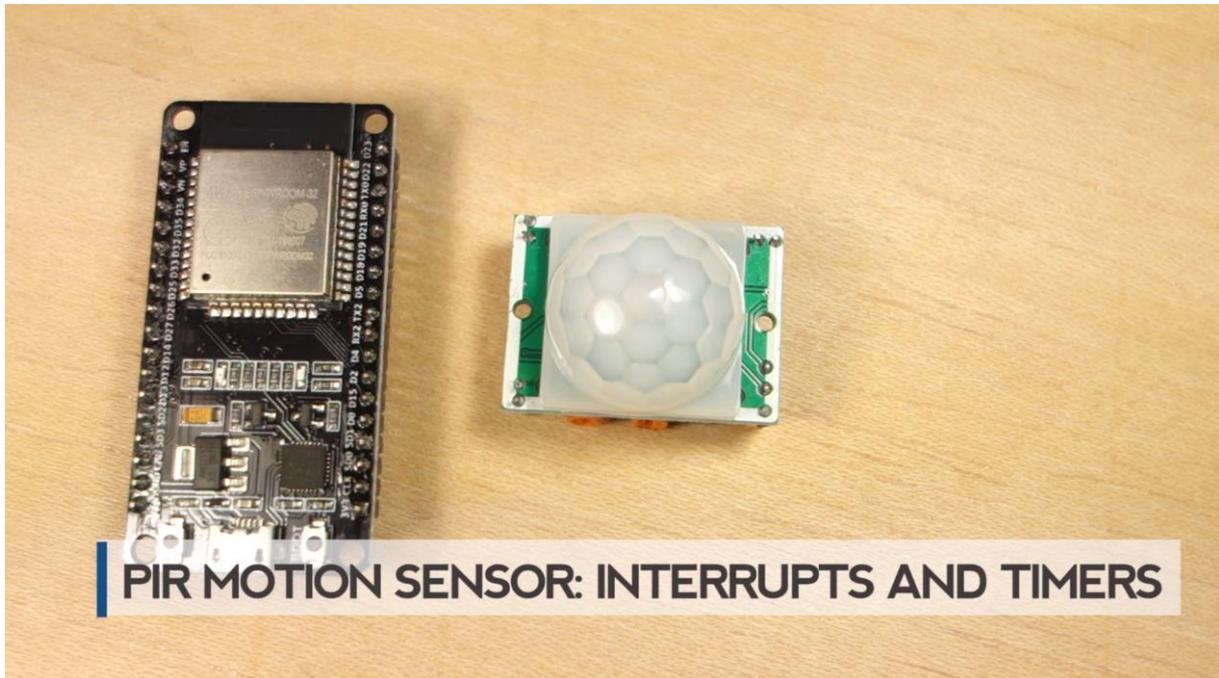
The closer the magnet is to the sensor, the greater the absolute values are.

Wrapping Up

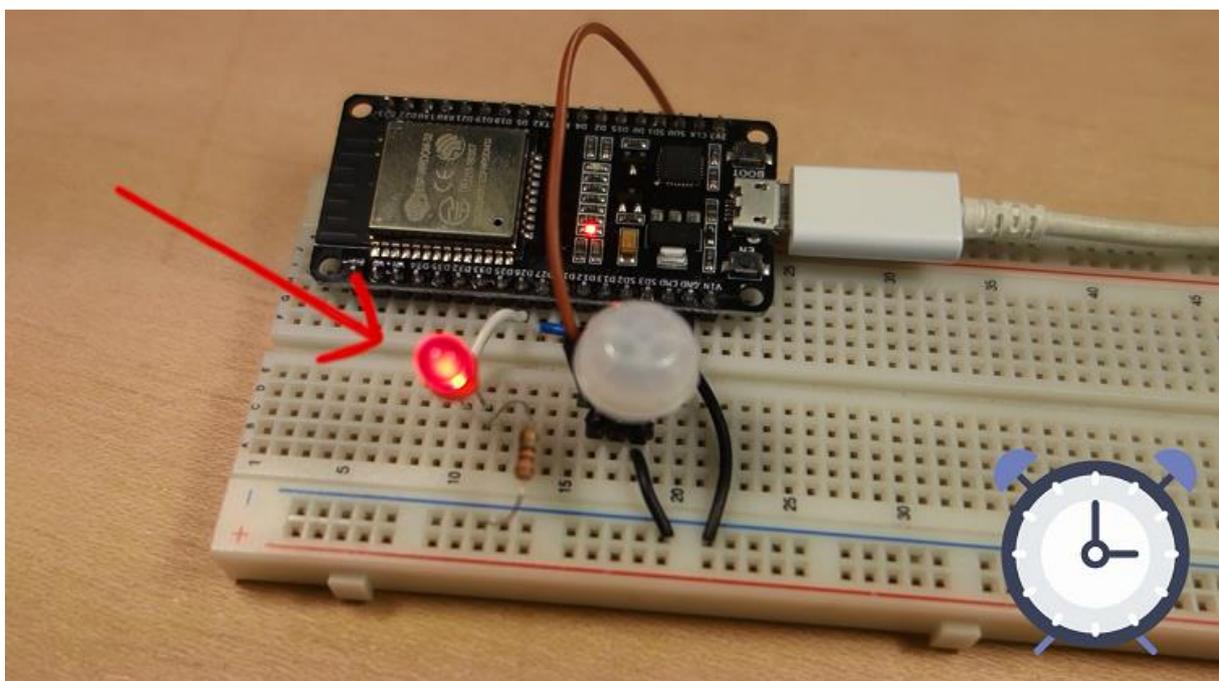
To wrap up:

- The ESP32 features a built-in hall effect sensor;
- The hall effect sensor can detect magnetic field changes in its surroundings;
- The measurements from the sensor can increase or become negative depending on the magnet pole facing the sensor.

Unit 6 - 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. When the timer is over, the LED is automatically turned off.



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

Introducing Interrupts

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

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

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

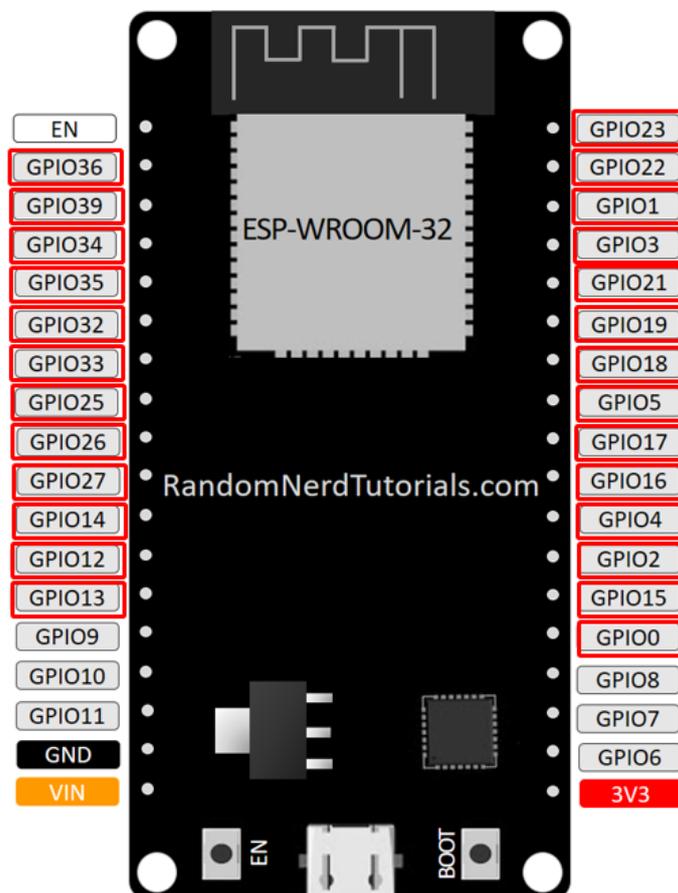
```
attachInterrupt(digitalPinToInterrupt(GPIO), funcion, mode);
```

GPIO Interrupt

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

```
digitalPinToInterrupt(27)
```

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



Function to be triggered

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

Mode

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

- **LOW:** to trigger the interrupt whenever the pin is LOW;
- **HIGH:** to trigger the interrupt whenever the pin is HIGH;
- **CHANGE:** to trigger the interrupt whenever the pin changes value - for example from HIGH to LOW or LOW to HIGH;
- **FALLING:** for when the pin goes from HIGH to LOW.
- **RISING:** to trigger when the pin goes from LOW to HIGH;

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

Introducing Timers



For this project we'll also be introducing 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'll use a timer.

The `delay()` function

Until now, we've used the `delay()` function that is pretty straightforward. It accepts a single `int` number as an argument. This number represents the time in milliseconds the program has to wait until moving on to the next line of code.

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

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

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

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

The millis() function

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

`millis()`

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

Blinking an LED with millis()

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

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Blink_Without_Delay/Blink_Without_Delay.ino

```
// 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;

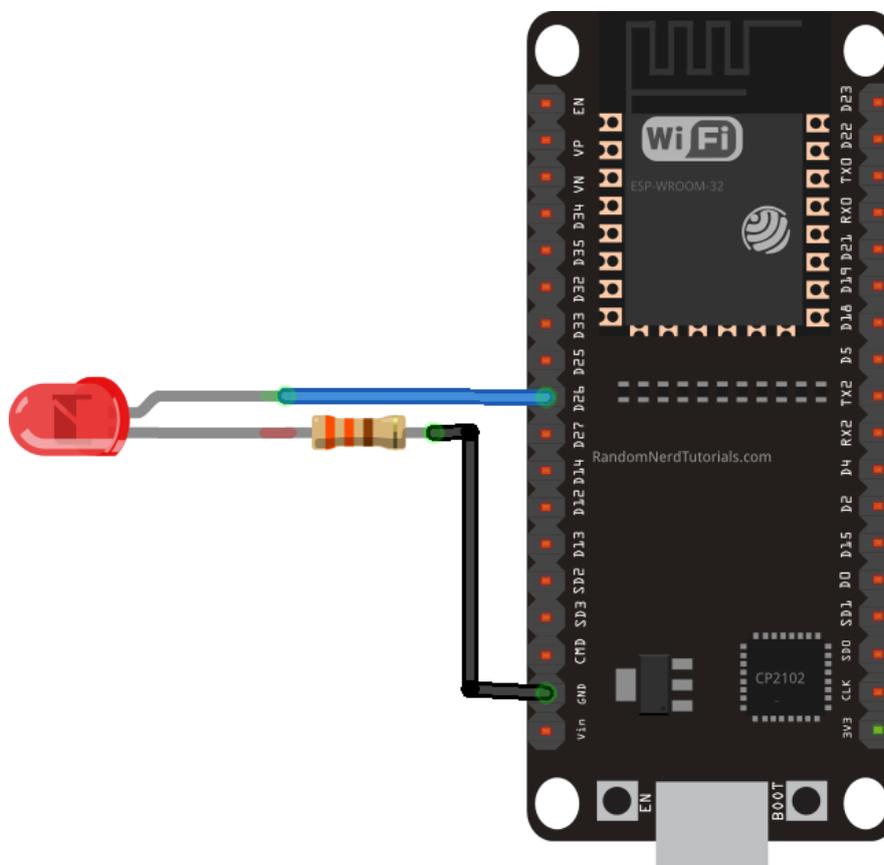
    // if the LED is off turn it on and vice-versa:
    if (ledState == LOW) {
      ledState = HIGH;
    } else {
      ledState = LOW;
    }
    // set the LED with the ledState of the variable:
    digitalWrite(ledPin, ledState);
  }
}
```

Let's take a closer look at this blink sketch that works without a `delay()` function (it uses the `millis()` function instead). Basically, this code subtracts the previous recorded time (`previousMillis`) from the current time (`currentMillis`). If the remainder is greater than the interval (in this case, 1000 milliseconds), the program updates the `previousMillis` variable to the current time, and either turns the LED on or off.

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 one second.

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



(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.)

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [330 Ohm resistor](#)
- [Jumper wires](#)
- [Breadboard](#)

ESP32 with PIR Motion Sensor

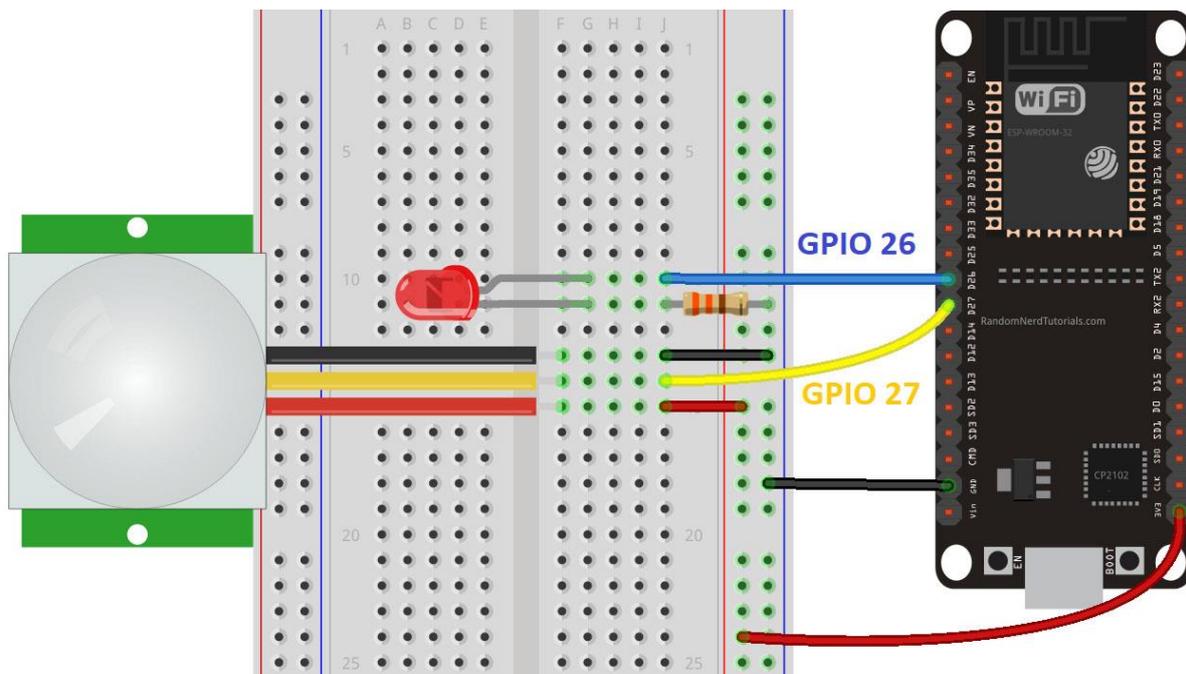
After understanding these two new concepts: **interrupts** and **timers**, let's continue with the project.

Schematic

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](#)
- [330 Ohm resistor](#)
- [Jumper wires](#)
- [Breadboard](#)

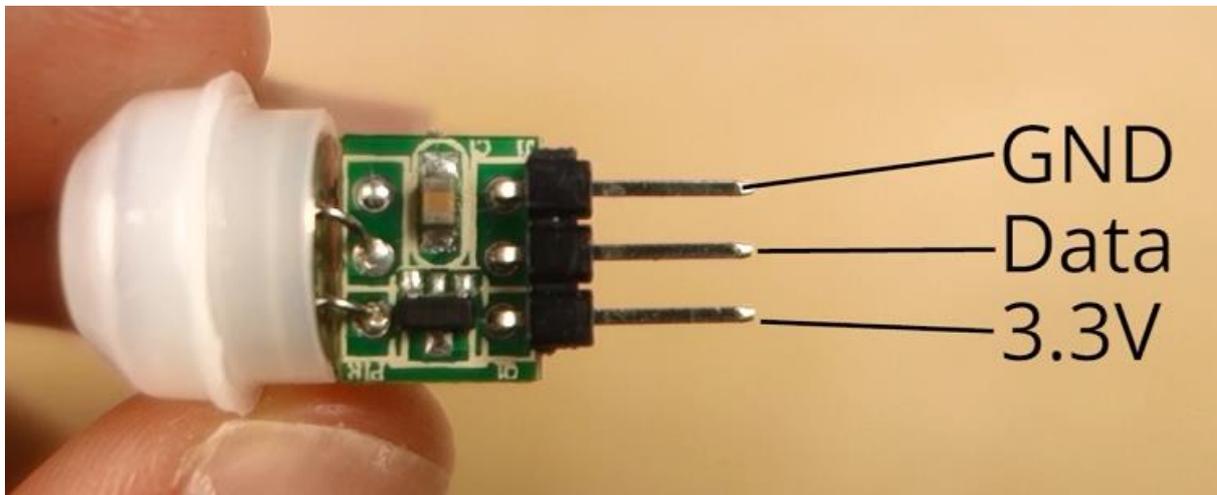
The circuit is easy to assemble, we'll be using an LED with a resistor. The LED is connected to GPIO 26. We'll be using the Mini AM312 PIR Motion Sensor that operates at 3.3V. It will be connected to GPIO 27. Simply follow the next schematic diagram.



(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.)

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 AM312, it operates at 5V. You can either modify it to operate at 3.3V or simply power it using the ESP32 Vin pin.

The following figure shows the AM312 PIR motion sensor pinout.



Code

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

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

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/PIR_Interrupts_Timers/PIR_Interrupts_Timers.ino

```
#define timeSeconds 10

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

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

// Checks if motion was detected, sets LED HIGH and starts a timer
void IRAM_ATTR detectsMovement() {
  Serial.println("MOTION DETECTED!!!");
  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();
  // 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;
  }
}
}

```

How the Code Works

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 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.

setup()

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

```

Serial.begin(115200);

```

Set the PIR Motion sensor as an INPUT PULLUP.

```

pinMode(motionSensor, INPUT_PULLUP);

```

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

```

attachInterrupt(digitalPinToInterrupt(motionSensor), detectsMovement, RISING);

```

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

The LED is an OUTPUT whose state starts at LOW.

```

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

```

loop()

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

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

Nothing else is done in the `loop()`. But, when motion is detected, the `detectsMovement()` function is called because we've set an interrupt previously on the `setup()`.

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

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

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

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

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

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

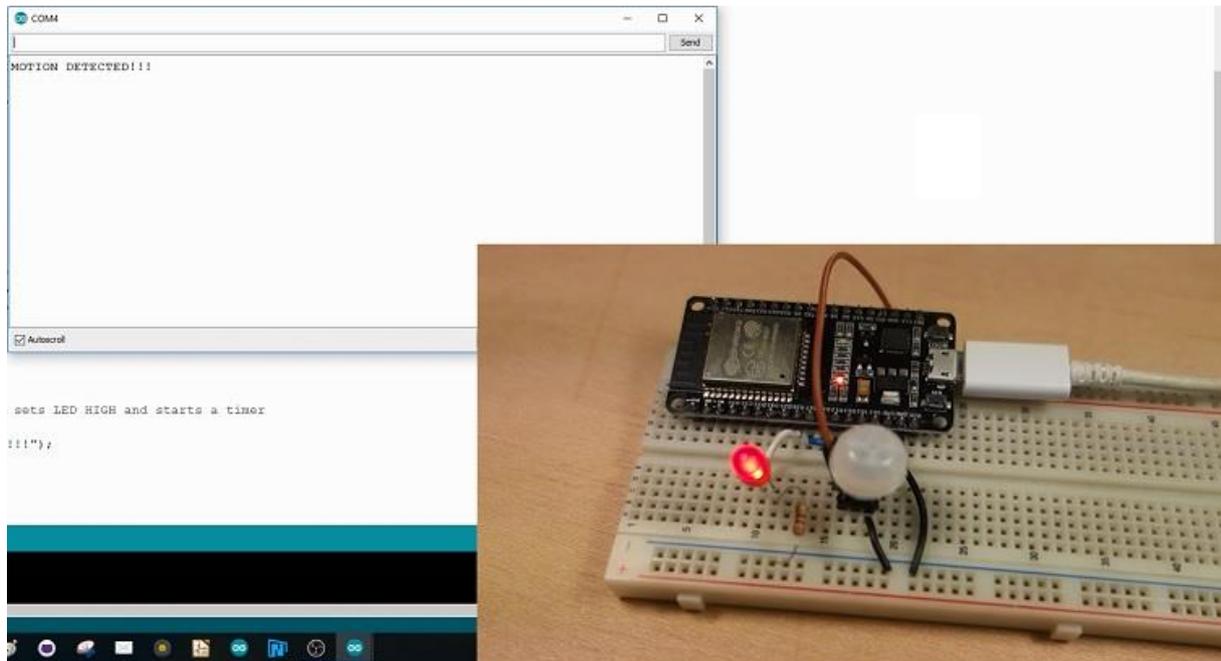
The "Motion stopped..." message will be printed in the Serial Monitor, the LED is turned off, and the `startTimer` variable is set to false.

Demonstration

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



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



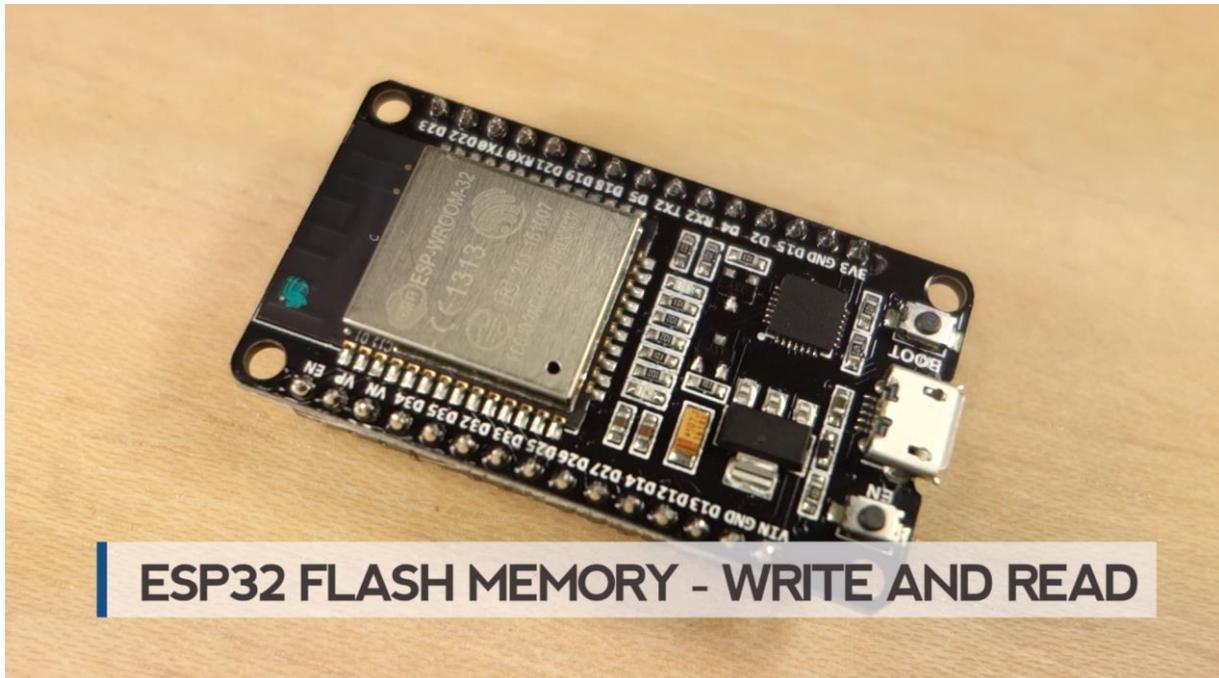
Wrapping Up

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

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

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

Unit 7 - ESP32 Flash Memory – Store Permanent Data (Write and Read)



In this Unit we're going to show you how to store and read values from the ESP32 flash memory. As an example we'll show you how to save the last GPIO state.

Flash Memory

The data saved in the flash memory remains there even when the ESP32 resets or when power is removed. The flash memory is very similar to the EEPROM. Both are non-volatile memories.

Saving data in the flash memory is especially useful to:

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

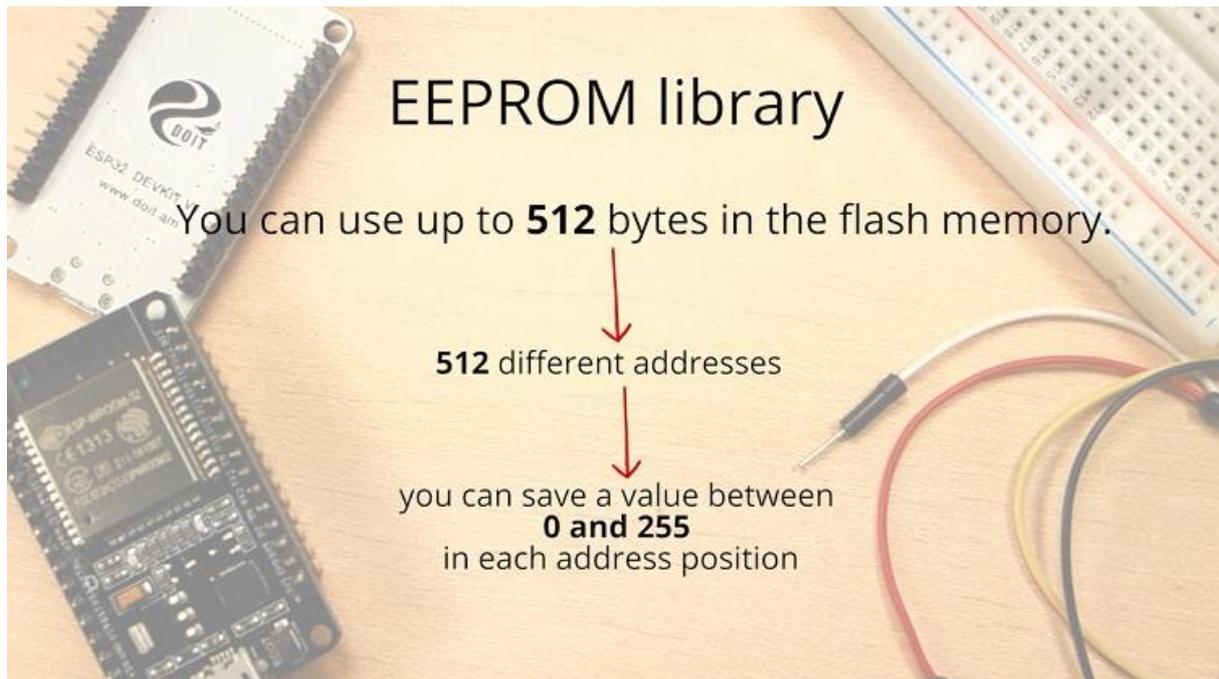
One limitation with flash memory is the number of times you can write data to it. Data can be read from flash as many times as you want, but most devices are designed for about 100,000 to 1,000,000 write operations. To learn more about this, we recommend [reading this article](#).

EEPROM Library

To read and write from the ESP32 flash memory using Arduino IDE, we'll be using the EEPROM library. Using this library with the ESP32 is very similar to using it with the Arduino. So, if you've used the Arduino EEPROM before, this is not much different.

So, we also recommend taking a look at our article about [Arduino EEPROM](#).

With the ESP32 and the EEPROM library you can use up to 512 bytes in the flash memory. This means you have 512 different addresses, and you can save a value between 0 and 255 in each address position.



Write

To write data to the flash memory, you use the `EEPROM.write()` function that accepts as arguments the location or address where you want to save the data, and the value (a byte variable) you want to save:

```
EEPROM.write(address, value);
```

For example, to write 9 on address 0, you'll have:

```
EEPROM.write(0, 9);
```

Followed by:

```
EEPROM.commit();
```

For the changes to be saved.

Read

To read a byte from the flash memory, you use the `EEPROM.read()` function. This function takes the address of the byte you want to read as an argument.

```
EEPROM.read(address);
```

For example, to read the byte stored previously in address 0, use:

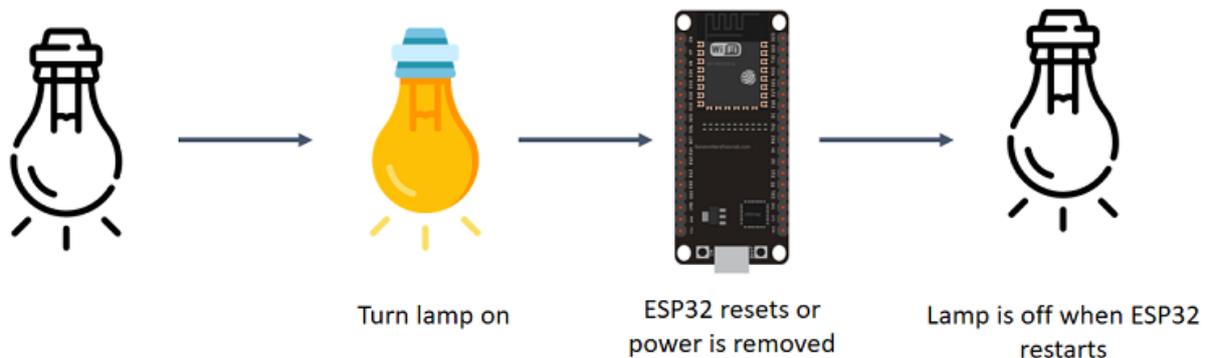
```
EEPROM.read(0);
```

This would return **9**, which is the value we stored in address 0.

Remember Last GPIO State

To show you how to save data in the ESP32 flash memory, we'll save the last state of an output, in this case an LED. For example, imagine the following scenario:

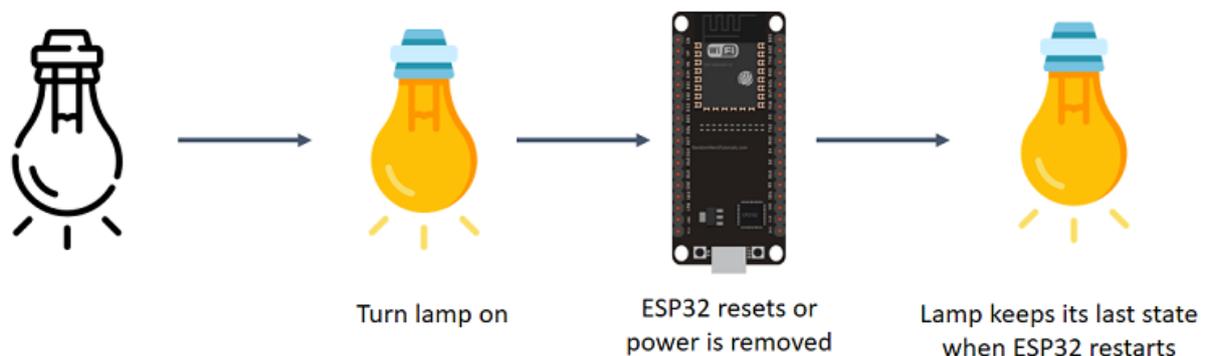
- 1) You're controlling a lamp with the ESP32
- 2) You set your lamp to turn on
- 3) The ESP32 suddenly loses power
- 4) When the power comes back on, the lamp stays off – because it doesn't keep its last state



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 lamp's state in the flash memory. Then, you just need to add a condition at the beginning of your sketch to check the last lamp state, and turn the lamp on or off accordingly.

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

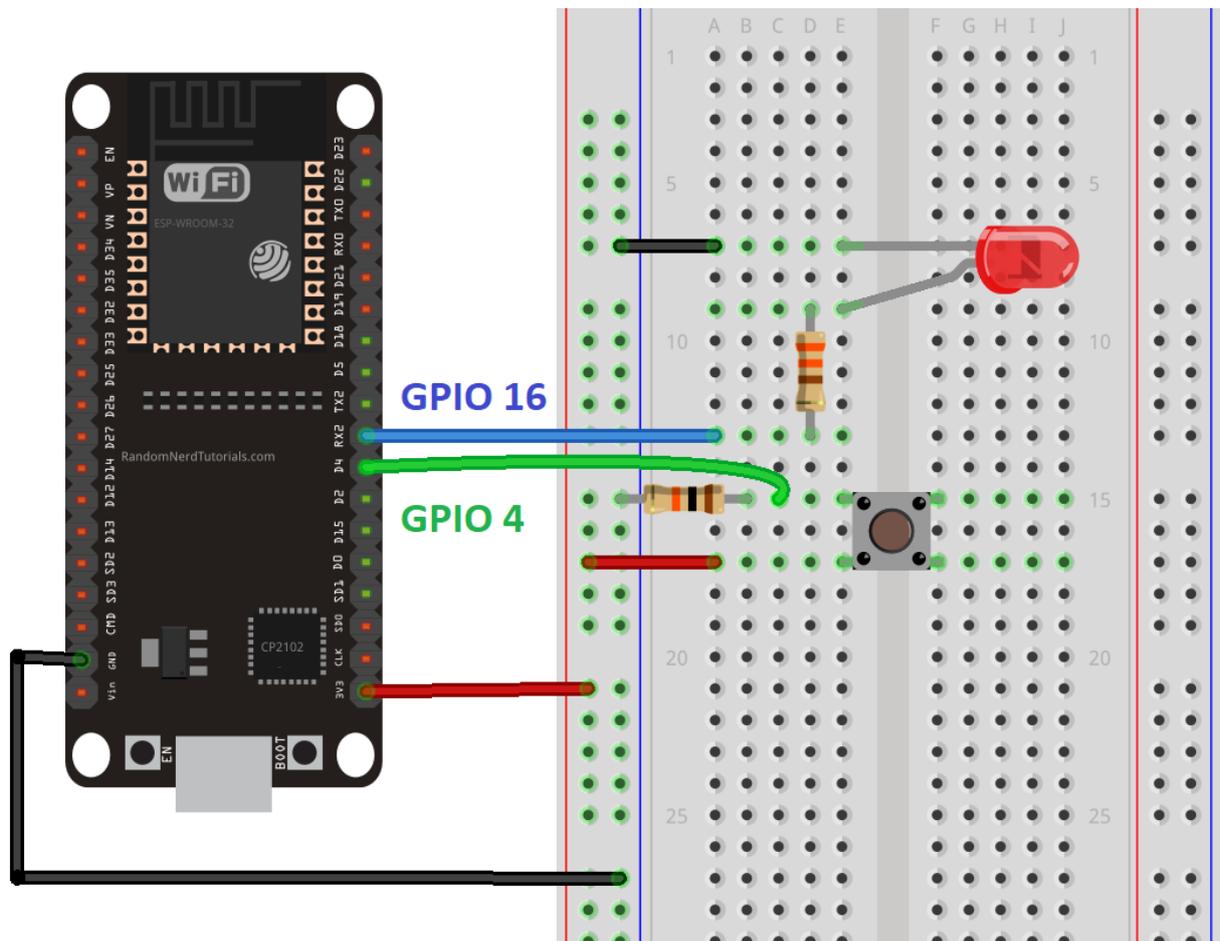


Schematic

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

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

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

Code

Copy the following code to the Arduino IDE and upload it to your ESP32. Make sure you have the right board and COM port selected.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/EEPROM_Last_LED_State/EEPROM_Last_LED_State.ino

```

// include library to read and write from flash memory
#include <EEPROM.h>

// define the number of bytes you want to access
#define EEPROM_SIZE 1

// constants won't change. They're used here to set pin numbers:
const int buttonPin = 4;    // the number of the pushbutton pin
const int ledPin = 16;     // the number of the LED pin

// Variables will change:
int ledState = HIGH;       // the current state of the output pin
int buttonState;          // the current reading from the input pin
int lastButtonState = LOW; // the previous reading from the input pin

// the following variables are unsigned longs because the time, measured in
// milliseconds, will quickly become a bigger number than can be stored in
// an int.
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);

    // initialize EEPROM with predefined size
    EEPROM.begin(EEPROM_SIZE);

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

    // read the last LED state from flash memory
    ledState = EEPROM.read(0);
    // set the LED to the last stored state
    digitalWrite(ledPin, ledState);
}

void loop() {
    // read the state of the switch into a local variable:
    int reading = digitalRead(buttonPin);

    // check to see if you just pressed the button
    // (i.e. the input went from LOW to HIGH), and you've waited long enough
    // since the last press to ignore any noise:

    // If the switch changed, due to noise or pressing:
    if (reading != lastButtonState) {
        // reset the debouncing timer
        lastDebounceTime = millis();
    }

    if ((millis() - lastDebounceTime) > debounceDelay) {
        // whatever the reading is at, it's been there for longer than the debounce
        // delay, so take it as the actual current state:

        // if the button state has changed:
        if (reading != buttonState) {
            buttonState = reading;

            // only toggle the LED if the new button state is HIGH
            if (buttonState == HIGH) {
                ledState = !ledState;
            }
        }
    }
    // save the reading. Next time through the loop, it'll be the lastButtonState:
    lastButtonState = reading;
}

```

```

// if the ledState variable is different from the current LED state
if (digitalRead(ledPin) != ledState) {
  Serial.println("State changed");
  // change the LED state
  digitalWrite(ledPin, ledState);
  // save the LED state in flash memory
  EEPROM.write(0, ledState);
  EEPROM.commit();
  Serial.println("State saved in flash memory");
}
}

```

How the Code Works

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. Let's see what you have to do to make the ESP32 remember the last state of a GPIO.

First, you need to include the EEPROM library.

```
#include <EEPROM.h>
```

Then, you define the EEPROM size. This is the number of bytes you'll want to access in the flash memory. In this case, we'll just save the LED state, so the EEPROM size is set to 1.

```
#define EEPROM_SIZE 1
```

We also define other variables that are required to make this sketch work.

```

// constants won't change. They're used here to set pin numbers:
const int buttonPin = 4;    // the number of the pushbutton pin
const int ledPin = 16;     // the number of the LED pin

// Variables will change:
int ledState = HIGH;       // the current state of the output pin
int buttonState;          // the current reading from the input pin
int lastButtonState = LOW; // the previous reading from the input pin

// the following variables are unsigned longs because the time, measured in
// milliseconds, will quickly become a bigger number than can be stored in
// an int.
unsigned long lastDebounceTime = 0; // the last time the output pin was toggled
unsigned long debounceDelay = 50;  // the debounce time; increase if the output flickers

```

setup()

In the `setup()` you initialize the EEPROM with the predefined size.

```
EEPROM.begin(EEPROM_SIZE);
```

To make sure your code initializes with the latest LED state, in the `setup()`, you should read the last LED state from the flash memory. It is stored on address zero.

```
ledState = EEPROM.read(0);
```

Then, you just need to turn the LED ON or OFF accordingly to the value read from the flash memory.

```
digitalWrite(ledPin, ledState);
```

loop()

The following part of the `loop()` checks if the pushbutton was pressed and changes the `ledState` variable every time we press the pushbutton. To make sure we don't get false positives we use a timer. This snippet of code is based on the pushbutton debounce sketch example from the Arduino IDE.

```
// read the state of the switch into a local variable:
int reading = digitalRead(buttonPin);

// check to see if you just pressed the button
// (i.e. the input went from LOW to HIGH), and you've waited long enough
// since the last press to ignore any noise:

// If the switch changed, due to noise or pressing:
if (reading != lastButtonState) {
  // reset the debouncing timer
  lastDebounceTime = millis();
}

if ((millis() - lastDebounceTime) > debounceDelay) {
  // whatever the reading is at, it's been there for longer than the debounce
  // delay, so take it as the actual current state:
  // if the button state has changed:
  if (reading != buttonState) {
    buttonState = reading;

    // only toggle the LED if the new button state is HIGH
    if (buttonState == HIGH) {
      ledState = !ledState;
    }
  }
}
}
```

You simply need to save the LED state in the flash memory everytime the LED state changes.

We check if the state of the GPIO is different from the `ledState` variable.

```
if (digitalRead(ledPin) != ledState) {
```

If it is, we'll change the LED state using the `digitalWrite()` function.

```
digitalWrite(ledPin, ledState);
```

And then we save the current state in the flash memory. For that, we use `EEPROM.write()`, and pass as arguments the address position, in this case 0, and the value to be saved, in this case the `ledState` variable.

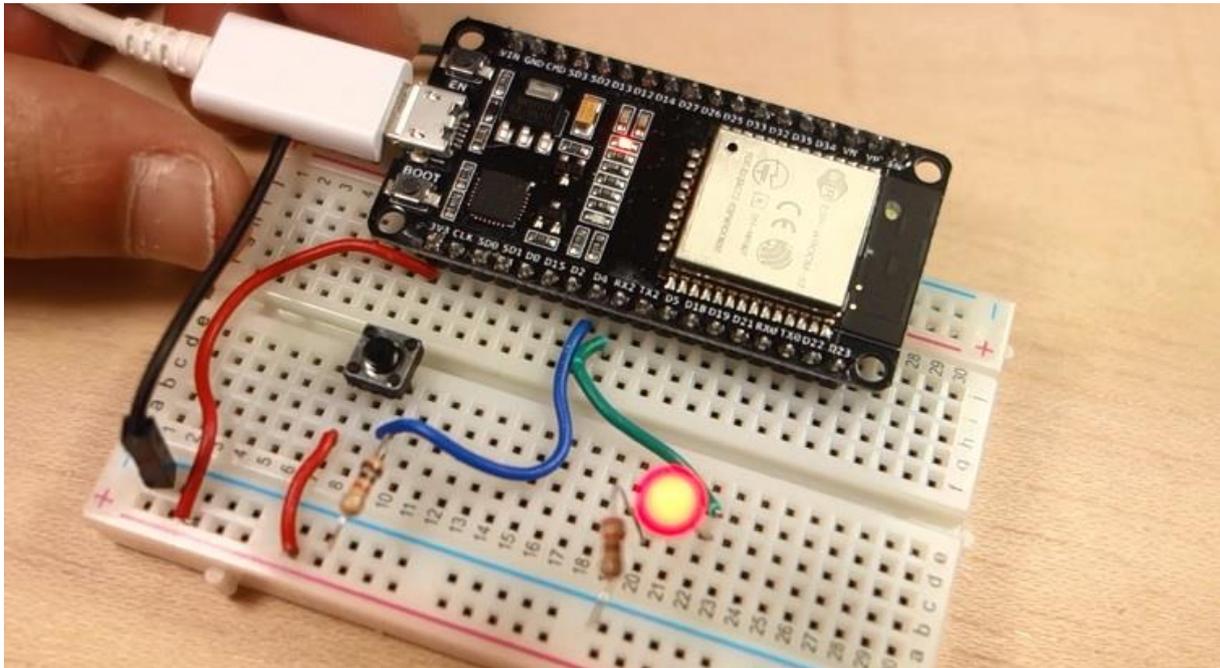
```
EEPROM.write(0, ledState);
```

Finally, we use the `EEPROM.commit()` for the changes to take effect.

```
EEPROM.commit();
```

Demonstration

After uploading the code to your ESP32, press the pushbutton to turn the LED on and off. The ESP32 should keep the last LED state after resetting or removing power.

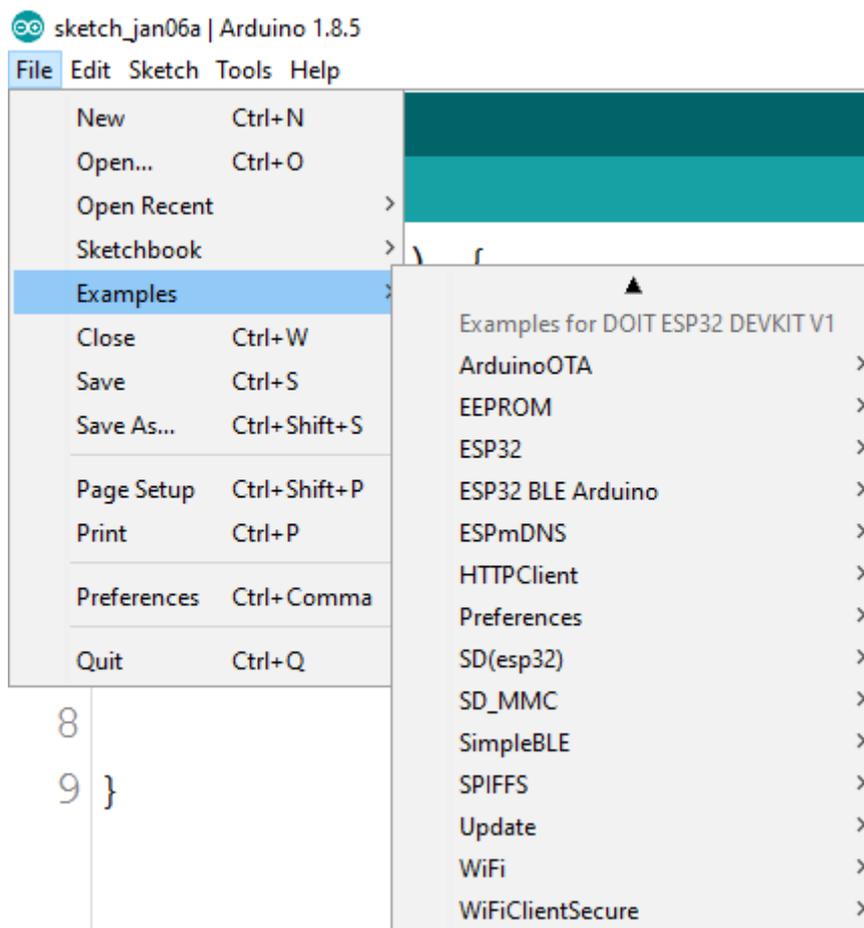


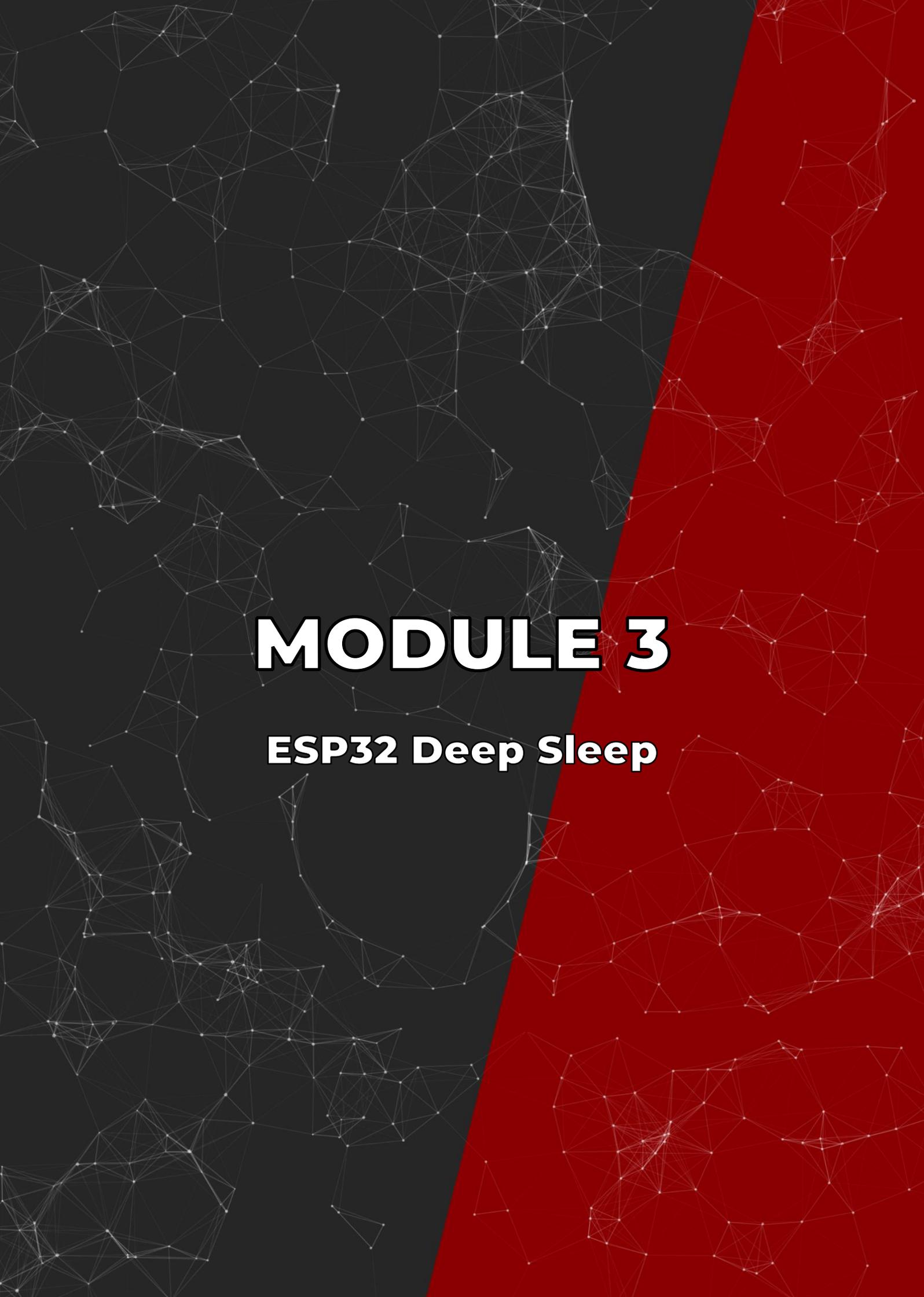
Wrapping Up

In summary, in this unit you've learned how to save data in the ESP32 flash memory using the EEPROM library. Data saved on the flash memory remains there even after resetting the ESP32 or removing power.

Unit 8 - Other ESP32 Sketch Examples

Until now, we've only covered some ESP32 sketch examples that come with the ESP32 add-on for the Arduino IDE. We'll explore more examples in the upcoming Modules, but feel free to open the other examples and try them on your ESP32.





MODULE 3

ESP32 Deep Sleep

Unit 1 - ESP32 Deep Sleep Mode



In this section you're going to learn what is deep sleep, 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 about the deep sleep mode. For ESP32 deep sleep examples, go to the next 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 on the following table from the ESP32 Espressif datasheet.

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

The [ESP32 Espressif datasheet](#) also provides a table comparing the power consumption of the different power modes.

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

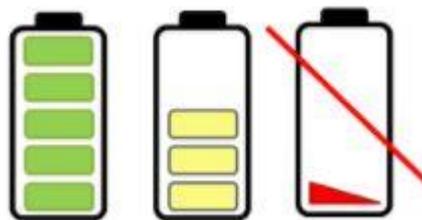
And here's also Table 10 to compare the power consumption in active mode:

Table 10: RF Power-Consumption Specifications

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

Why Deep Sleep Mode?

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



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

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

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

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

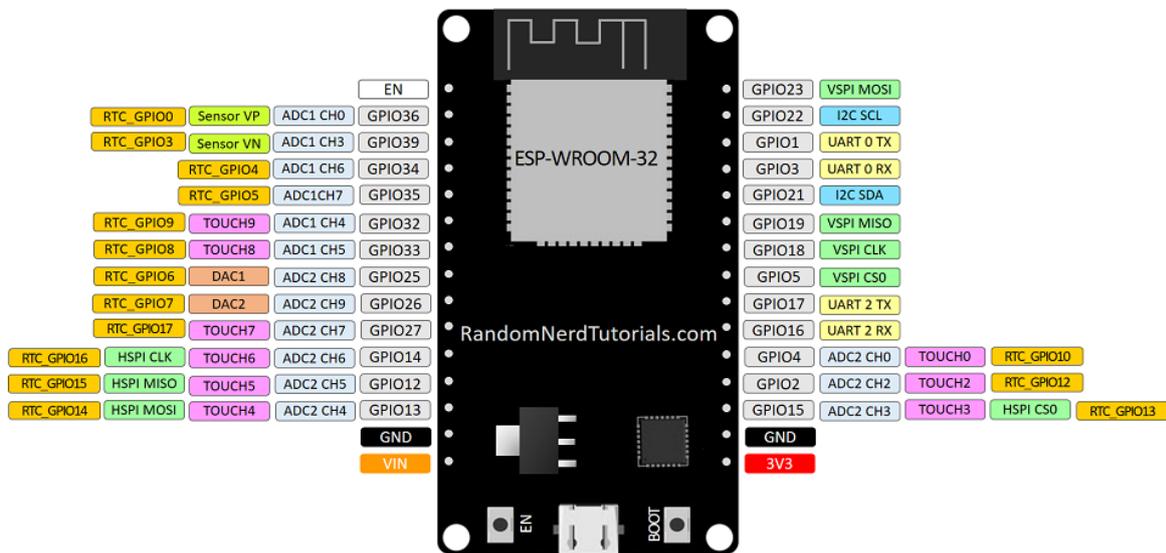
This mode of operation is useful if you need to wake up the main CPU by an external event, timer, or both, while maintaining minimal power consumption.

RTC_GPIO Pins

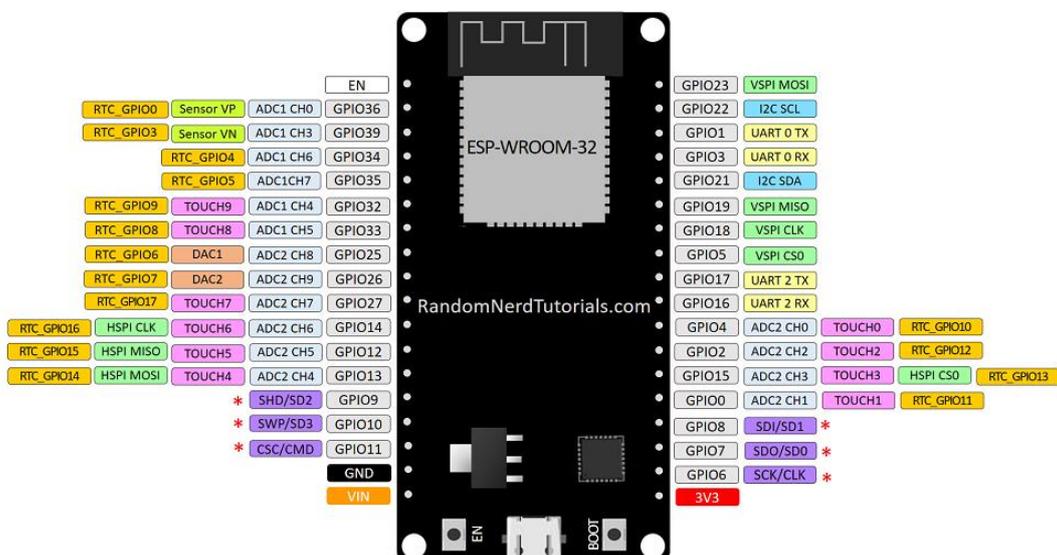
During deep sleep, some of the ESP32 pins can be used by the ULP co-processor, namely the RTC_GPIO pins, and the Touch Pins. The ESP32 datasheet provides a table identifying the RTC_GPIO pins. You can find that table [here](#) at page 7.

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

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 using 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-ups;
- You can use the touch pins;
- You can use the ULP co-processor to wake-up.

Note: for now, we'll only cover the first three wake-up options, because at the moment we couldn't find a suitable and straightforward example to use the ULP co-processor 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 configure what will wake-up the ESP32. You can use one or combine more than one wake-up source.
- 2) You can decide what peripherals to shut down or keep on during deep sleep. However, by default, the ESP32 automatically powers down the peripherals that are not needed with the wake-up source you define.
- 3) Finally, you use the `esp_deep_sleep_start()` function to put your ESP32 into deep sleep mode.

Note: the following units in this module will show you how to use the different wake-up sources.

Additional Resources



Alongside with 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](#)

Unit 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 especially useful if you are running projects that require time stamping or daily tasks, while maintaining low power consumption.

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

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/DeepSleep/TimerWakeUp/TimerWakeUp.ino>

```

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

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

This code is under Public Domain License.

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

#define uS_TO_S_FACTOR 1000000 /* Conversion factor for micro 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.

The first comment describes what is powered off during deep sleep with timer wake-up.

```

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

```

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

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

Define the Sleep Time

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

```
#define uS_TO_S_FACTOR 1000000 /* Conversion factor for micro 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 reason by which the 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;  
  }  
}
```

setup()

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

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

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

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

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

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

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

```
esp_sleep_enable_timer_wakeup(time_in_us);
```

This function accepts as argument the time to sleep in microseconds as we've seen previously. In our case, we have the following:

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

Then, after all the tasks are performed, the ESP32 goes to sleep by calling the following function:

```
esp_deep_sleep_start();
```

loop()

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

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

Testing the Example

Open the Serial Monitor at a baud rate of 115200.

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

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

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

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

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

Wrapping Up

In summary, in this Unit we've shown you how to use the timer wake-up source to wake-up the ESP.

- 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 to perform periodic tasks with the ESP32, like daily tasks, without draining much power.

Unit 3 – Deep Sleep 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 touchpin is simple. In the Arduino IDE, you need to use the following function:

```
esp_sleep_enable_touchpad_wakeup();
```

Code

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

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/DeepSleep/TouchWakeUp/TouchWakeUp.ino>

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

This code is under Public Domain License.

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

```

*/

#define Threshold 40 /* Greater the value, more the sensitivity */

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 1 : Serial.println("Wakeup caused by external signal using
RTC_IO"); break;
        case 2 : Serial.println("Wakeup caused by external signal using
RTC_CNTL"); break;
        case 3 : Serial.println("Wakeup caused by timer"); break;
        case 4 : Serial.println("Wakeup caused by touchpad"); break;
        case 5 : Serial.println("Wakeup caused by ULP program"); break;
        default : Serial.println("Wakeup was not caused by deep sleep"); break;
    }
}

/*
Method to print the touchpad by which ESP32
has been awoken from sleep
*/
void print_wakeup_touchpad(){
    touch_pad_t pin;

    touchPin = esp_sleep_get_touchpad_wakeup_status();

    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;
    }
}

void callback(){
    //placeholder callback function
}

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();

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

//Configure Touchpad as wakeup source
esp_sleep_enable_touchpad_wakeup();

//Go to sleep now
Serial.println("Going to sleep now");
delay(1000);
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 setting a threshold value for the touch pins.

```
#define Threshold 40 /* Greater the value, more the sensitivity */
```

When you touch a touch pin, the values read will drop. The threshold value set here, will tell the ESP32 that when the value read is below 40, it should wake up. You can adjust that value accordingly to the desired sensitivity.

Attach Interrupts

You need to attach interrupts of the touch sensors to a call back function so that the touch sensor works as a wake-up source. For example, take a look at the following line:

```

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

```

Here we are setting that when the touch pin 3, that corresponds to GPIO 15, reads a value below the threshold set in the Threshold variable, it will wake up the ESP32

The `callback()` function will only be executed if the ESP32 is awake. If the ESP32 is asleep and you touch T3, the ESP will wake up – the `callback()` function won’t be executed if you just press and release the touch pin;

If the ESP32 is awake and you touch T3, the callback function will be executed. So, if you want to execute the `callback()` function when you wake up the ESP32, you need to hold the touch on that pin for a while, until the function is executed. In this case the `callback()` function is empty.

```

void callback(){
  //placeholder callback function
}

```

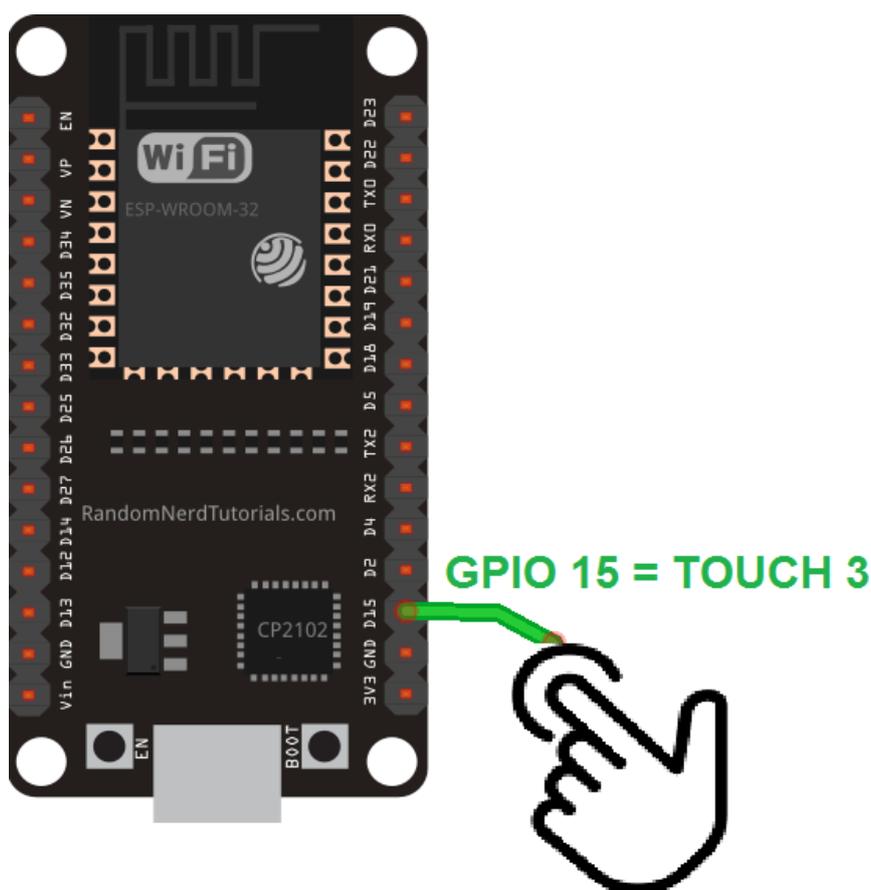
If you want to wake up the ESP32 using different touch pins, you just have to attach interrupts to those pins.

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

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

Schematic

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



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

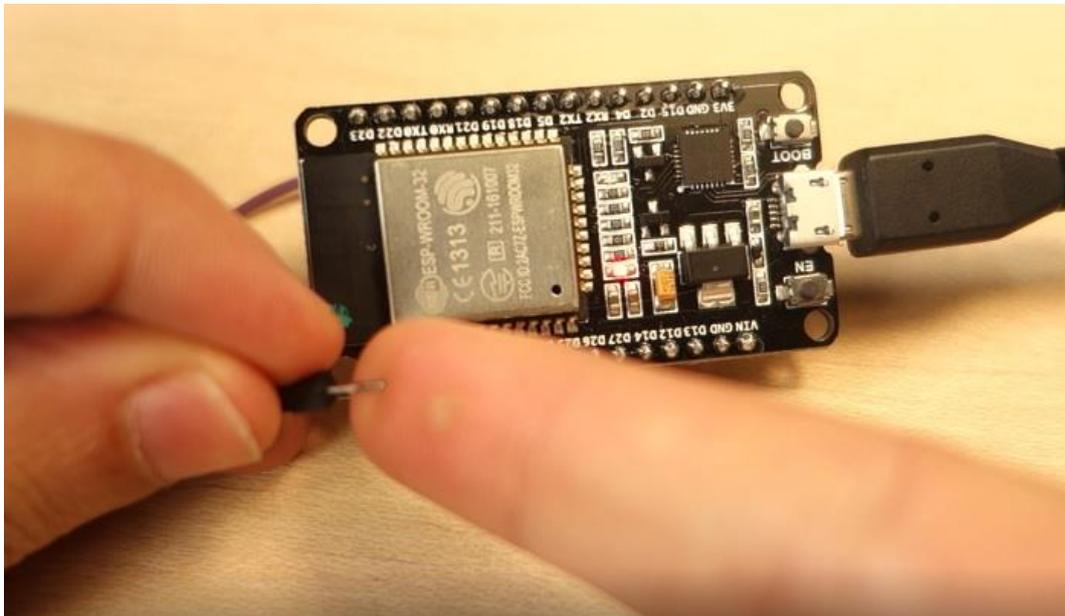
Testing the Example

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



The ESP32 goes into deep sleep mode.

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



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

```
COM7
ets Jun  8 2016 00:22:57

rst:0x5 (DEEPSLEEP_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:812
load:0x40078000,len:0
load:0x40078000,len:11392
entry 0x40078a9c
Boot number: 4
Wakeup caused by touchpad
Touch detected on GPIO 15
Going to sleep now
```

Wrapping Up

This unit showed you how to wake up the ESP32 using the touch pins. In summary:

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

Unit 4 - Deep Sleep External Wake Up



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

External Wake Up (ext0)

This wake-up source allows you to use a pin to wake up the ESP32. The ext0 wake-up source option uses RTC GPIOs to wake-up. So, RTC peripherals will be kept on during deep sleep if this wake-up source is requested.

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

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

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

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

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

External Wake Up (ext1)

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

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

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

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

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

Important note: at the time of writing this unit, there's a typo in the deep sleep documentation. The documentation uses `esp_deep_sleep_enable_ext1_wakeup()`, but the function that works is the one we are showing here: `esp_sleep_enable_ext1_wakeup()`.

This function accepts two arguments:

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

Note: you can only use pins that are RTC GPIOs.

Code

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

Important note: at the time of writing this unit, there's a typo in the sketch example from the ESP32 library. The documentation uses `esp_deep_sleep_enable_ext1_wakeup()`, but the function that works is the one we are showing here: `esp_sleep_enable_ext1_wakeup()`. The code we provide here is fixed.

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/DeepSleep/ExternalWakeUp/ExternalWakeUp.ino>

```
/*
Deep Sleep with External Wake Up
=====
This code displays how to use deep sleep with
an external trigger as a wake up source and how
to store data in RTC memory to use it over reboots
This code is under Public Domain License.
Hardware Connections
=====
Push Button to GPIO 33 pulled down with a 10K Ohm
resistor
NOTE:
=====
Only RTC IO can be used as a source for external wake
source. They are pins: 0,2,4,12-15,25-27,32-39.
Author:
Pranav Cherukupalli <cherukupallip@gmail.com>
*/
```

```

#define BUTTON_PIN_BITMASK 0x20000000 // 2^33 in hex

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 doesnt need
peripherals to be powered on.
Note that using internal pullups/pulldowns also requires
RTC peripherals to be turned on.
*/
    esp_sleep_enable_ext0_wakeup(GPIO_NUM_33,1); //1 = High, 0 = Low

    //If you were to use ext1, you would use it like
    //esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIG
H);

    //Go to sleep now
    Serial.println("Going to sleep now");
    delay(1000);
    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.

This code is very similar with 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));
```

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

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

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

ext0

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

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

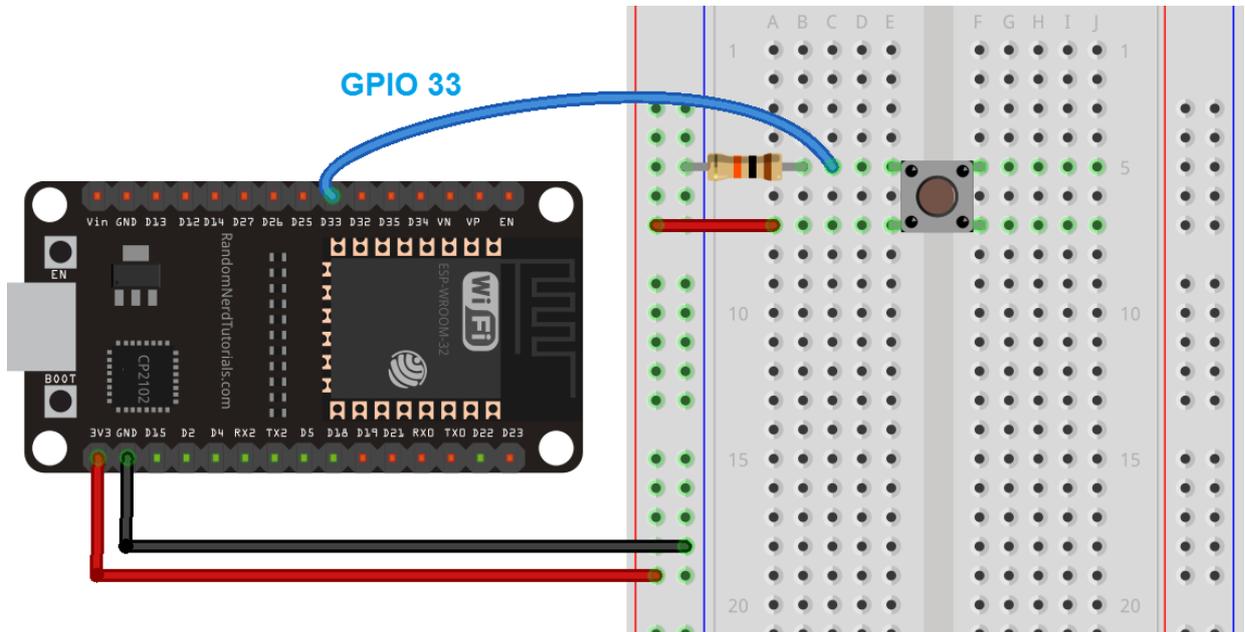
Instead of GPIO 33, you can use any other RTC GPIO pin.

Schematic

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [Pushbutton](#)
- [10k Ohm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)

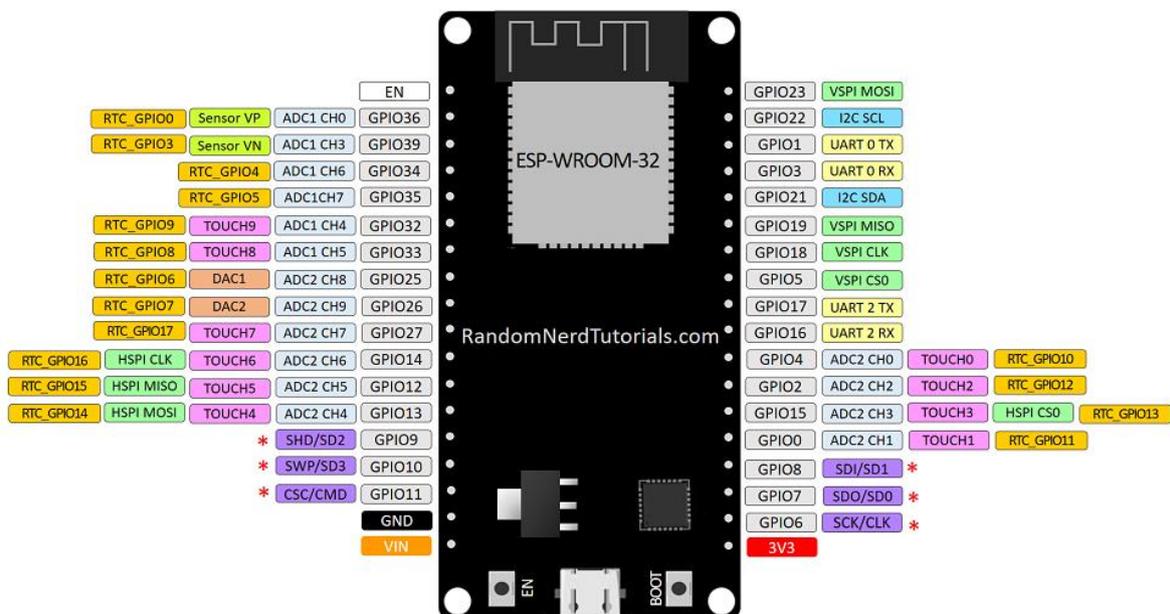
To test this example, wire a pushbutton to your ESP32 by following the next schematic diagram. The button is connected to GPIO 33 using a pull down 10K Ohm resistor.



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

Note that only RTC GPIOs can be used as a wake up source. These are highlighted in a rectangular orange box in the figure below. So, instead of GPIO 33, you could also use any RTC GPIO pins to connect your button.

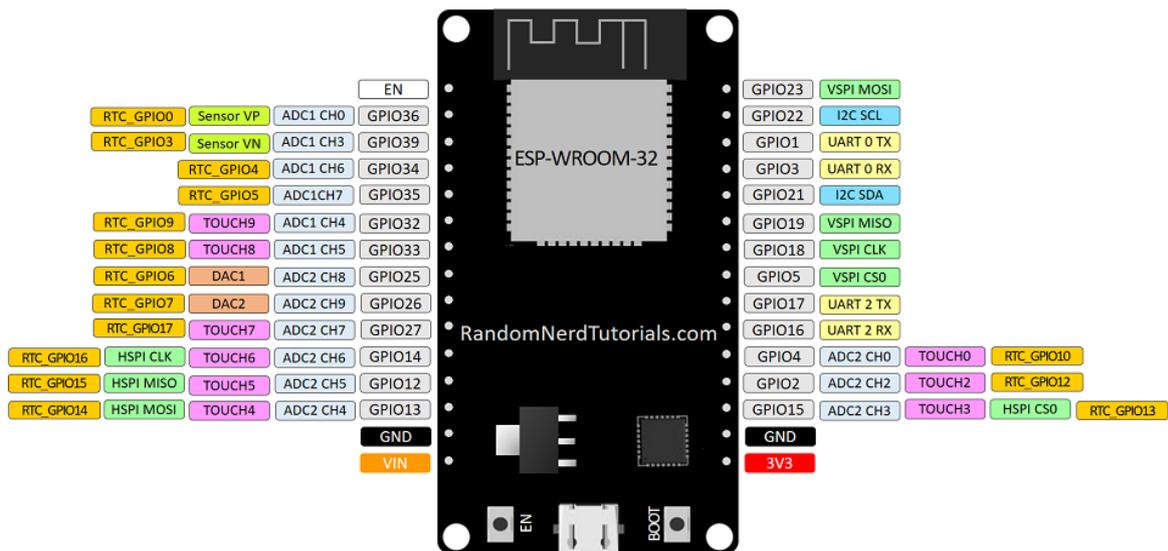
ESP32 DEVKIT V1 – DOIT version with 36 GPIOs



* 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.

ESP32 DEVKIT V1 – DOIT

version with 30 GPIOs

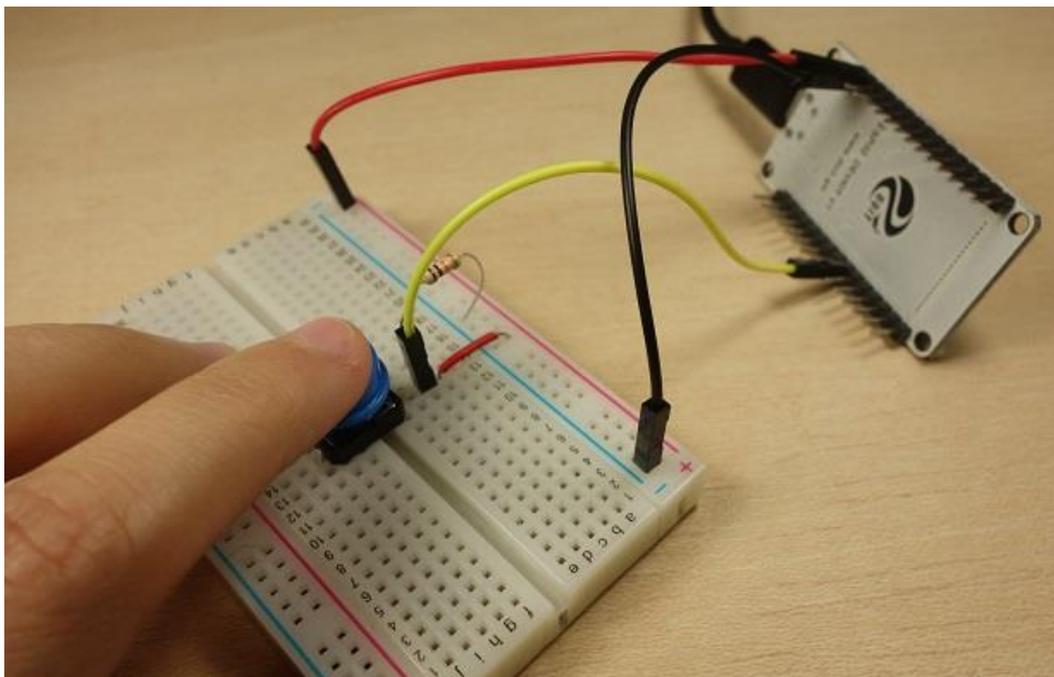


Testing the Example

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.

```
COM7
Boot number: 1
Wakeup was not caused by deep sleep
Going to sleep now
Boot number: 2
Wakeup caused by external signal using RTC_IO
Going to sleep now
Boot number: 3
Wakeup caused by external signal using RTC_IO
Going to sleep now
Boot number: 4
Wakeup caused by external signal using RTC_IO
Going to sleep now
Autoscroll Both NL & CR 115200 baud Clear output
```

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

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

ext1

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

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

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

Uncomment that function so that you have:

```
esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIGH);
```

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

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

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

This is only defining one pin as a wake-up source, GPIO33. You need to modify the bitmask to configure more GPIOs as a wake-up source.

Identifying the GPIO used as a wake up source

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

```
esp_sleep_get_ext1_wakeup_status();
```

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

```
GPIO = log(GPIO_NUMBER) / log(2);
```

External Wake Up – Multiple GPIOs

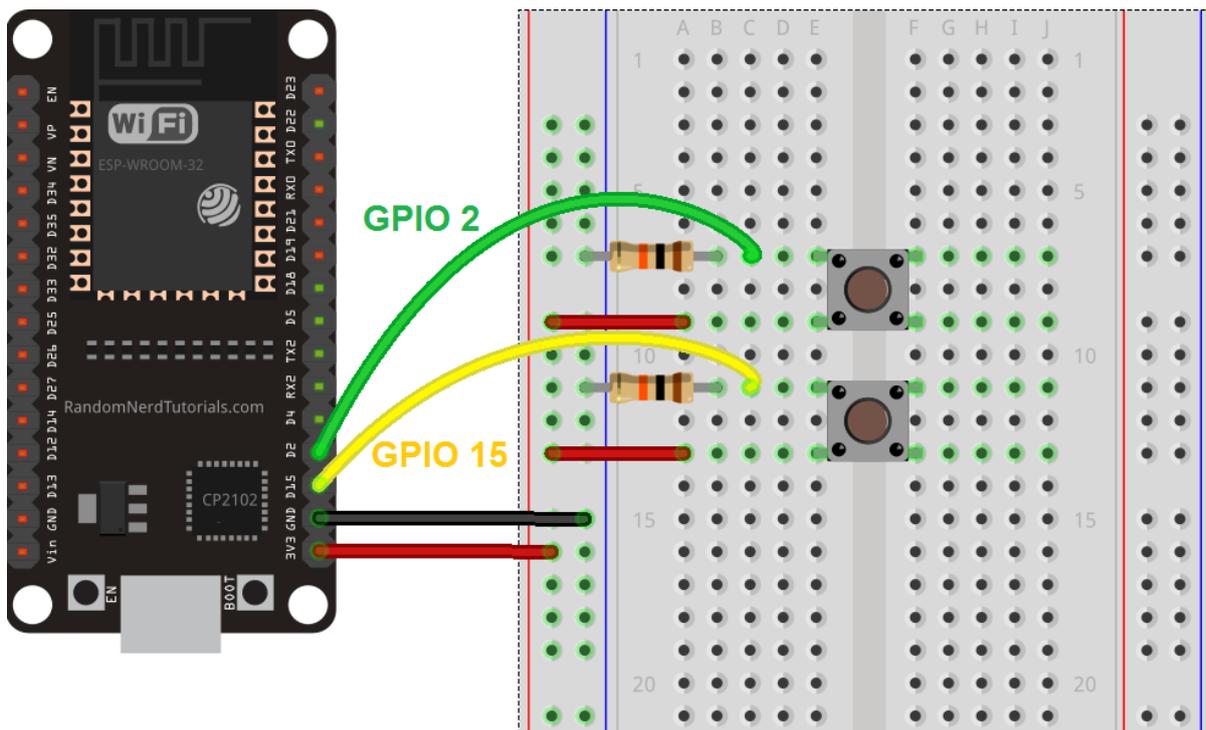
Now, you should be able to wake up the ESP32 using different buttons, and identify which button caused the wake up. In this example we'll use GPIO 2 and GPIO 15 as a wake up source.

Schematic

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) use the `esp_sleep_get_ext1_wakeup_status()` function to get the GPIO that triggered wake up.

The next sketch has all those changes implemented.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/DeepSleep/ExternalWakeUp_ext1_with_GPIO/ExternalWakeUp_ext1_with_GPIO.ino

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

This code is under Public Domain License.

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

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

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

#define BUTTON_PIN_BITMASK 0x8004 // GPIOs 2 and 15

RTC_DATA_ATTR int bootCount = 0;

/*
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 1 : Serial.println("Wakeup caused by external signal using
RTC_IO"); break;
        case 2 : Serial.println("Wakeup caused by external signal using
RTC_CNTL"); break;
        case 3 : Serial.println("Wakeup caused by timer"); break;
```

```

    case 4 : Serial.println("Wakeup caused by touchpad"); break;
    case 5 : Serial.println("Wakeup caused by ULP program"); break;
    default : Serial.println("Wakeup was not caused by deep sleep"); break;
  }
}

/*
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));
}

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();

  //Print the GPIO used to wake up
  print_GPIO_wake_up();

  /*
  First we configure the wake up source
  We set our ESP32 to wake up for an external trigger.
  There are two types for ESP32, ext0 and ext1 .
  ext0 uses RTC_IO to wakeup thus requires RTC peripherals
  to be on while ext1 uses RTC Controller so doesnt need
  peripherals to be powered on.
  Note that using internal pullups/pulldowns also requires
  RTC peripherals to be turned on.
  */
  //esp_deep_sleep_enable_ext0_wakeup(GPIO_NUM_15,1); //1 = High, 0 = Low

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

  //Go to sleep now
  Serial.println("Going to sleep now");
  delay(1000);
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}

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

```

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

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

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

```

void print_GPIO_wake_up(){
  int GPIO_reason = esp_sleep_get_ext1_wakeup_status();

```

```
Serial.print("GPIO that triggered the wake up: GPIO ");
Serial.println((log(GPIO_reason))/log(2), 0);
}
```

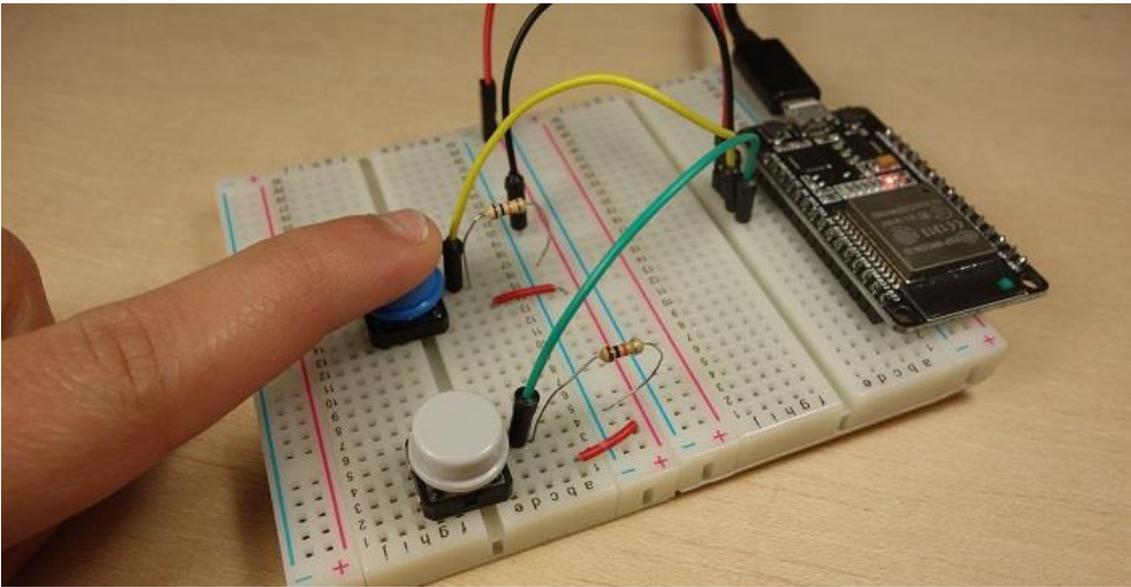
And finally, you enable ext1 as a wake-up source:

```
esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIGH);
```

Testing the Sketch

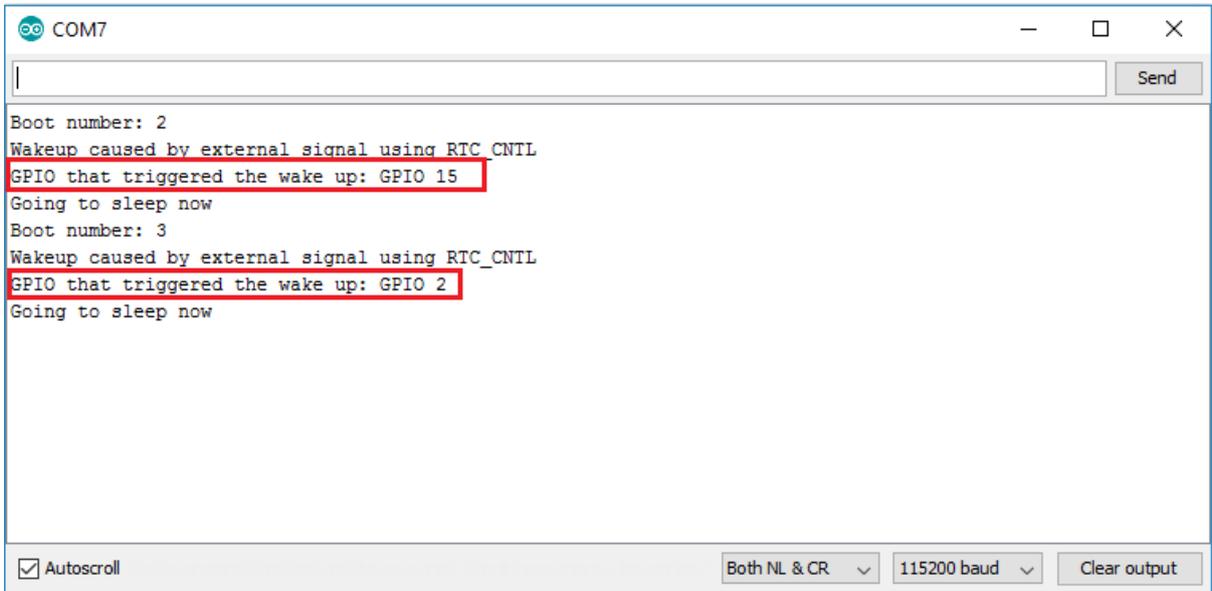
Having two buttons connected to GPIO 2 and GPIO 15, you can upload the code provided to your ESP32. Make sure you have the right board and COM port selected.

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



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

You should get something similar on the Serial Monitor.



Wrapping Up

In this unit you've learned how to wake up the ESP32 using an external wake-up. This means that you are now able to wake up the ESP32 by triggering the state of a GPIO pin.

In summary:

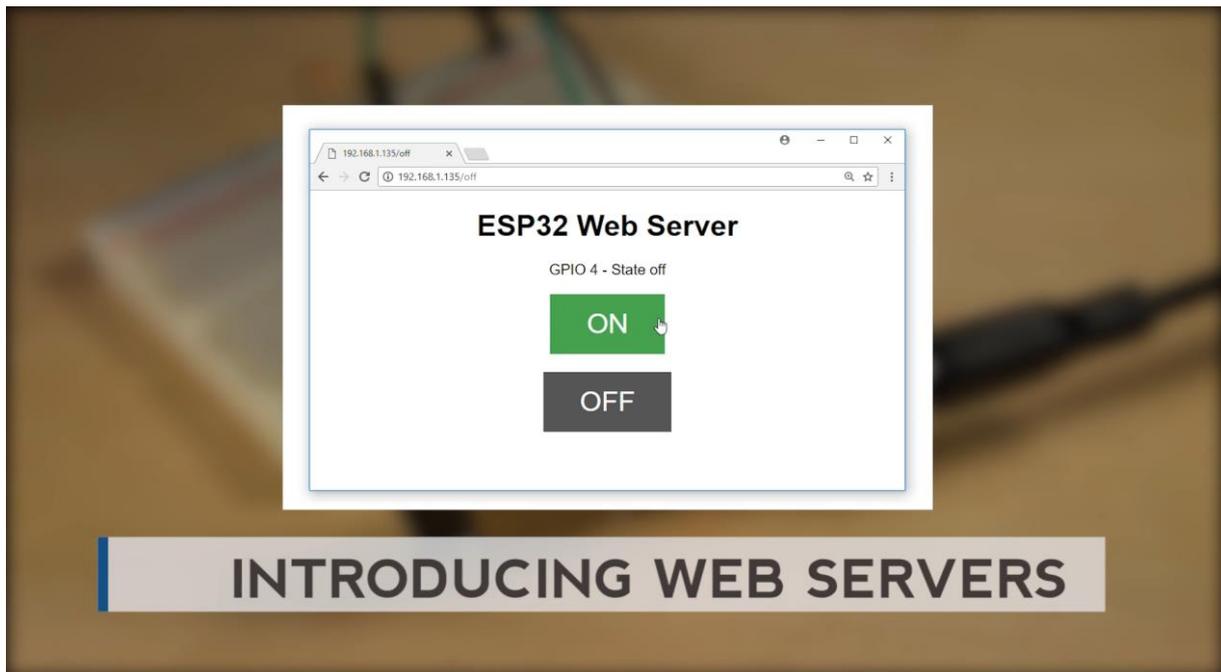
- You can only use RTC GPIOs as an external wake up;
- You can use two different methods: ext0 and ext1;
- ext0 allows you to wake up the ESP32 using one single GPIO pin;
- ext1 allows you to wake up the ESP32 using several GPIO pins.



MODULE 4

ESP32 Web Servers

Unit 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 remotely control outputs through the web, and we'll also show you how you can display sensor readings in a web page. After creating the web server, we'll show you how you can add more outputs and sensor readings to fulfill your needs, and how you can customize the appearance of your web page. 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 this Unit you'll learn what a web server is and how an ESP32 web server works. We'll take a look at some terms that you've probably heard before, but you may not know exactly what they mean.

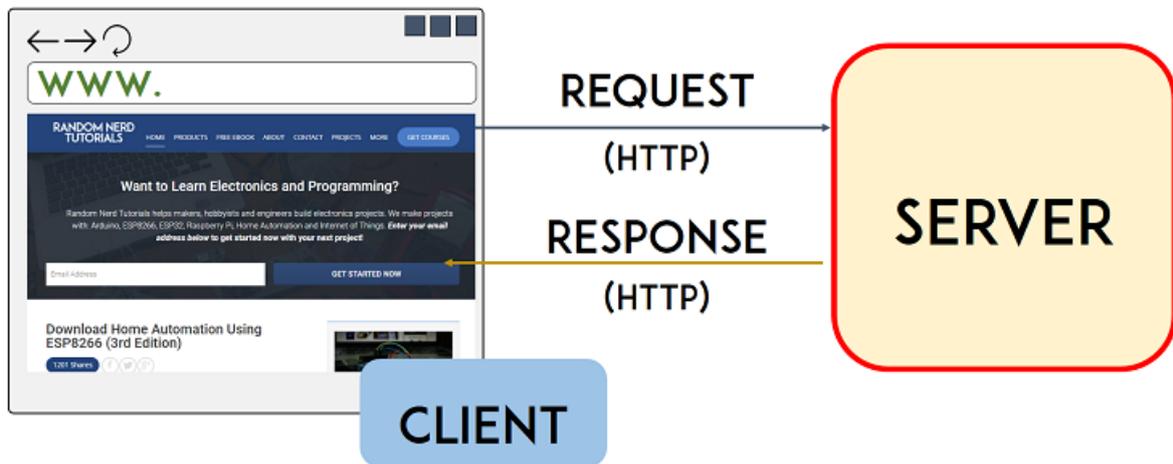
Request-response

Request-response is a message exchange pattern, in which a requestor sends a request message to a replier system that receives and processes the request, and returns a message in response. This is a simple, yet powerful messaging pattern especially in client-server architectures.



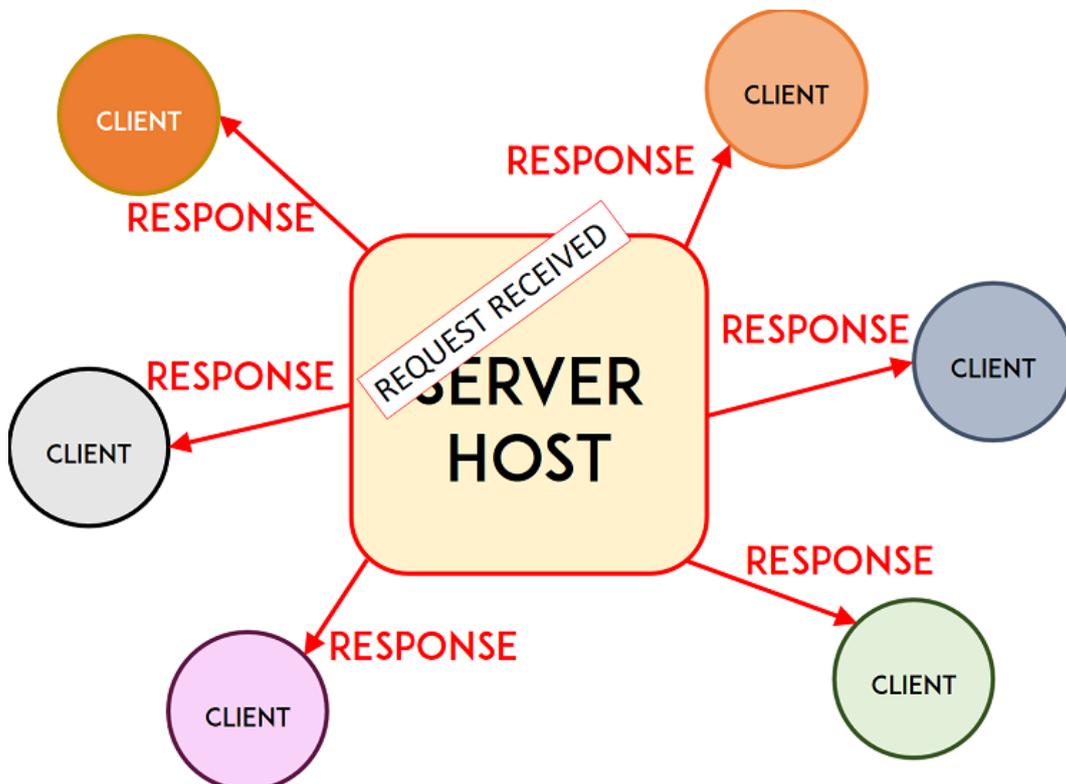
Client-Server

When you type an URL in your browser, what happens in the background is that 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 see the web page you requested in your browser. Clients and servers communicate over a computer network.

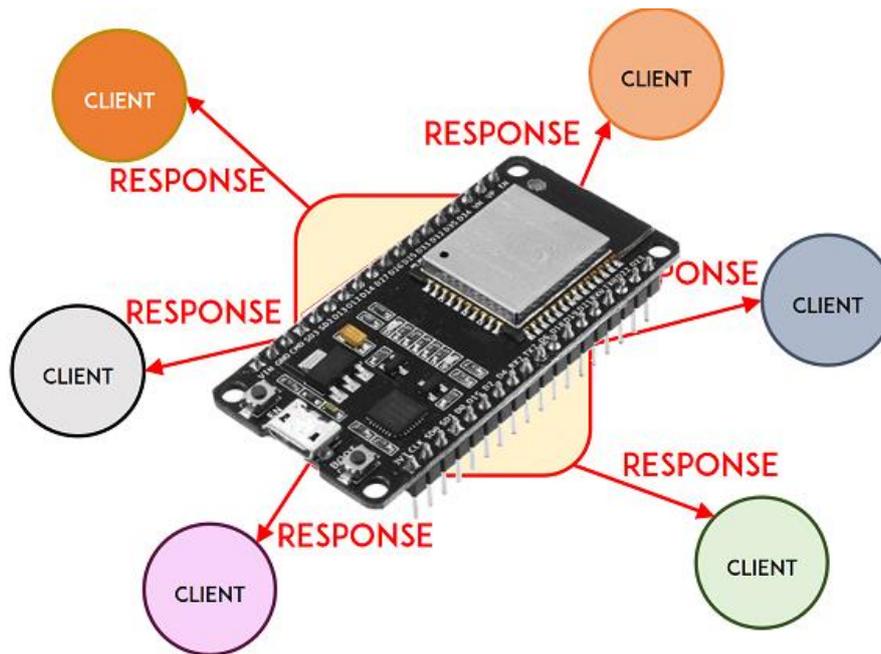


Server Host

A server host runs one or more server programs to share their resources with clients. So, you can imagine a web server as a piece of software that listens for incoming HTTP requests, and sends responses when requested.

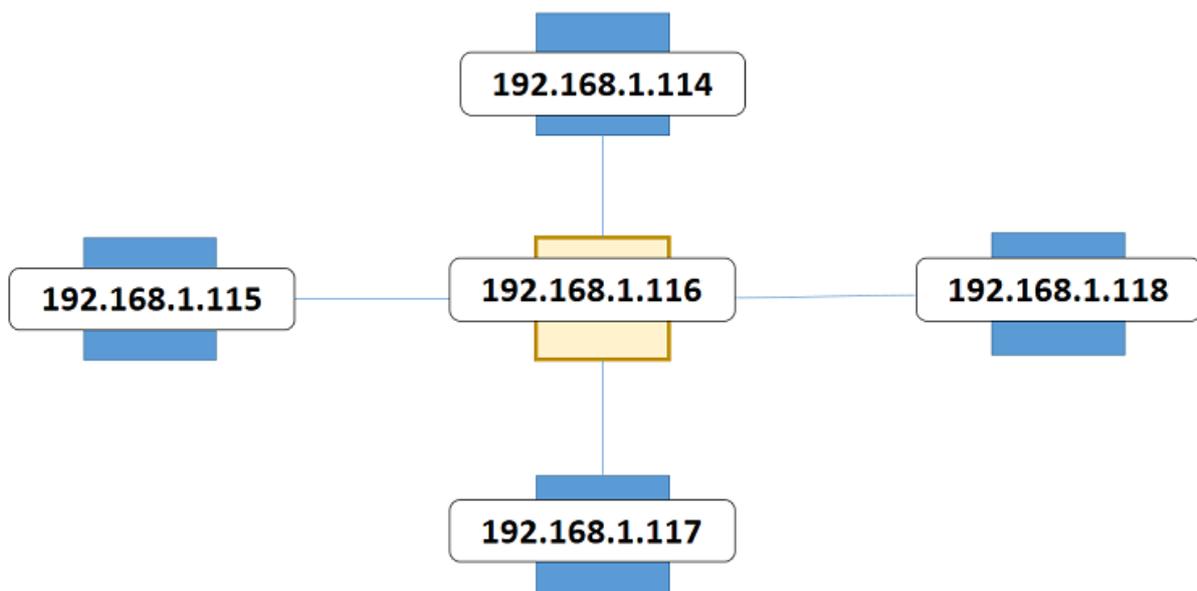


Your ESP can act as a server host, listening for HTTP requests from clients. When a new client makes a request, the ESP sends an HTTP response.



IP Address

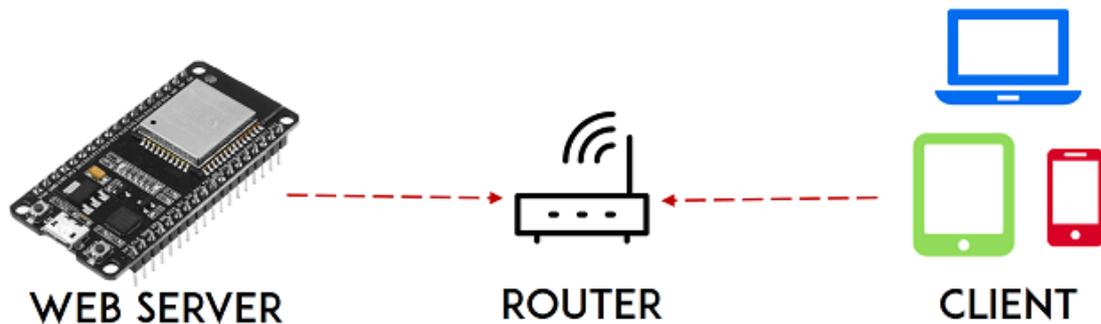
An IP address is a numerical label assigned to each device connected to a computer network. This way, any information sent to that device can reach it by referring to its IP address. So, your ESP has an IP address too.



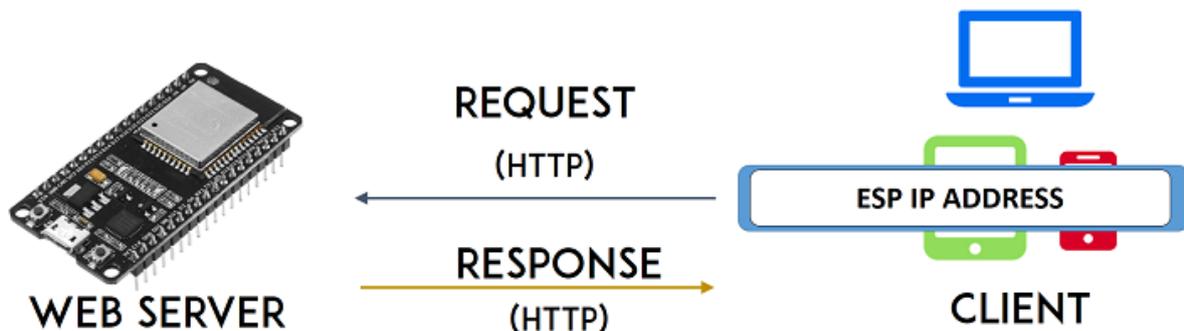
ESP32 Web Server

Let's take a look at a practical example with an IoT device called ESP32 that acts as a web server in the local network.

Typically, a web server with the ESP32 in the local network looks something 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 ESP IP address in your browser, you are sending an HTTP request to your ESP32. Then, the ESP32 responds back with a response that can contain a value, a reading, HTML text to display a web page, or any data you programmed in your ESP.



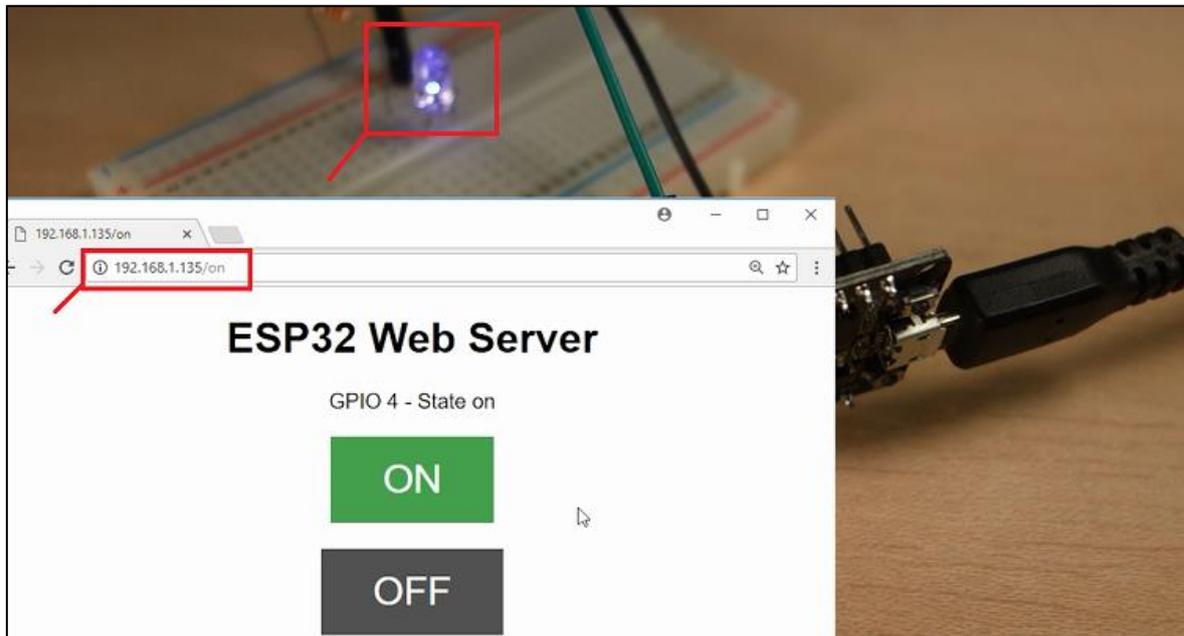
Web Server Example

So, how can you put all of this together to make IoT projects with your ESP? Your ESP32 has GPIOs, so you can connect devices, and control them through the web.

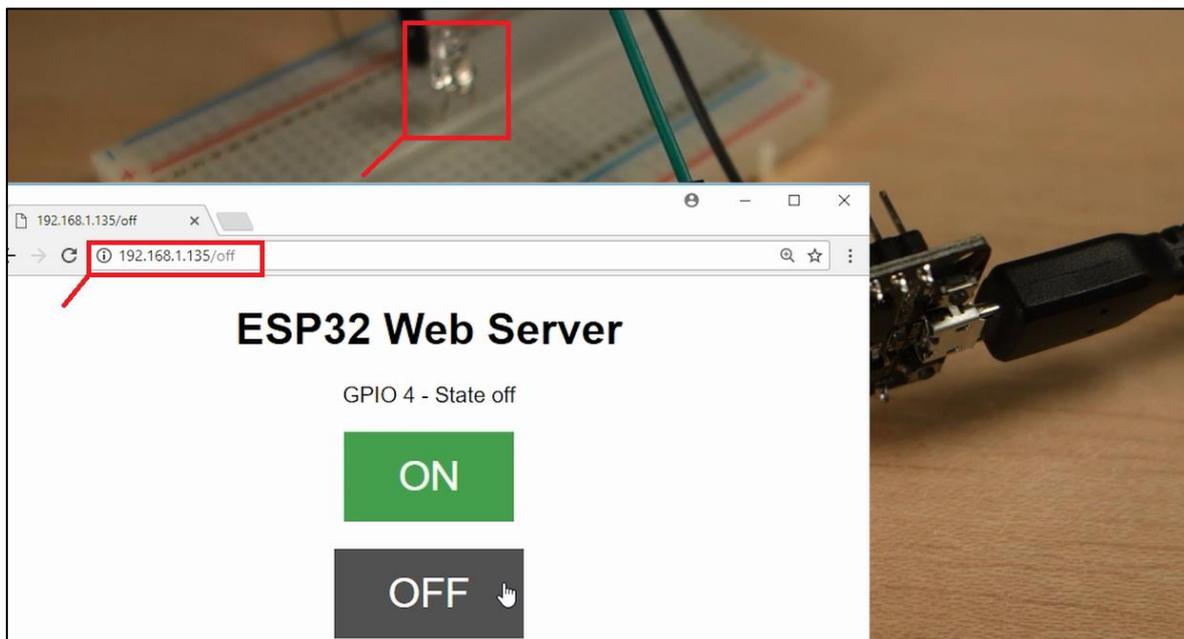
Here's an example of a web server we've built to control an output. The following web page shows up when you enter the ESP IP address in a browser.



When you press the ON button, the URL changes to the ESP IP address followed by /ON. The ESP receives a request on that URL, so it checks with an if statement which URL is being requested and changes the LED state accordingly.



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.



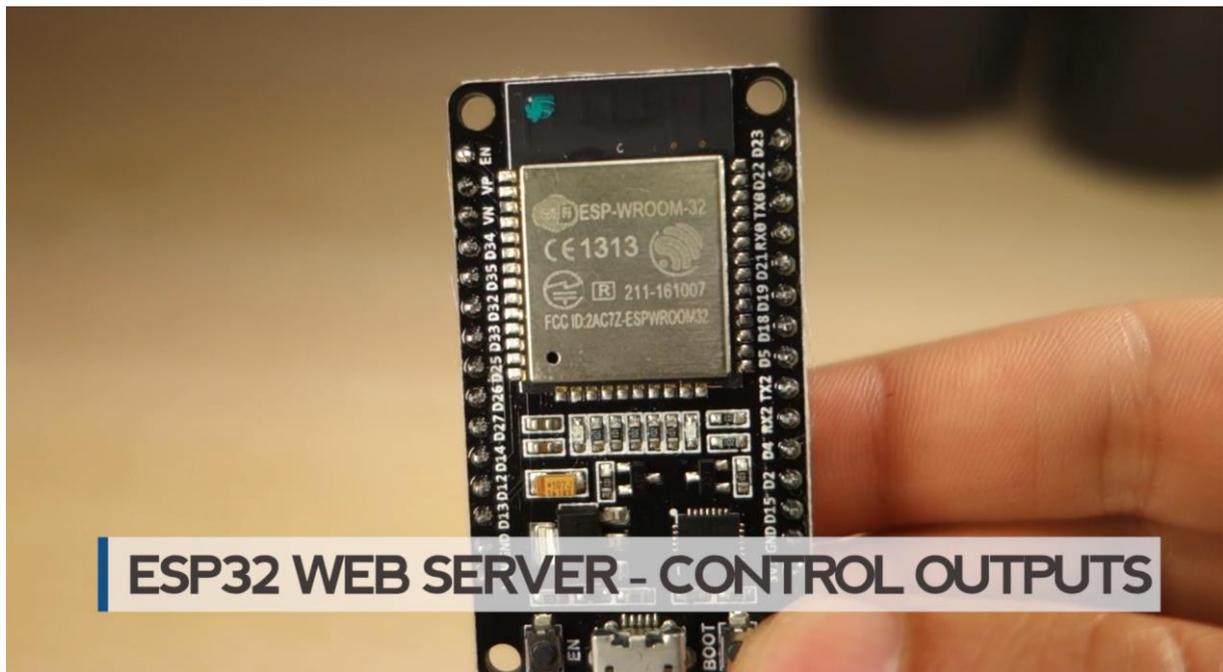
The same concept can be applied to control as many outputs as your ESP32 allows you to.

Wrapping Up

This was just an overview on 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.

Unit 2 - Web Server – Control Outputs



In this Unit you're going to 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 that has a browser: smartphone, tablet, laptop, on the local network.

Project Outline

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

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

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

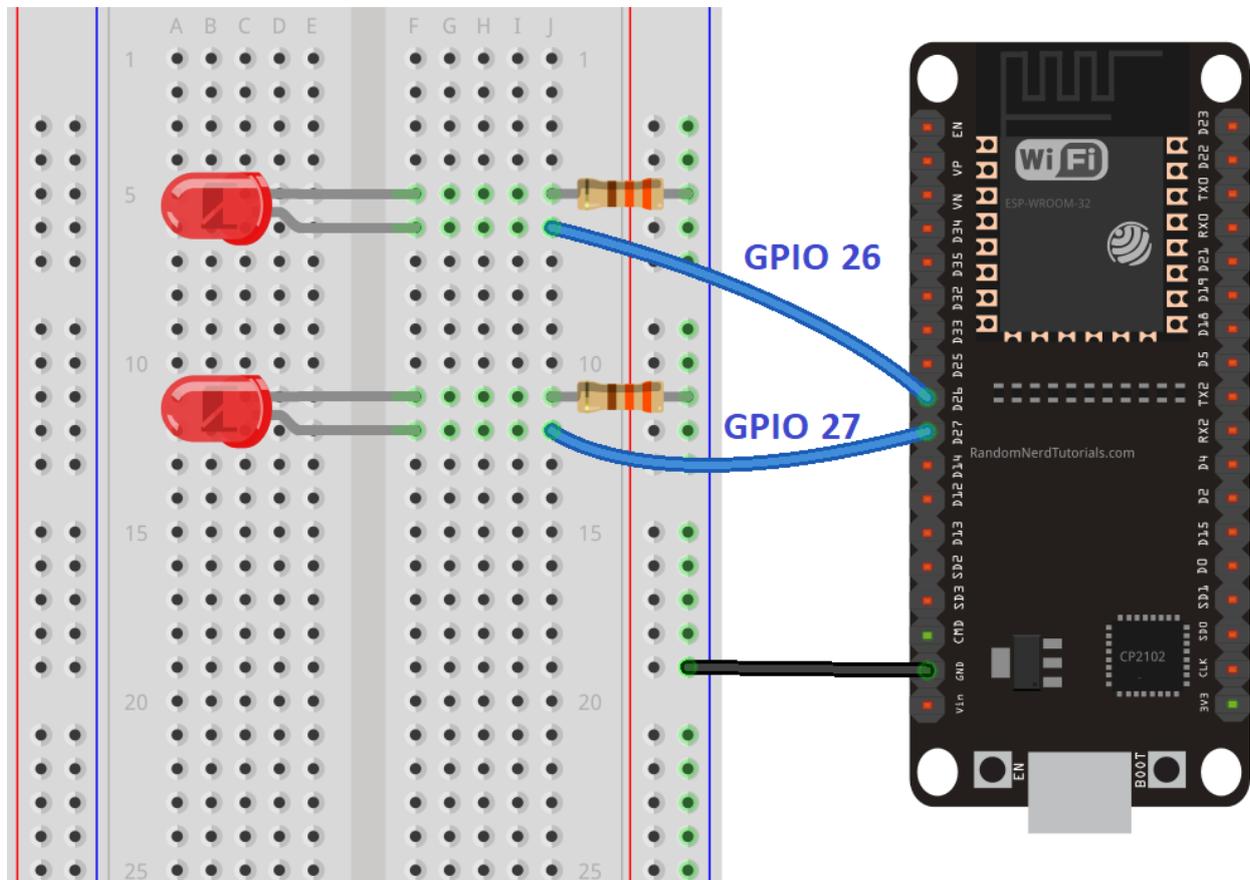
Schematic

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 330 Ohm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)



(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.)

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 for you.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi_Web_Server_Outputs/WiFi_Web_Server_Outputs.ino

```

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

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

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

```

```

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

// Variable to store the HTTP request
String header;

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

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

// Current time
unsigned long currentTime = millis();
// Previous time
unsigned long previousTime = 0;
// Define timeout time in milliseconds (example: 2000ms = 2s)
const long timeoutTime = 2000;

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 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 // make a String to hold incoming
    String currentLine = ""; // data from the client
    while (client.connected() && currentTime - previousTime <= timeoutTime)
  { // loop while the client's connected
    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();

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

// 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>");

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

// Display current state, and ON/OFF buttons for GPIO 26
client.println("<p>GPIO 26 - State " + output26State + "</p>");
// If the output26State is off, it displays the ON button
if (output26State=="off") {
client.println("<p><a                                href=\"/26/on\"><button
class=\"button\">ON</button></a></p>");
} else {
client.println("<p><a href=\"/26/off\"><button class=\"button
button2\">OFF</button></a></p>");
}

// Display current state, and ON/OFF buttons for GPIO 27

```

```

        client.println("<p>GPIO 27 - State " + output27State + "</p>");
        // If the output27State is off, it displays the ON button
        if (output27State=="off") {
            client.println("<p><a href=\""/27/on\"><button
class=\"button\">ON</button></a></p>");
        } else {
            client.println("<p><a href=\""/27/off\"><button class=\"button
button2\">OFF</button></a></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("");
}
}

```

Setting Your Network Credentials

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

```

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

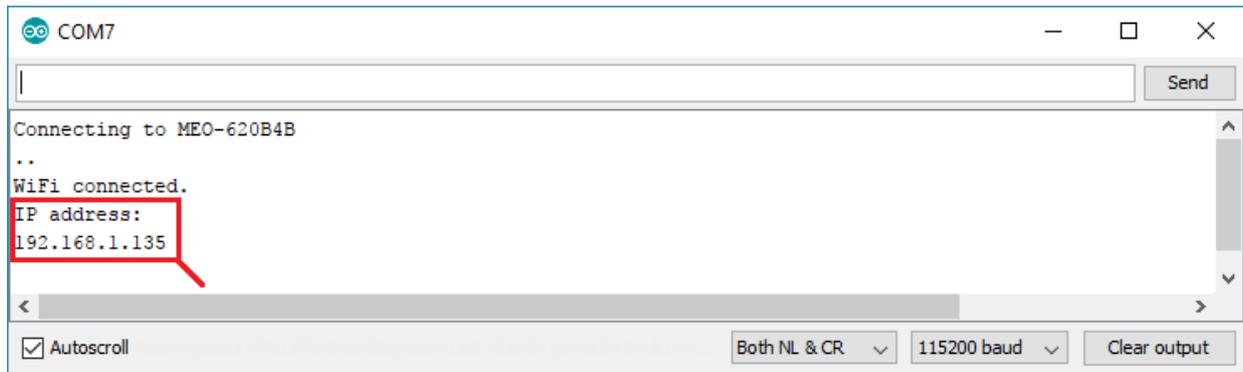
```

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, otherwise you'll get an error when trying to upload. Open the Serial Monitor at a baud rate of 115200.



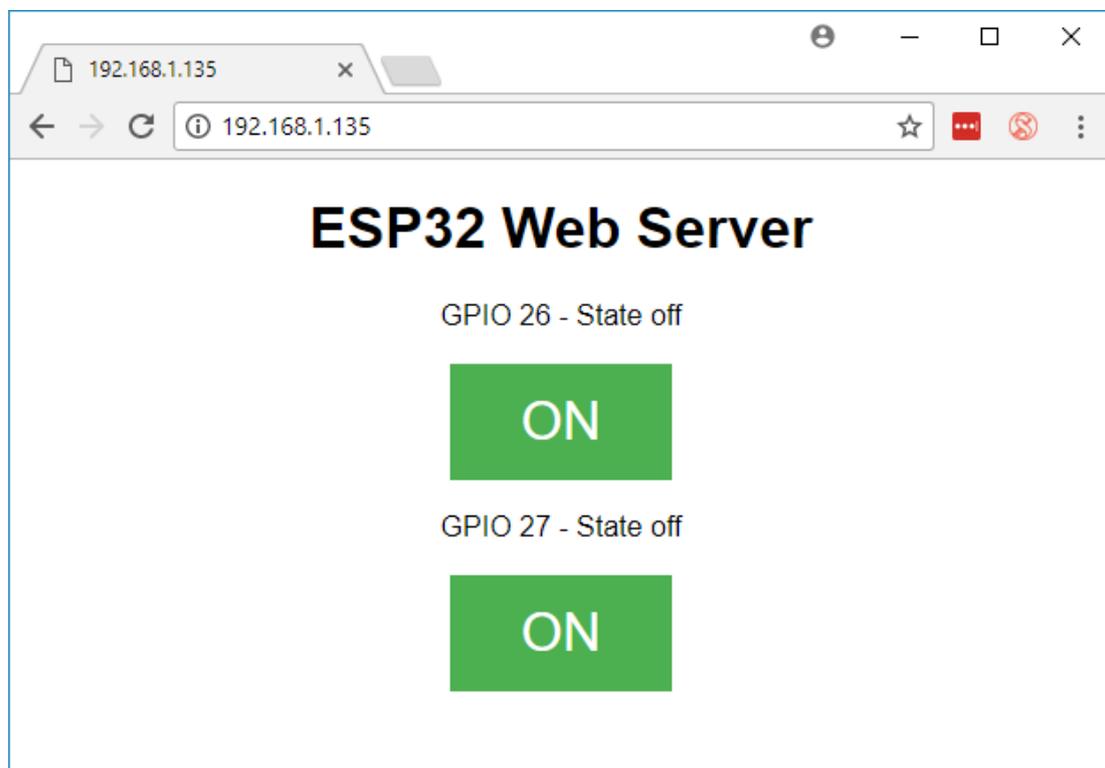
The ESP32 connects to Wi-Fi, and outputs the ESP IP address on the Serial Monitor. Copy that IP address, because you need it to access the ESP32 web server.



Note: If nothing shows up on the Serial Monitor, press the ESP32 “EN” button (enable button next to the micro USB port).

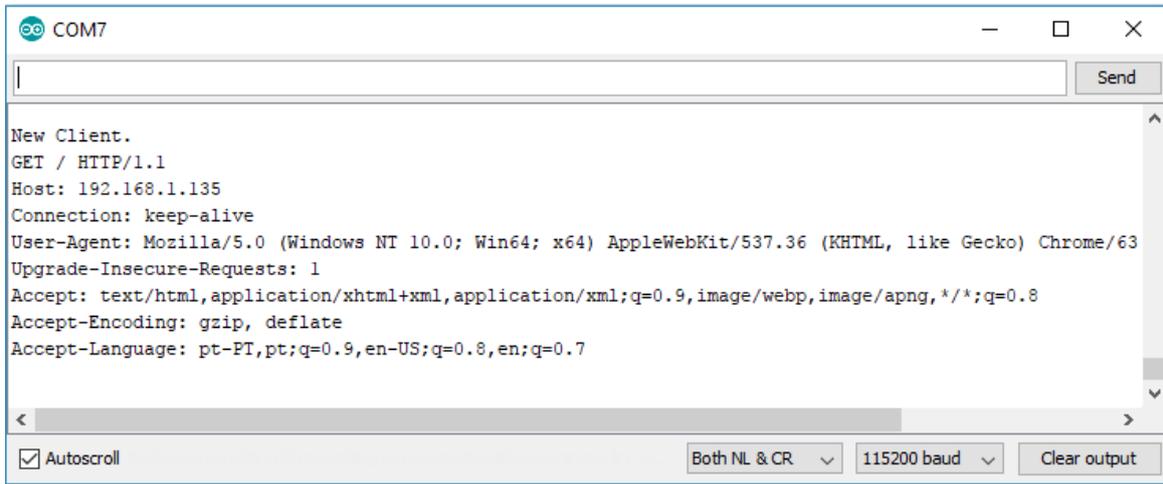
Accessing the Web Server

Open your browser, paste the ESP32 IP address, and you’ll see the following page.



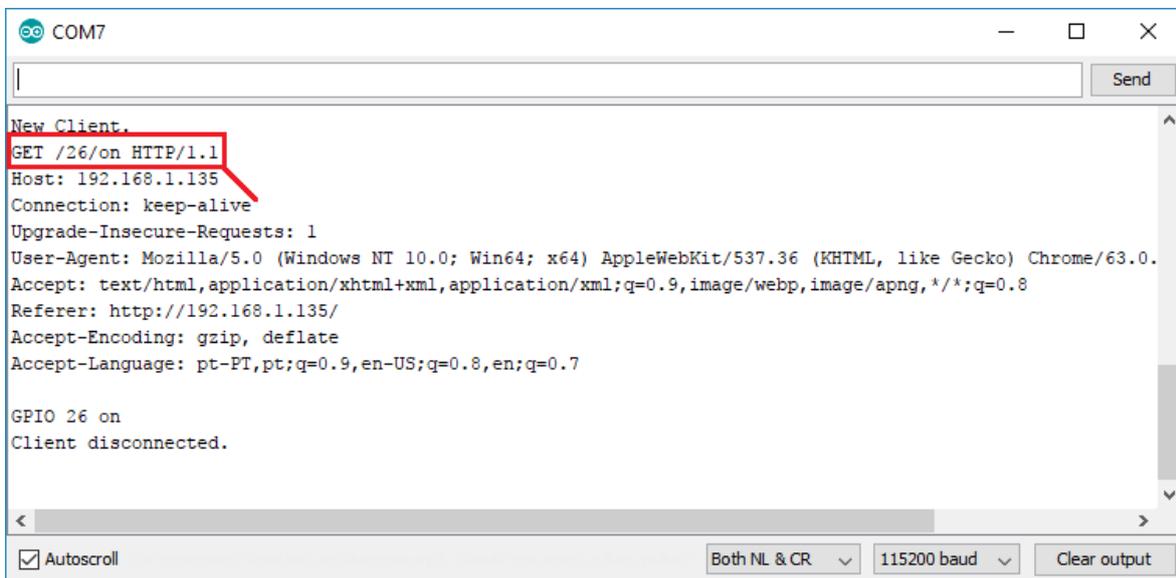
If you take a look at the Serial Monitor, you can see what’s going on on the background. The ESP receives an HTTP request from a new client (in this case, your browser).

You can also see other information about the HTTP request, the HTTP header fields that define the operating parameters of an HTTP transaction.



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 ESP 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.

How the code Works

Now, let's take a closer look at the code to see how it works, so that you are able to modify it to fulfill your needs.

The first thing you need to do is to include the WiFi library. This is the same library used to create a web server with the Arduino using the Ethernet shield.

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

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

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

Then, you set your web server to port 80.

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

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

```
String header;
```

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

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

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

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

setup()

Now, let's go into the `setup()`. The `setup()` function only runs once when your ESP 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 ESP IP address in the Serial Monitor.

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

loop()

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

The ESP is always listening for incoming clients with this line:

```
WiFiClient client = server.available();
```

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

```
if (client) { // If a new client connects,  
    Serial.println("New Client."); // print a message out in the serial port  
    String currentLine = ""; // make a String to hold incoming data  
    while (client.connected()) { // loop while the client's connected  
        if (client.available()) { // if there's bytes to read from 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 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 start with a response code (e.g. HTTP/1.1 200 OK)  
                    // and a content-type so the client knows what's coming  
                    client.println("HTTP/1.1 200 OK");  
                    client.println("Content-type:text/html");  
                    client.println("Connection: close");  
                    client.println();
```

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

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

For example, if you've pressed the GPIO 26 ON button, the ESP receives a request on the **/26/ON** URL, and we receive that information on the HTTP header. So, we can check if the header contains the expression GET **/26/on**. If it contains, it will print a message on the Serial Monitor, it will change the output26statevariable to ON, and turns the LED on.

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

Displaying the HTML web page

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

Note: in Unit 3 and Unit 4, you'll learn about HTML and CSS basics, so that you can easily modify the web page to fulfill your needs.

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

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

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

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

```
client.println("<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">");
```

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

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

Styling the Web Page

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

```
client.println("<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}");
```

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

```
client.println(".button { background-color: #4CAF50; border: none; color: white; padding: 16px 40px;");
client.println("text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer;}");
```

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

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

Setting the Web Page First Heading

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

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

Displaying the Buttons and Corresponding State

Then, you write a paragraph to display the GPIO 26 current state. As you can see we use the output26Statevariable, so that the state updates instantly when this variable changes.

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

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

```
if (output27State=="off") {
  client.println("<p><a href=\"/27/on\"><button class=\"button\">ON</button></a></p>");
} else {
  client.println("<p><a href=\"/27/off\"><button class=\"button button2\">OFF</button></a></p>");
}
```

We use the same procedure for GPIO 27.

Closing the Connection

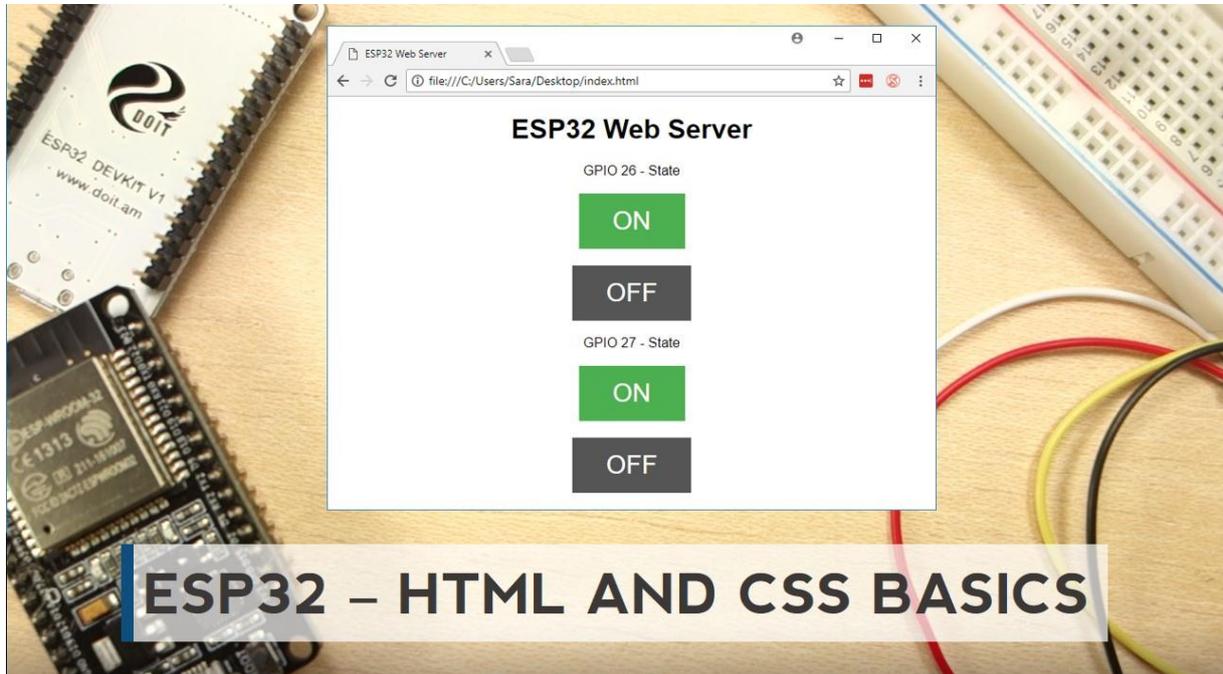
Finally, when the response ends, we clear the header variable, and stop the connection with the client with `client.stop()`.

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

Wrapping Up

Now that you know how the code works, you can modify the code to add more outputs, or modify your web page. To modify your web page you may need to know some HTML and CSS basics. We'll cover those topics in Units 3 and 4.

Unit 3 - ESP32 Web Server – HTML and CSS Basics (Part 1/2)



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 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. Then, we'll add the HTML text to the Arduino code, so that the ESP32 serves that web page when you make a request on its IP address. This Unit is a bit long so it is divided into two parts:

- **Part 1** – You'll learn HTML and CSS basics by building the web page in the previous project step-by-step
- **Part 2** – You'll learn how to add the HTML text to the Arduino IDE

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.

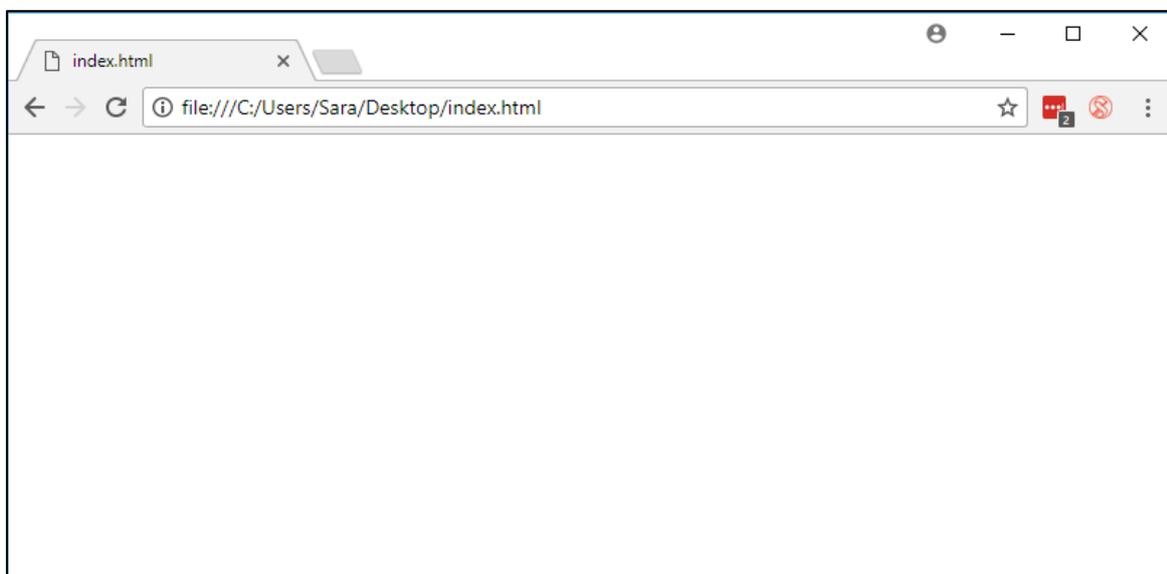
The structure of the web page should go between the `<html>` and `</html>` tags. The `<html>` tag indicates the beginning of a web page and the `</html>` tag indicates the end of the page.

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 content of the page like headings, text, buttons, tables, etc.

Open a Text Editor program (you can use any text editor you like, we use [Atom](#)) 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`.



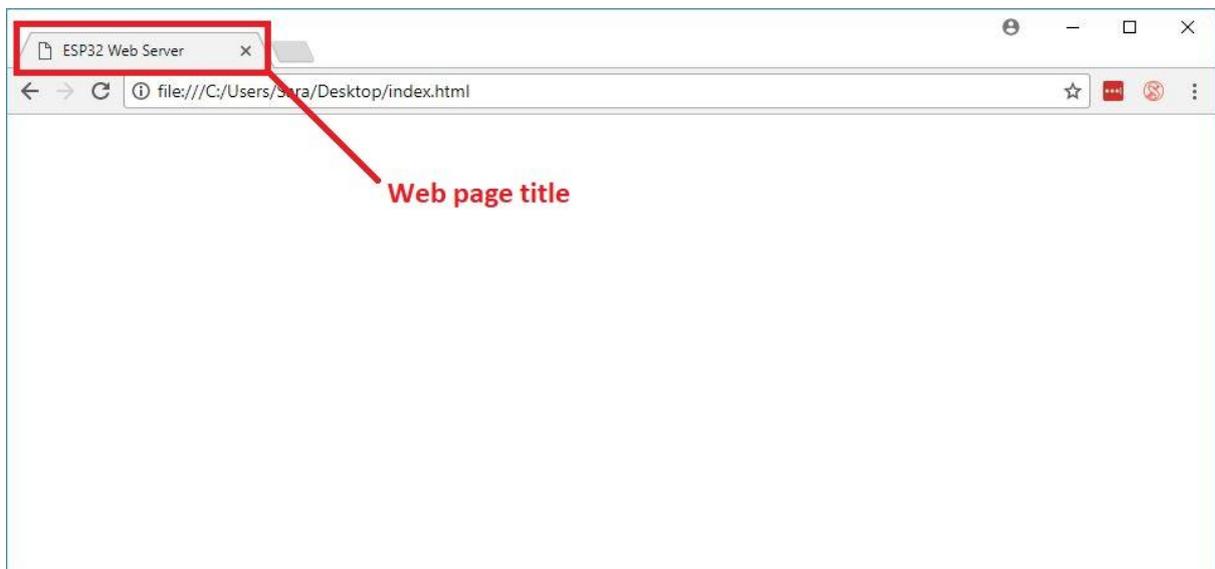
Title

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 at 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.

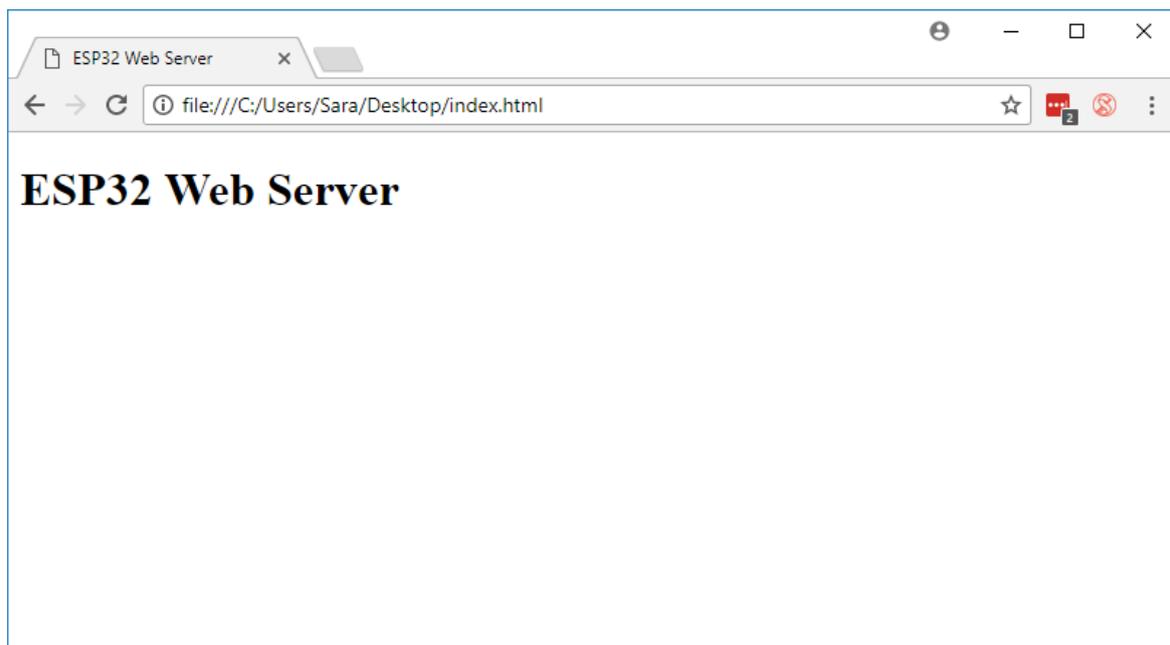
Add some headings to your document. You can use the following text as a reference.

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>ESP32 Web Server</title>
</head>
<body>
  <h1>ESP32 Web Server</h1>
</body>
</html>
```

Here we've added a first heading with the text **"ESP32 Web Server"**. We recommend you adding 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.

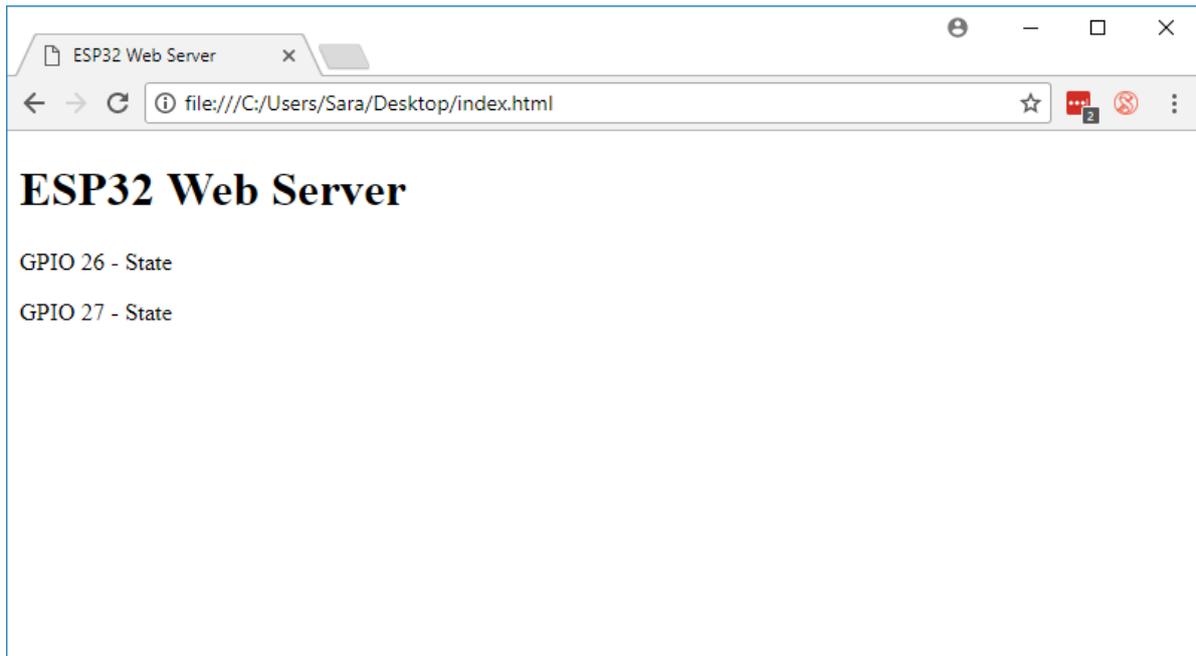


Paragraphs

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.



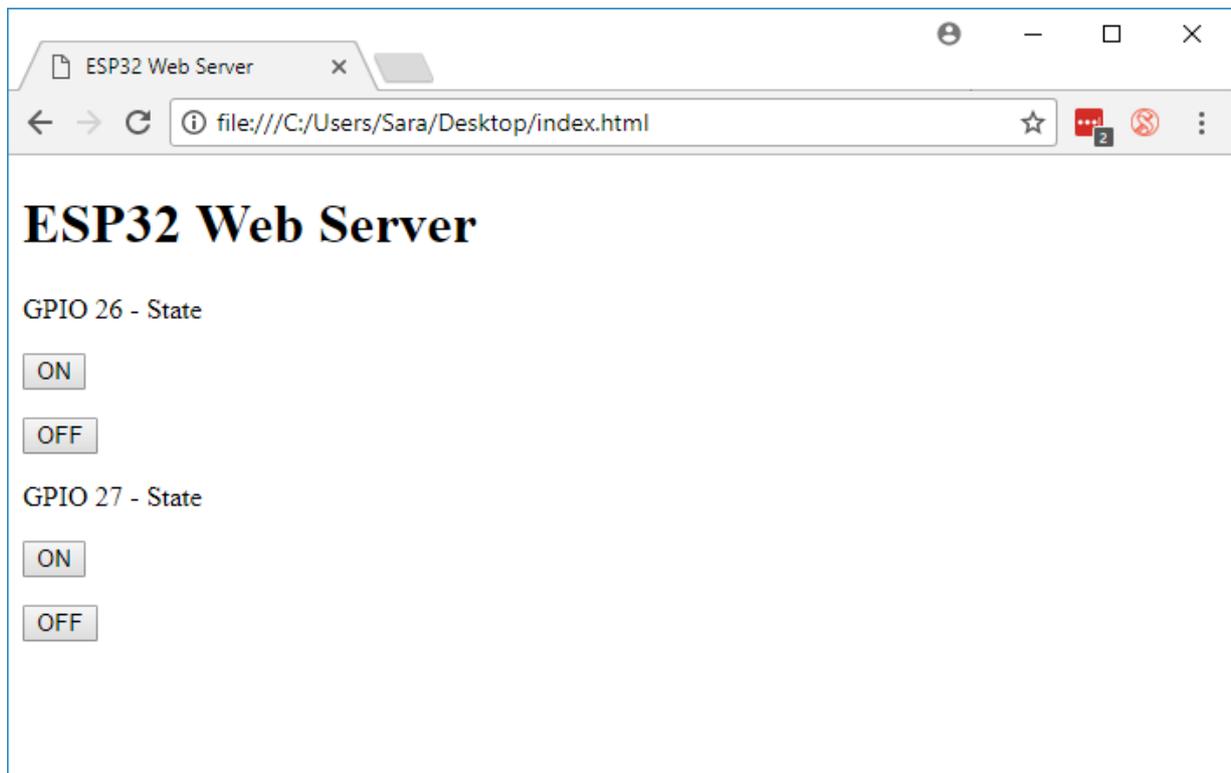
The “ON” and “OFF” states will then be added to each of the paragraphs using a variable created in the Arduino IDE code, that saves the state of each GPIO.

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 hyperlink 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'll take a look at this part of the code in the Arduino IDE later in this 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>
```

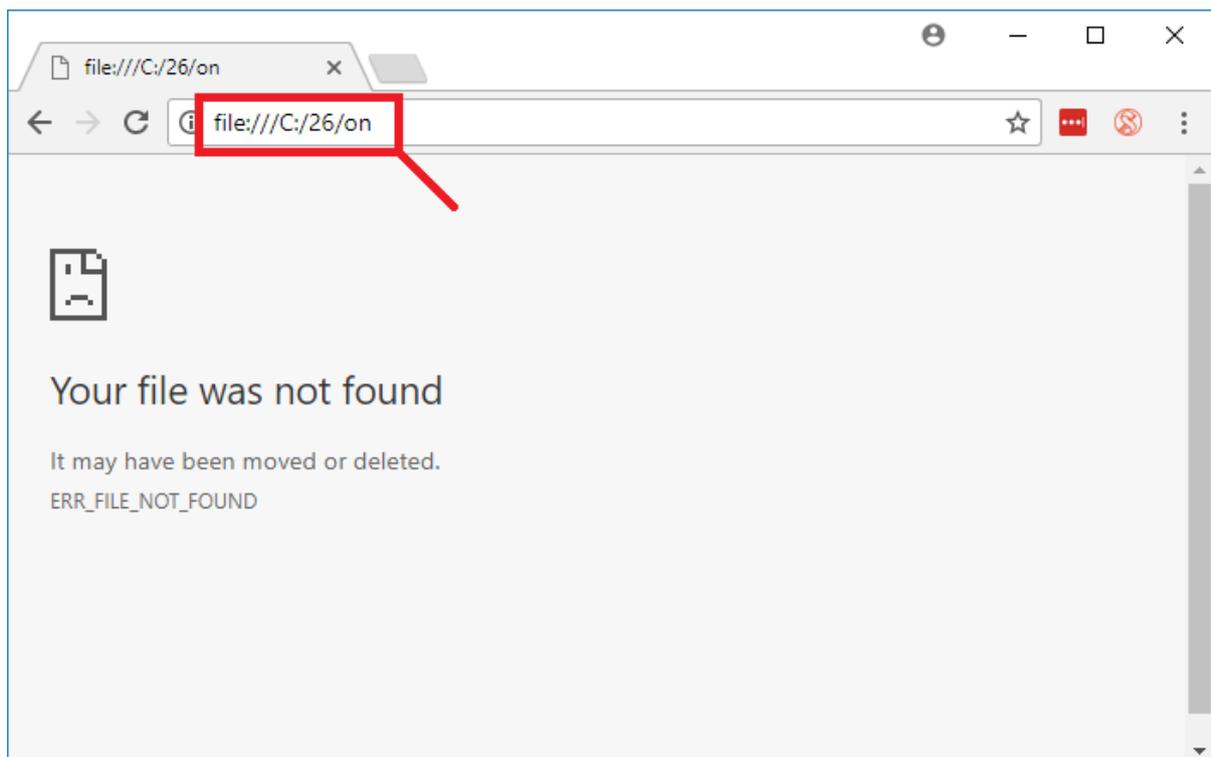
When you click the GPIO 26 OFF button, you want to redirect to /26/off:

```
<a href="/26/on"><button>ON</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 – 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 classnames for an element. Classnames 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 classname.

To add the class attribute to an HTML element, you use the following syntax:

```
<element class="classname">
```

An HTML element can have more than one classname that must be separated by a space. In our example we specify the classname “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 your 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. We're going to add the CSS to the HTML file because it is easier to add to the Arduino IDE.

When added to the HTML file, the CSS should go 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 certain 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.

Styling the page

In our example, we style the html element with the following style:

```
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:

```
<!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.

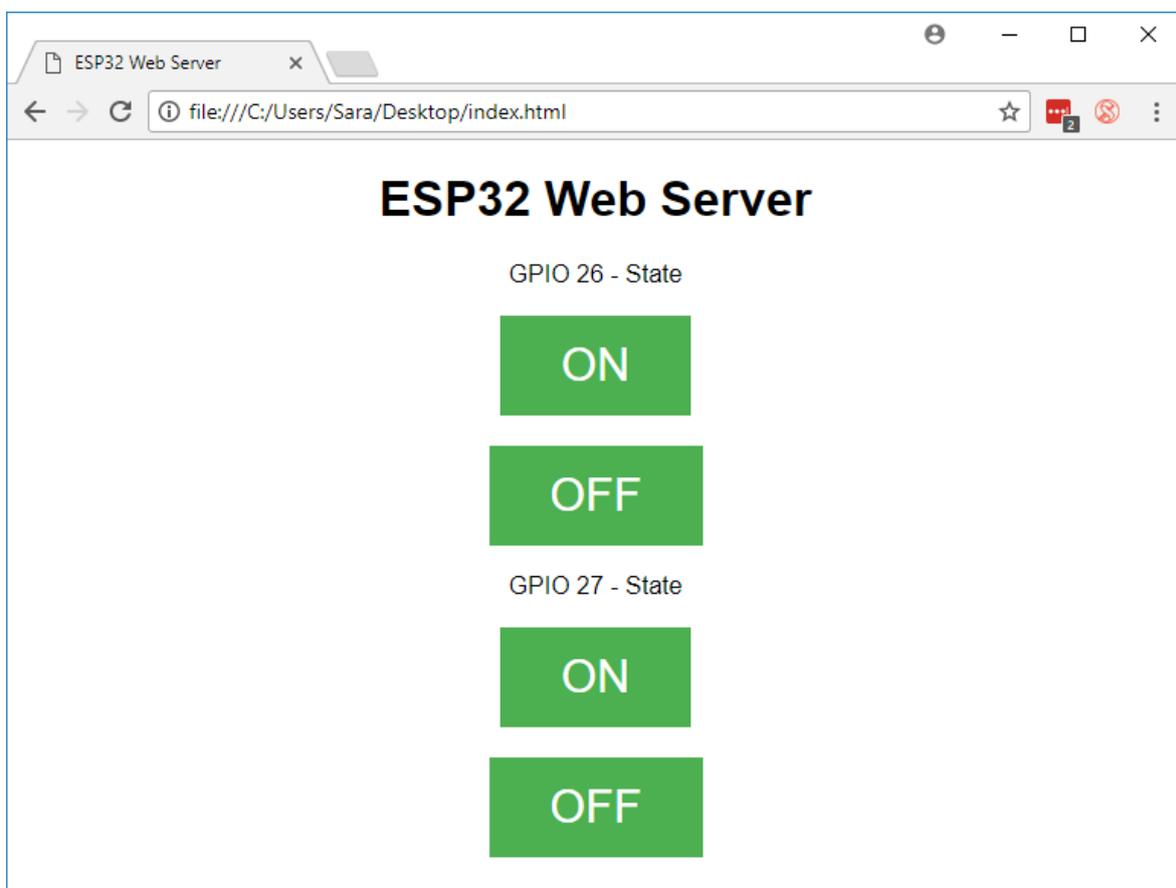
```
.button {
  background-color: #4CAF50;
  border: none;
  color: white;
  padding: 16px 40px;
  text-decoration: none;
  font-size: 30px;
  margin: 2px;
  cursor: pointer;
}
```

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 will change 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 would 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:

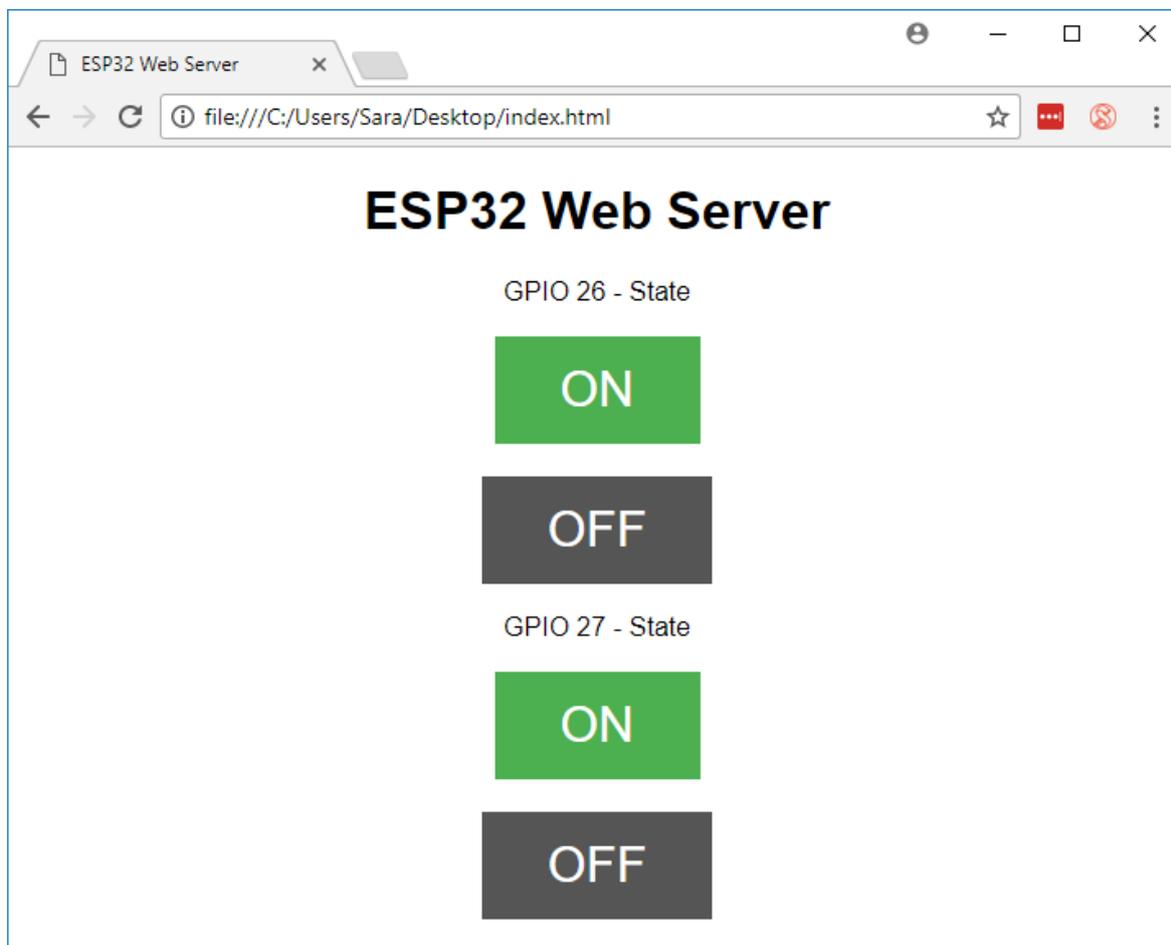


The off button belongs to classname “button” and classname “button2”, so it will have properties from both. We define the style for the “button2” classname as follows:

```
.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" classname 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.

Complete HTML Text

Here is how your complete HTML file should be.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi_Web_Server_Outputs/index.html

```
<!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;
    }
  </style>
</head>
</html>
```

```

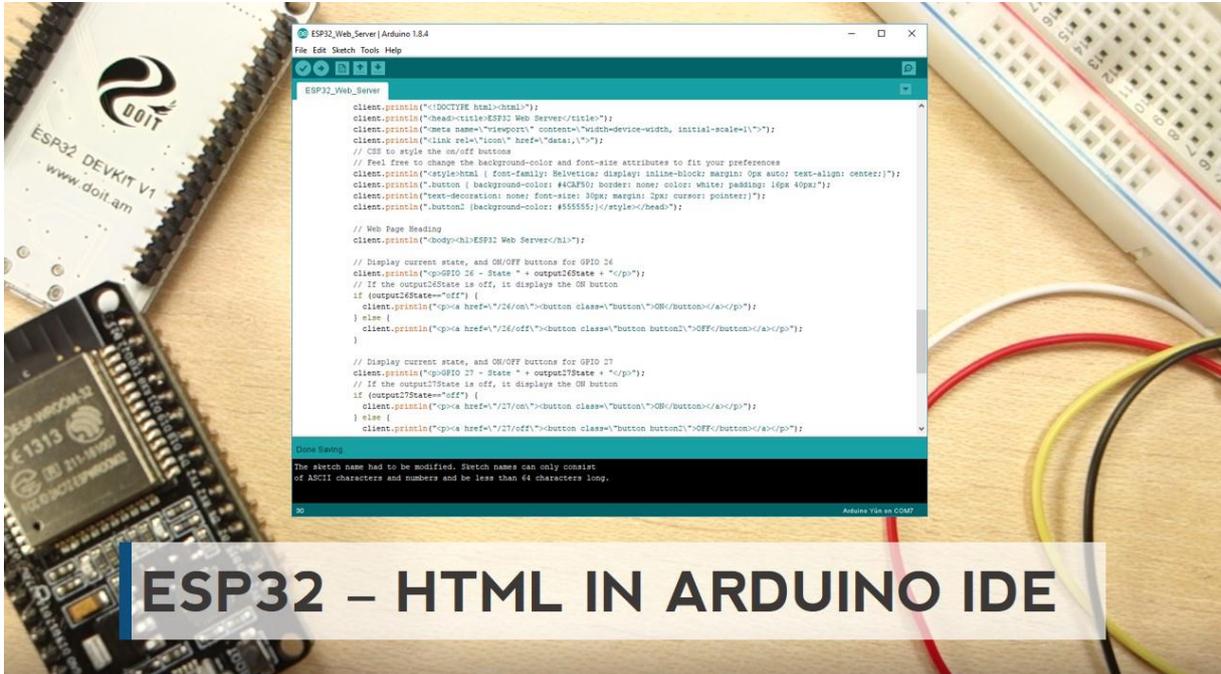
    text-align: center;
  }
  .button {
    background-color: #4CAF50;
    border: none;
    color: white;
    padding: 16px 40px;
    text-decoration: none;
    font-size: 30px;
    margin: 2px;
    cursor: pointer;
  }
  .button2 {
    background-color: #555555;
  }
</style>
</head>
<body>
  <h1>ESP32 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>

```

Continue to the Next Unit

Continue reading the next Unit to learn how to serve the web page you've just build with your ESP32 using Arduino IDE.

Unit 4 - ESP32 Web Server – HTML in Arduino IDE (Part 2/2)



In the previous Unit you've learn some HTML and CSS basics, and you've built the web page for your web server. In this Unit you're going to learn how to add your HTML text to the code in Arduino IDE. Basically, the ESP32 needs to send your HTML text as a response when you make a request on the ESP32 IP address.

Before proceeding, refresh your memory by taking a look at the code used in the "ESP32 Web Server – Control Outputs" Unit.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi_Web_Server_Outputs/WiFi_Web_Server_Outputs.ino

```
/*  
  Rui Santos  
  Complete project details at http://randomnerdtutorials.com  
  */  
  
// Load Wi-Fi library  
#include <WiFi.h>  
  
// Replace with your network credentials  
const char* ssid = "  
const char* password = "  
  
// Set web server port number to 80  
WiFiServer server(80);  
  
// Variable to store the HTTP request
```

```

String header;

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

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

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 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,
    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();

```

```

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

// 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>");

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

// Display current state, and ON/OFF buttons for GPIO 26
client.println("<p>GPIO 26 - State " + output26State + "</p>");
// If the output26State is off, it displays the ON
button
if (output26State=="off") {
    client.println("<p><a href=\"/26/on\"><button
class=\"button\">ON</button></a></p>");
} else {
    client.println("<p><a href=\"/26/off\"><button class=\"button
button2\">OFF</button></a></p>");
}

// Display current state, and ON/OFF buttons for GPIO 27
client.println("<p>GPIO 27 - State " + output27State + "</p>");
// If the output27State is off, it displays the ON
button
if (output27State=="off") {
    client.println("<p><a href=\"/27/on\"><button
class=\"button\">ON</button></a></p>");
} else {
    client.println("<p><a href=\"/27/off\"><button class=\"button
button2\">OFF</button></a></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("");
}
}

```

client.println()

Previously, you've seen that the ESP32 sends text to the client using `client.println()`. You should write as an argument the text you want to send – it must be a string. So, it must go between double quotes. For example, to send the first line of your `index.html` file:

```
client.println("<!DOCTYPE html><html>");
```

Sending double quotes to the client

There are some parts of the HTML file that contain double quotes (""), like the following line:

```
<p><a href="/27/on"><button class="button">ON</button></a></p>
```

To send those double quotes to the client, without interfering with the double quotes of the `client.println()`, you need to add an escape character called backslash (\) before the double quotes.

For example, to send the preceding HTML line to the client, you need to add the backslashes highlighted in yellow.

```
client.println("<p><a href=\"/26/on\"><button class=\"button\">ON</button></a></p>");
```

Indentation

In HTML, it doesn't matter how you indent your code. Indentation doesn't change the way HTML works. So, you can compact all your HTML in the same line, and it will work the same:

```
<!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>
```

So, you can put all that HTML in one single `client.println("")` line. However, if you do that, your code will become very difficult to read. So, we recommend “indenting” your HTML file in a way that is compact, but readable at the same time.

Sending HTML to the Client

In summary, to send all HTML text to the client:

- 1) Compact your HTML text in some lines, while being readable at the same time
- 2) Add a backslash (\) before every double quote
- 3) Use the `client.println()` function to send the HTML text to the client.

To send all your HTML text to the client, you would do as follows:

```
// 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;}</style>");
client.println(".button { background-color: #4CAF50; border: none; color:
white; padding: 16px 40px;}</style>");
client.println("text-decoration: none; font-size: 30px; margin: 2px;
cursor: pointer;}</style>");
client.println(".button2 {background-color: #555555;}</style></head>");

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

// Display current state, and ON/OFF buttons for GPIO 26
client.println("<p>GPIO 26 - State " + output26State + "</p>");
// If the output26State is off, it displays the ON button
if (output26State=="off") {
  client.println("<p><a href=\"/26/on\"><button
class=\"button\">ON</button></a></p>");
} else {
  client.println("<p><a href=\"/26/off\"><button class=\"button
button2\">OFF</button></a></p>");
}

// Display current state, and ON/OFF buttons for GPIO 27
client.println("<p>GPIO 27 - State " + output27State + "</p>");
// If the output27State is off, it displays the ON button
if (output27State=="off") {
  client.println("<p><a href=\"/27/on\"><button
class=\"button\">ON</button></a></p>");
} else {
```

```
client.println("<p><a href=\"/27/off\"><button class=\"button  
button2\">OFF</button></a></p>");  
}  
client.println("</body></html>");
```

Using Variables to Display the Current State

When you tested the ESP32 web server, you've seen that it updates the GPIO state automatically when you press one of the buttons. To do that, we've created two variables:

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

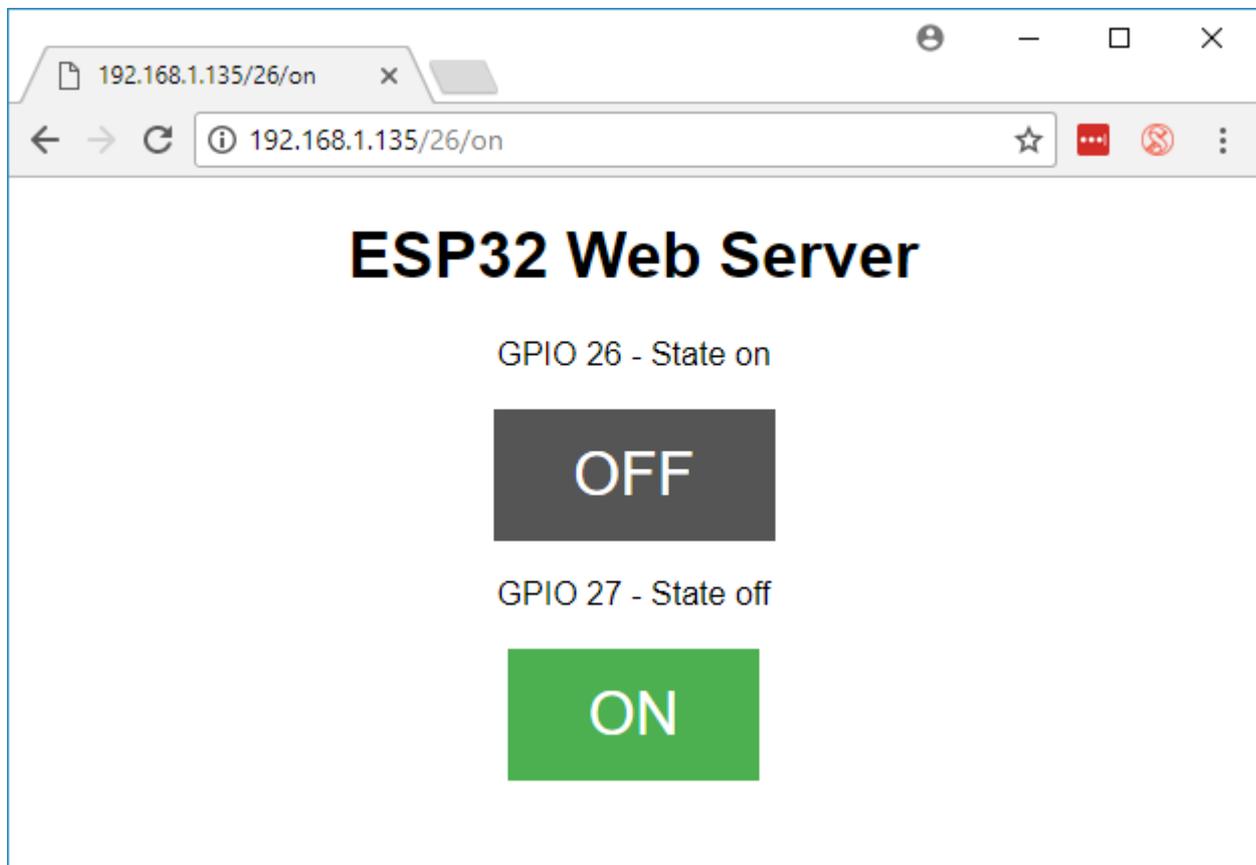
Those values save the state of each GPIO. To display the current state on the web page, you just need to send the state variable along the HTML code. In other words, you need to concatenate the variable with the HTML text you're sending. For example, to display the state of GPIO 27, you should do as follows:

```
client.println("<p>GPIO 27 - State " + output27State + "</p>");
```

You can use this method to send whatever variable you want.

Adding Conditions

The web server you've built in the ["ESP32 Web Server - Control Outputs" Unit](#), shows one button for each GPIO, depending on its current state.



If the current state of the GPIO is off, we show the ON button, if not, we display the OFF button. So, we can add conditions to the code to send either the HTML text to display the ON button, or the HTML text to display the off button, as shown in the snippet below:

```
if (output26State=="off") {
  client.println("<p><a href=\"/26/on\"><button
class=\"button\">ON</button></a></p>");
} else {
  client.println("<p><a href=\"/26/off\"><button class=\"button
button2\">OFF</button></a></p>");
}
```

Wrapping Up

In these two Units, you've learned how to build a simple web page using HTML and CSS. You've also learned how to send the HTML text to the client using code in the Arduino IDE. The idea behind these two Units is provide you the basics so that you are able to build and customize your own web pages.

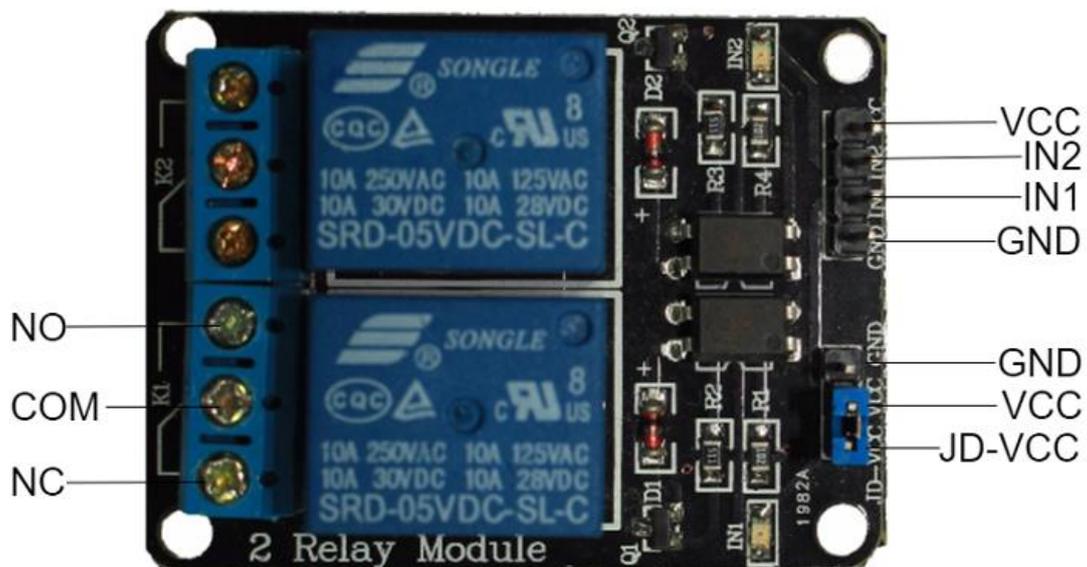
Unit 5 - ESP32 Web Server – Control Outputs (Relay)



In this Unit you're going to learn how to remotely control two 12V lamps using your ESP32 and a relay module. We'll modify the project in the "ESP32 Web Server – Control Outputs" Unit to control a relay module with two terminals instead of the LEDs.

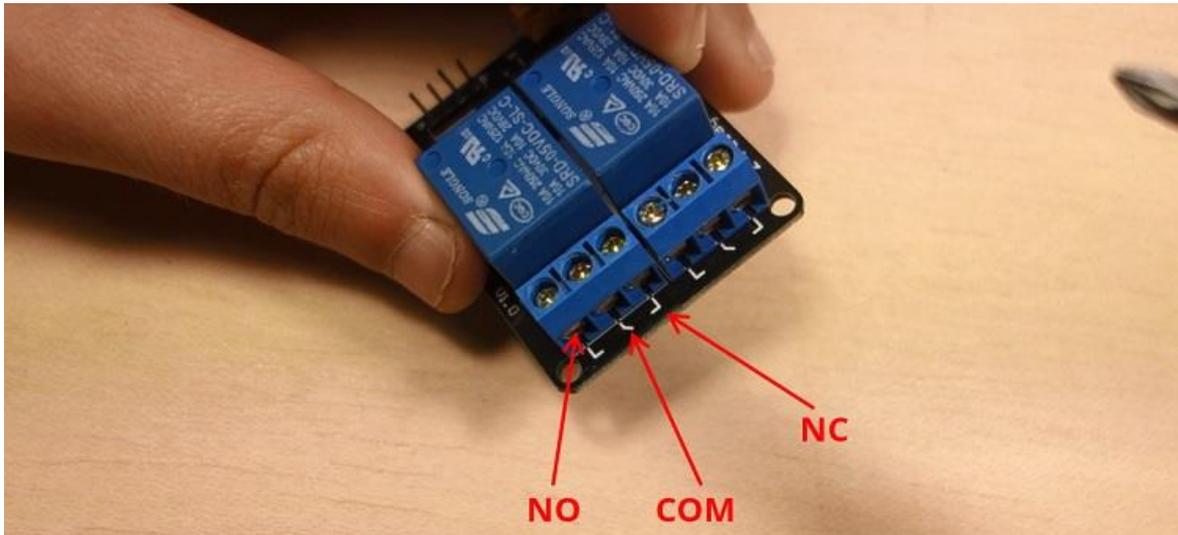
Introducing the Relay Module

A relay is an electrically operated switch that can be turned on or off, letting the current go through or not, and can be controlled with low voltages, like the 3.3V provided by the ESP32 GPIOs. The relay module we'll use in this project has two relays—the two blue cubes shown in the figure below. So, it is perfect to control two lamps.



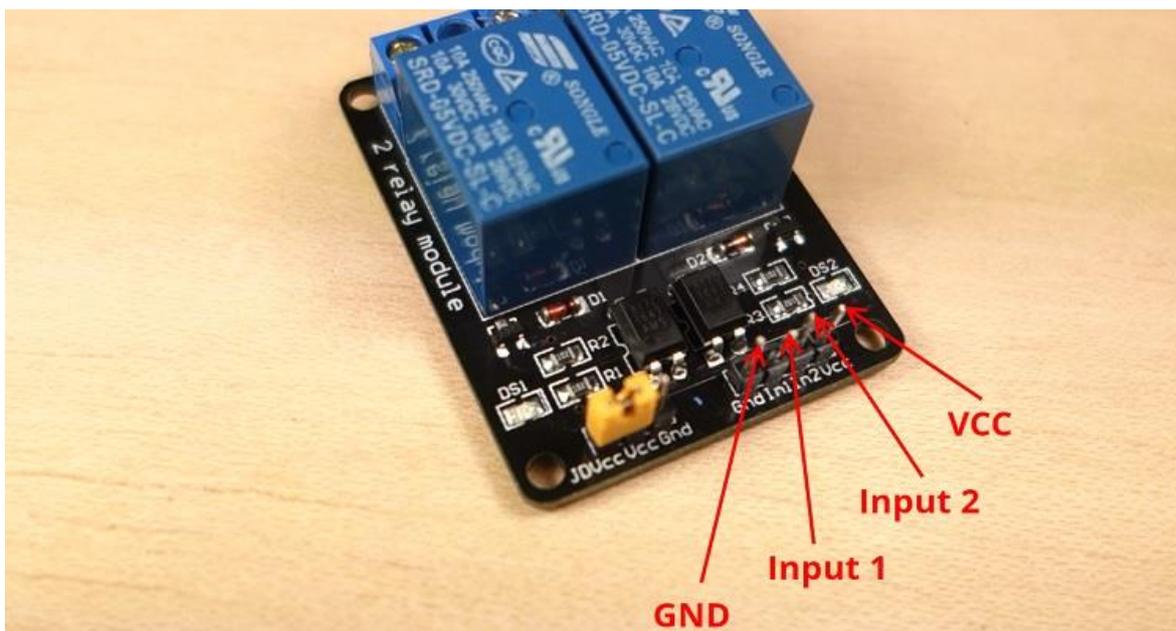
Relay Pinout

The left side of the relay module is the high-voltage side, and it has two connectors with three sockets: the common (COM), the normally closed (NC) and the normally open (NO).



The normally closed configuration is used when you want the relay to be closed by default, meaning the current is flowing unless you send a signal from to the relay module to open the circuit and stop the current flow. The normally open configuration works the other way around: the relay is always open, so the circuit is open unless you send a signal to close it.

On the low voltage side, you have a set of four pins and a set of three pins. In the first set you have VCC and GND that power up the module, and input 1 (IN1) and input 2 (IN2) to control the relays. IN1 controls the first relay (the blue cube at the bottom) and IN2 controls the second relay.



The second set of pins has GND, VCC and JD-VCC pins. Notice that the module has a jumper cap connecting the VCC and JD-VCC pins - the jumper cap is that yellow thing

around those pins, yours may have a different color. This set of three pins can be further used to physically isolate the relays from the ESP32 using the module's built in optocoupler, preventing any damage in case of electrical spikes.

Relay Usage

We're going to use a normally open configuration because we just want to light up the lamp occasionally. For this you use the COM and NO sockets. In a normally open configuration, there is no contact between the COM and NO sockets unless you trigger the relay. The relay is triggered when the input goes below about 2V. This means that if you send a LOW signal from the ESP32, the relay turns on, and if you send a HIGH signal, the relay turns off – so it works with inverted logic:

- Send a LOW signal to turn on the relay
- Send a HIGH signal to turn off the relay

We're not going to use the module's built-in optocoupler, you would need an independent power source to power up the relay module. Don't worry it's still safe to use the relay in this configuration as electrical spikes are very unlikely. You should let the jumper cap connected to the VCC and JD-VCC pins as it is.

Modifying the Code

The relay works with inverted logic, so you need to make some changes to the code to make it work properly. In simple words, you need to replace the "HIGH" in your code with "LOW" and vice-versa.

To initially set the outputs to low, you need to:

```
digitalWrite(output26, HIGH);  
digitalWrite(output27, HIGH);
```

And to turn the GPIOs on and off accordingly to the button pressed:

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

That's it! These are all the modifications you need to do to the code. The complete code is shown below.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi_Web_Server_Relays/WiFi_Web_Server_Relays.ino

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

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

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

// Variable to store the HTTP request
String header;

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

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

// Current time
unsigned long currentTime = millis();
// Previous time
unsigned long previousTime = 0;
// Define timeout time in milliseconds (example: 2000ms = 2s)
const long timeoutTime = 2000;

void setup() {
  Serial.begin(115200);
  // Initialize the output variables as outputs
  pinMode(output26, OUTPUT);
  pinMode(output27, OUTPUT);
  // Set outputs to LOW
  digitalWrite(output26, HIGH);
  digitalWrite(output27, HIGH);

  // 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
    while (client.connected() && currentTime - previousTime <= timeoutTime)
{ // loop while the client's connected
    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();

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

                // 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>");

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

        // Display current state, and ON/OFF buttons for GPIO 26
        client.println("<p>GPIO 26 - State " + output26State + "</p>");
        // If the output26State is off, it displays the ON button
        if (output26State=="off") {
            client.println("<p><a                                     href=\\\"/26/on\\\"><button
class=\\\"button\\\">ON</button></a></p>");
        } else {
            client.println("<p><a href=\\\"/26/off\\\"><button class=\\\"button
button2\\\">OFF</button></a></p>");
        }

        // Display current state, and ON/OFF buttons for GPIO 27
        client.println("<p>GPIO 27 - State " + output27State + "</p>");
        // If the output27State is off, it displays the ON button
        if (output27State=="off") {
            client.println("<p><a                                     href=\\\"/27/on\\\"><button
class=\\\"button\\\">ON</button></a></p>");
        } else {
            client.println("<p><a href=\\\"/27/off\\\"><button class=\\\"button
button2\\\">OFF</button></a></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("");
}
}
}

```

We recommend uploading the code to the ESP32 before assembling the circuit. So, upload the preceding sketch to your ESP32. Make sure you have the right board and COM port selected. Then, open the Serial Monitor at a baud rate of 115200 and save the ESP32 IP address.

Important: don't forget to modify the code with your own SSID and password.

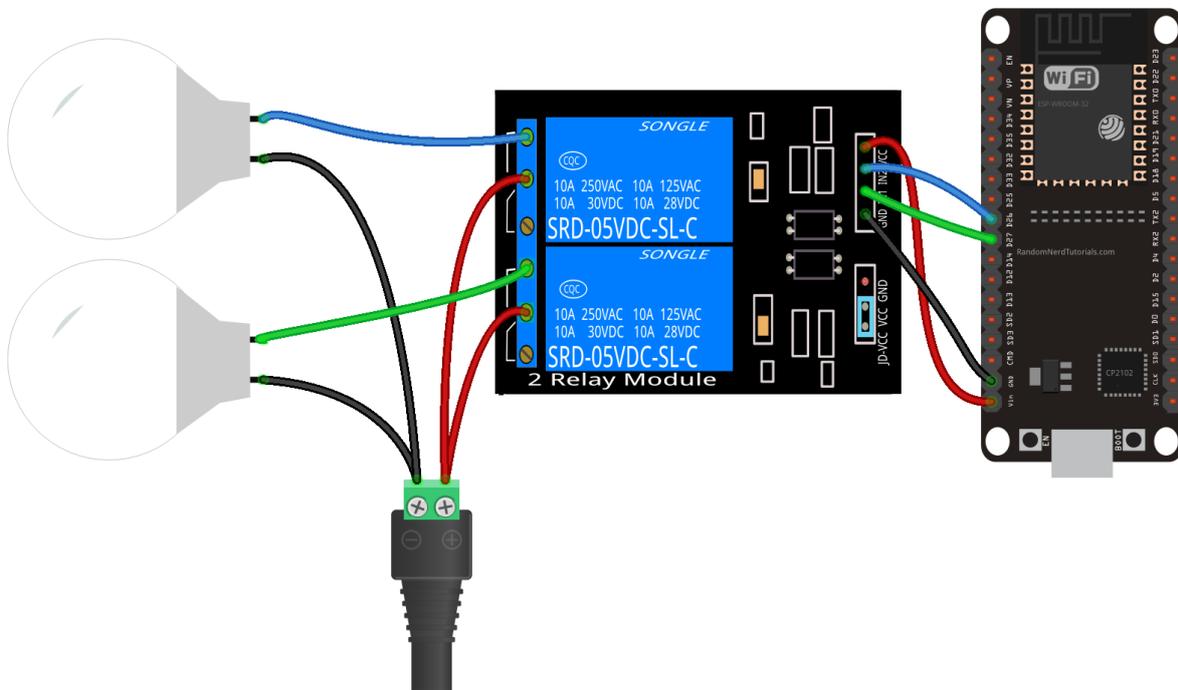
Building the Circuit

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [Relay module](#)
- 12V lamp
- 12V lamp holder
- [Male DC barrel jack 2.1mm](#)
- [12V power adaptor](#)
- [Jumper wires](#)

Wire the circuit by following the instructions below, and use the next schematic diagram as a reference.

You should remove the power from the ESP32 when assembling the circuit.



(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.)

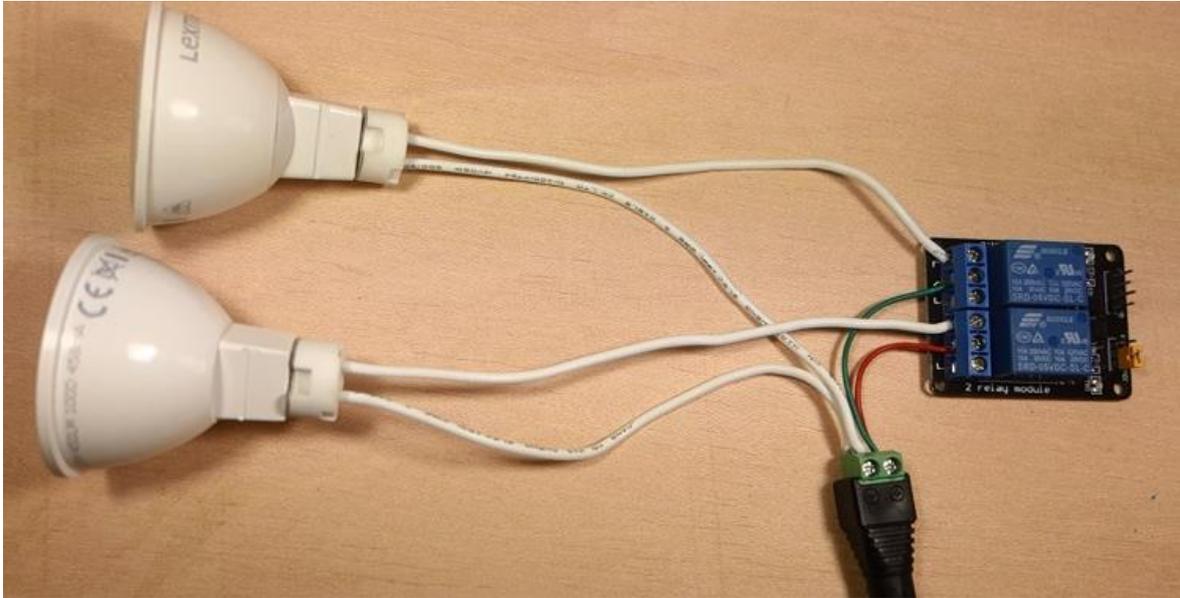
High voltage side

To connect the 12V power adaptor to the relay, it may be handy to have a [DC barrel power jack](#). The DC barrel power jack makes the connections between the adaptor and the relay easier as it connects perfectly on the power adapter terminal.



Connect the high voltage side of the relay as follows:

- Connect one of the lamp holder terminals to the (-) sign terminal on the DC barrel power jack – do this process for both lamps;
- Connect the DC barrel power jack (+) sign terminal to the relay COM sockets;
- Connect the lamp's other terminal to the relay NO socket – do this process for both lamps.



Low voltage side

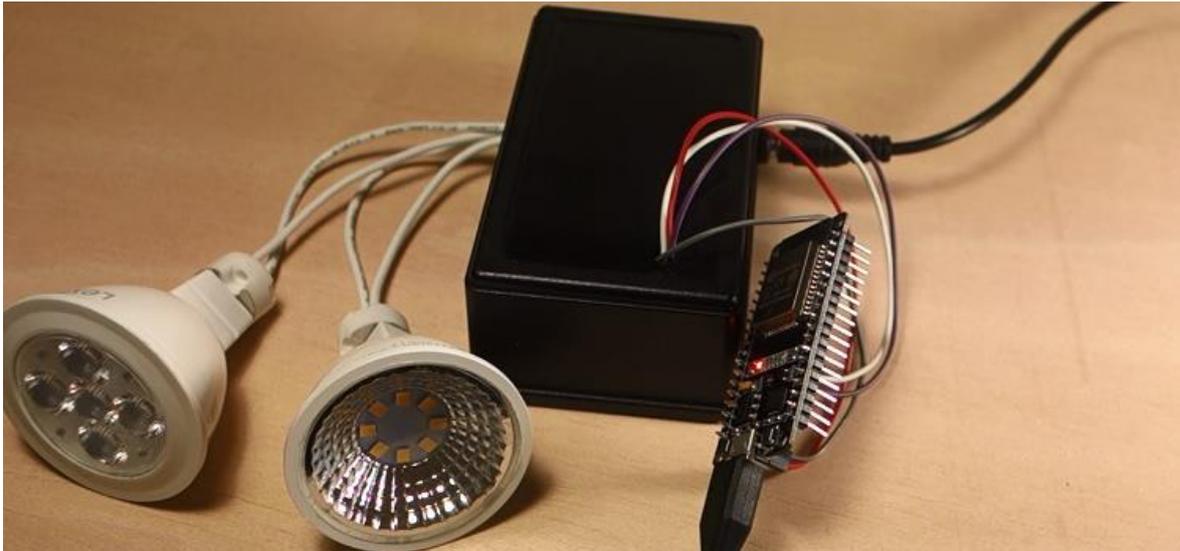
Wire the low voltage side of the relay to the ESP32 as follows:

- Connect the relay IN1 pin to GPIO 27;
- Connect the relay IN2 pin to GPIO 26;
- Connect the relay GND pin to GND;
- Connect the relay VCC pin to the ESP32 VIN pin.

For safety reasons, you may want to place the relay inside a plastic box enclosure as shown below.



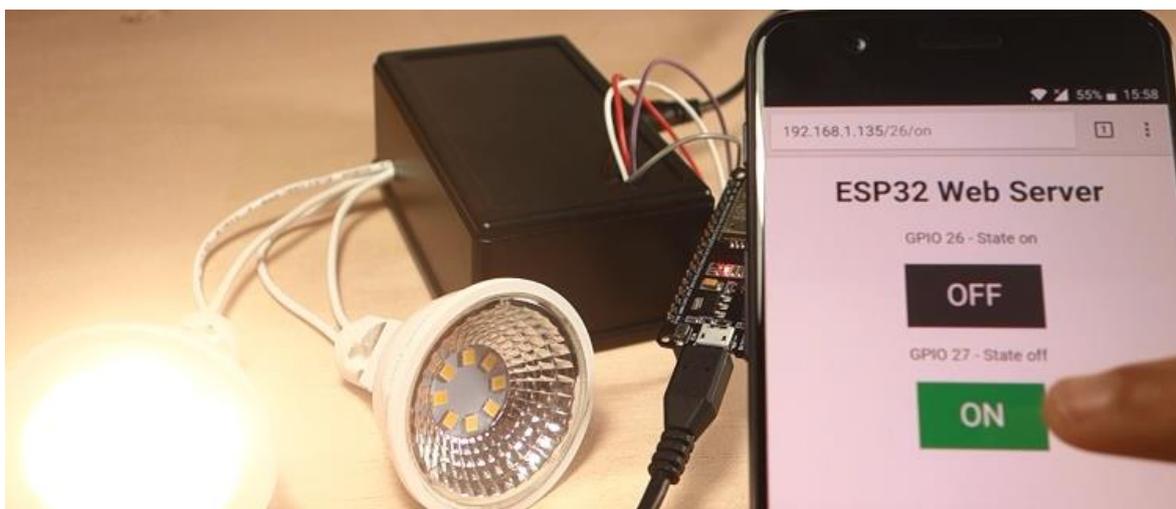
After assembling the circuit, you'll have something as shown in the next figure.



Demonstration

With the circuit ready and the code uploaded to your ESP32. Follow these next instructions:

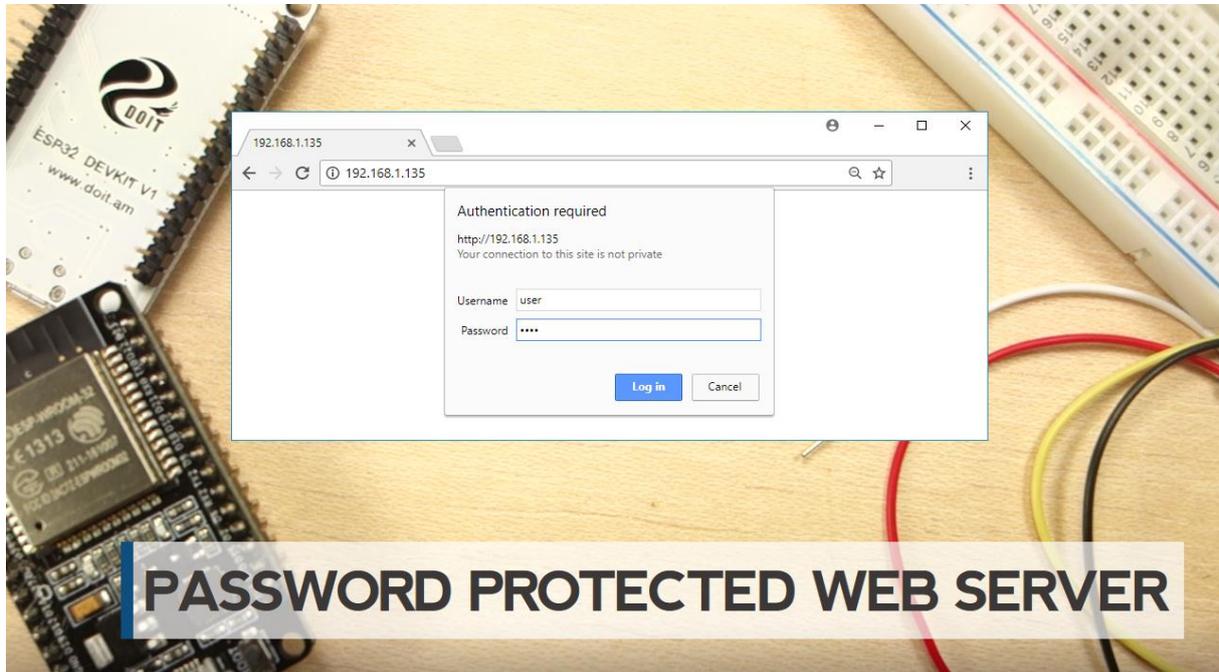
1. Apply power to the ESP32;
2. Connect the 12V power adapter to the DC barrel power jack;
3. Apply power by plugging the 12V power adapter into a wall socket;
4. Go to your browser and type the ESP32 IP address – you should see the web server you've created previously;
5. Click on the buttons to control the lamps.



Wrapping Up

In this Unit you've learned how to replace the LEDs in your project with a relay to control a 12V lamp. Instead of a lamp you may want to control any other electronics appliances. If you are comfortable dealing with mains voltage, you can use the relay to control mains voltage. You can read our tutorial on how to control mains voltage with a relay [here](#).

Unit 6 - Making Your ESP32 Web Server Password Protected



In some projects you might want to keep your ESP32 web server protected. After implementing the feature presented in this Unit, when someone tries to access your web server, they need to enter a valid username and a password. We'll add this feature to the web server created in Unit 2.

Adding Username and Password

The following code protects your web server with username and password. By default, the username is user and the password is pass. I'll show you how to change that in just a moment.

Upload the next sketch and upload it to your ESP32 (after replacing the variables with the SSID and password):

SOURCE CODE

[https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi Web Server Outputs Password Protected/WiFi Web Server Outputs Password Protected.ino](https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi%20Web%20Server%20Outputs%20Password%20Protected/WiFi%20Web%20Server%20Outputs%20Password%20Protected.ino)

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

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

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

```

// Variable to store the HTTP request
String header;

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

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

// Current time
unsigned long currentTime = millis();
// Previous time
unsigned long previousTime = 0;
// Define timeout time in milliseconds (example: 2000ms = 2s)
const long timeoutTime = 2000;

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 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
    while (client.connected() && currentTime - previousTime <= timeoutTime)
{ // loop while the client's connected
      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) {
  // checking if header is valid
  // dXNlcjpwYXNz = 'user:pass' (user:pass) base64 encode
  // Finding the right credential string, then loads web page
  if(header.indexOf("dXNlcjpwYXNz") >= 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();
    // turns the GPIOs on and off
    if (header.indexOf("GET /26/on") >= 0) {
      Serial.println("GPIO 26 on");
      output26State = "on";
      digitalWrite(output26, HIGH);
    } else if (header.indexOf("GET /26/off") >= 0) {
      Serial.println("GPIO 26 off");
      output26State = "off";
      digitalWrite(output26, LOW);
    } else if (header.indexOf("GET /27/on") >= 0) {
      Serial.println("GPIO 27 on");
      output27State = "on";
      digitalWrite(output27, HIGH);
    } else if (header.indexOf("GET /27/off") >= 0) {
      Serial.println("GPIO 27 off");
      output27State = "off";
      digitalWrite(output27, LOW);
    }
    // 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>");
    // Web Page Heading
    client.println("<body><h1>ESP32 Web Server</h1>");
    // Display current state, and ON/OFF buttons for GPIO 26
    client.println("<p>GPIO 26 - State " + output26State + "</p>");
    // If the output26State is off, it displays the ON button
    if (output26State=="off") {
      client.println("<p><a                                href=\"/26/on\"><button
class=\"button\">ON</button></a></p>");
    } else {
      client.println("<p><a                                href=\"/26/off\"><button
class=\"button button2\">OFF</button></a></p>");
    }

    // Display current state, and ON/OFF buttons for GPIO 27
    client.println("<p>GPIO 27 - State " + output27State + "</p>");
    // If the output27State is off, it displays the ON button

```

```

        if (output27State=="off") {
            client.println("<p><a href=\"/27/on\"><button
class=\"button\">ON</button></a></p>");
        } else {
            client.println("<p><a href=\"/27/off\"><button
class=\"button button2\">OFF</button></a></p>");
        }
        client.println("</body></html>");

        // The HTTP response ends with another blank line
        client.println();
        // Break out of the while loop
        break;
    }
    // Wrong user or password, so HTTP request fails...
    else {
        client.println("HTTP/1.1 401 Unauthorized");
        client.println("WWW-Authenticate: Basic realm=\"Secure\"");
        client.println("Content-Type: text/html");
        client.println();
        client.println("<html>Authentication failed</html>");
    }
} 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("");
}
}
}

```

Here are the new sections of code that make your ESP32 web server prompt the username and password form:

```

// checking if header is valid
// dXNlcjpwYXNz = 'user:pass' (user:pass) base64 encode
// Finding the right credential string, then loads web page
if(header.indexOf("dXNlcjpwYXNz") >= 0) {
    // Your web server homepage
}

```

When you enter the right credentials, it loads your web page. However, if you enter the wrong credentials, it prints a text message in your browser saying "Authentication failed".

```

else {
    client.println("HTTP/1.1 401 Unauthorized");
    client.println("WWW-Authenticate: Basic realm=\"Secure\"");
    client.println("Content-Type: text/html");
    client.println();
    client.println("<html>Authentication failed</html>");
}
}

```

Encoding Your Username and Password

At this point if you upload the code written in the preceding section, your username is user and your password is pass. I'm sure you want to change and customize this example with your own credentials.

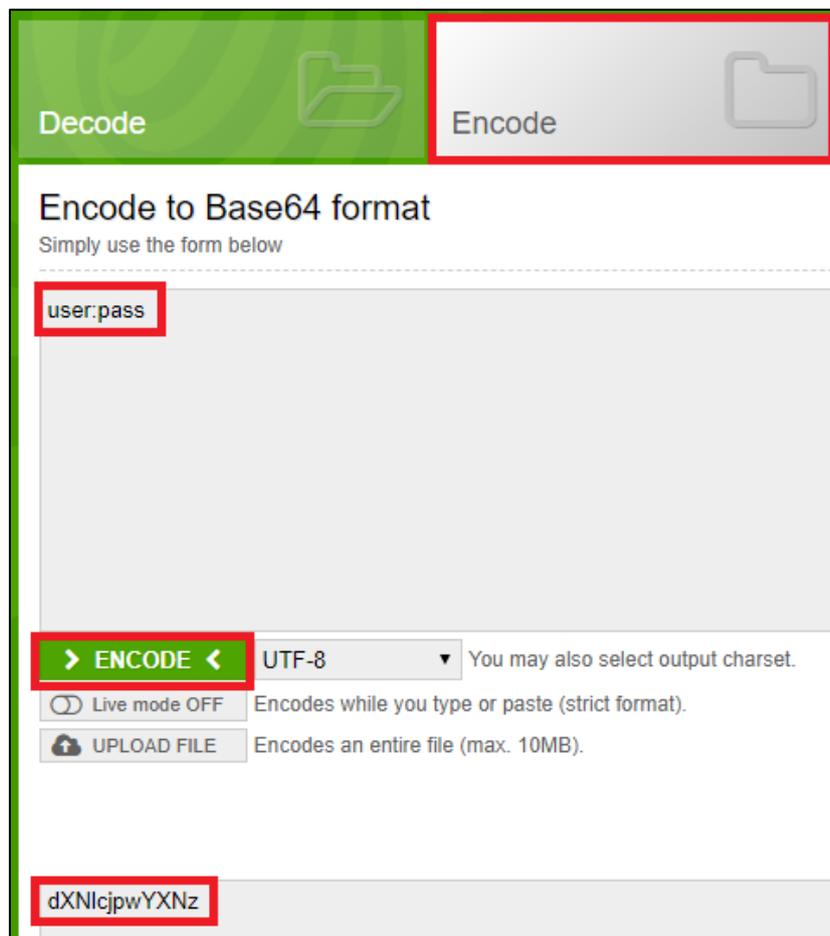
Go to the following website: <https://www.base64encode.org>. In the first field, type the following:

```
your_username:your_password
```

In my case, the username is user and the password is pass:

```
user:pass
```

Note: you need to type the ":" between your username and your password (as illustrated in the figure below):



The screenshot shows the 'Encode to Base64 format' interface. At the top, there are two buttons: 'Decode' (green) and 'Encode' (grey). The 'Encode' button is highlighted with a red box. Below the buttons, the title 'Encode to Base64 format' is displayed, followed by the instruction 'Simply use the form below'. A text input field contains 'user:pass' and is highlighted with a red box. Below the input field, there is a green 'ENCODE' button with left and right arrows, also highlighted with a red box. To the right of the button is a dropdown menu set to 'UTF-8' and the text 'You may also select output charset.'. Below the dropdown are two options: 'Live mode OFF' (checked) and 'UPLOAD FILE'. At the bottom, the output field contains 'dXNlcjpwYXNz' and is highlighted with a red box.

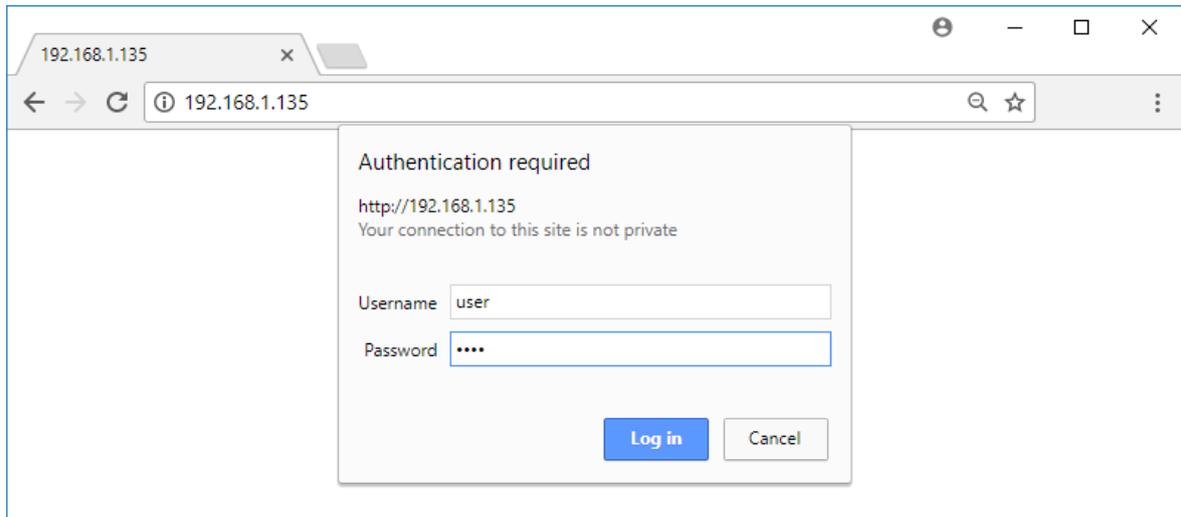
Then, press the green "Encode" button to generate your base64 encoded string. In my example is **dXNlcjpwYXNz**. Copy your encoded string and replace it in your sketch in the following exact if statement:

```
if(header.indexOf("dXNlcjpwYXNz") >= 0) {
```

After adding your string, SSID and password, you can upload the new sketch to your ESP32.

Testing the Web Server

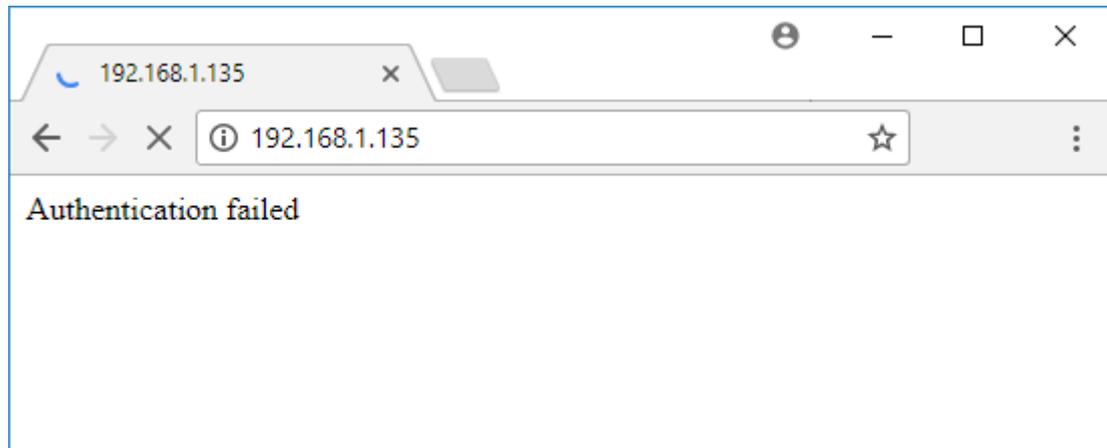
Now, when you try to access your ESP IP address, you'll be required to enter your username and password. Then, press the "Log in" button:



You'll be able to see your ESP32 web server:



If you enter the wrong password, you'll see an error message:

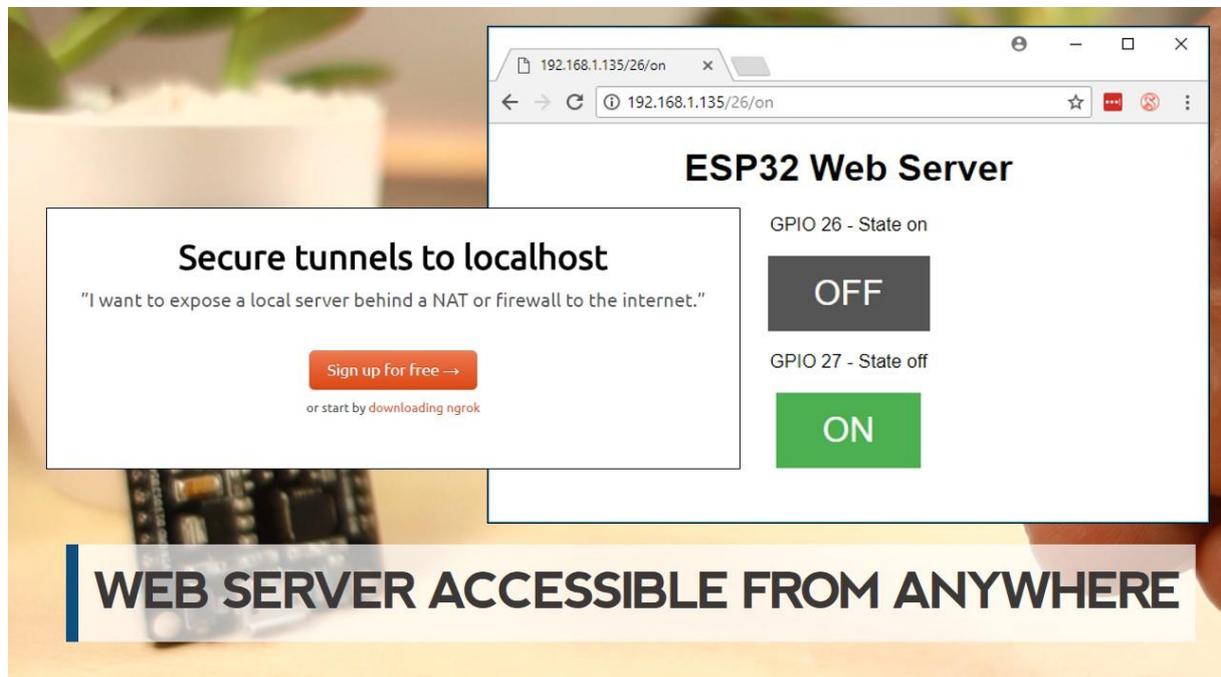


Wrapping Up

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.

In the next Unit you'll learn how to make your web server accessible from anywhere (outside the local network).

Unit 7 - 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 ESP web server is only accessible when you are connected to your local network.

Luckily, in just a few minutes you'll be able to control and monitor your home from anywhere.

Note: This method requires having a computer turned on, but you can run this software on a low-cost/low-power computer such as the Raspberry Pi.

Preparing Your ESP32

In order to make your ESP32 web server accessible from anywhere, you need to change the default port number in the Arduino code.

If you've followed the previous Unit called "**Making Your ESP32 Web Server Password Protected**" or any of the previous Units that show how to build a web server, we've always configured the port number to 80.

To make it work for this example, you need to modify port 80:

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

To port 8888 (or any other port number, for some reason ngrok doesn't work if you're using port 80):

```
// Set web server port number to 8888
```

```
WiFiServer server(8888);
```

Here's the code that you need to upload to your ESP32 (add your SSID and password):

SOURCE CODE

[https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi Web Server Outputs_PP_Port8888/WiFi Web Server Outputs_PP_Port8888.ino](https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi%20Web%20Server%20Outputs_PP_Port8888/WiFi%20Web%20Server%20Outputs_PP_Port8888.ino)

```
/*  
  Rui Santos  
  Complete project details at http://randomnerdtutorials.com  
  */  
  
// Load Wi-Fi library  
#include <WiFi.h>  
  
// Replace with your network credentials  
const char* ssid = "REPLACE_WITH_YOUR_SSID";  
const char* password = "REPLACE_WITH_YOUR_PASSWORD";  
  
// Set web server port number to 8888  
WiFiServer server(8888);  
  
// Variable to store the HTTP request  
String header;  
  
// Auxiliar variables to store the current output state  
String output26State = "off";  
String output27State = "off";  
  
// Assign output variables to GPIO pins  
const int output26 = 26;  
const int output27 = 27;  
  
// Current time  
unsigned long currentTime = millis();  
// Previous time  
unsigned long previousTime = 0;  
// Define timeout time in milliseconds (example: 2000ms = 2s)  
const long timeoutTime = 2000;  
  
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 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
    while (client.connected() && currentTime - previousTime <= timeoutTime)
{ // loop while the client's connected
      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) {
            // checking if header is valid
            // dXNlcjpwYXNz = 'user:pass' (user:pass) base64 encode
            // Finding the right credential string, then loads web page
            if(header.indexOf("dXNlcjpwYXNz") >= 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();
              // turns the GPIOs on and off
              if (header.indexOf("GET /26/on") >= 0) {
                Serial.println("GPIO 26 on");
                output26State = "on";
                digitalWrite(output26, HIGH);
              } else if (header.indexOf("GET /26/off") >= 0) {
                Serial.println("GPIO 26 off");
                output26State = "off";
                digitalWrite(output26, LOW);
              } else if (header.indexOf("GET /27/on") >= 0) {
                Serial.println("GPIO 27 on");
                output27State = "on";
                digitalWrite(output27, HIGH);
              } else if (header.indexOf("GET /27/off") >= 0) {
                Serial.println("GPIO 27 off");
                output27State = "off";
                digitalWrite(output27, LOW);
              }
              // 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>");

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

        // Display current state, and ON/OFF buttons for GPIO 26
        client.println("<p>GPIO 26 - State " + output26State + "</p>");
        // If the output26State is off, it displays the ON button
        if (output26State=="off") {
            client.println("<p><a href=\"/26/on\"><button
class=\"button\">ON</button></a></p>");
        } else {
            client.println("<p><a href=\"/26/off\"><button
class=\"button button2\">OFF</button></a></p>");
        }

        // Display current state, and ON/OFF buttons for GPIO 27
        client.println("<p>GPIO 27 - State " + output27State + "</p>");
        // If the output27State is off, it displays the ON button
        if (output27State=="off") {
            client.println("<p><a href=\"/27/on\"><button
class=\"button\">ON</button></a></p>");
        } else {
            client.println("<p><a href=\"/27/off\"><button
class=\"button button2\">OFF</button></a></p>");
        }
        client.println("</body></html>");

        // The HTTP response ends with another blank line
        client.println();
        // Break out of the while loop
        break;
    }
    // Wrong user or password, so HTTP request fails...
    else {
        client.println("HTTP/1.1 401 Unauthorized");
        client.println("WWW-Authenticate: Basic realm=\"Secure\"");
        client.println("Content-Type: text/html");
        client.println();
        client.println("<html>Authentication failed</html>");
    }
    } 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("");
}
}

```

After, uploading the code go to your browser and type:

```
http://YOUR_ESP_IP_ADDRESS:8888
```

In my case, it's:

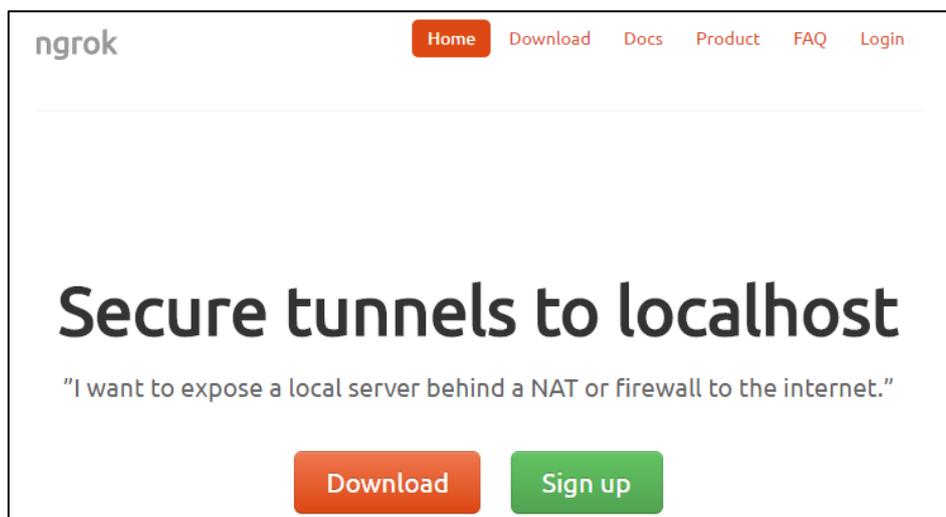
```
http://192.168.1.70:8888
```

Your web server should load, otherwise make sure you've changed the port number or that you're using the right IP address.

Introducing ngrok

You are going to use a free service called **ngrok** to create a tunnel to your ESP32.

Go to <https://ngrok.com> to create your account. Click the green "Sign up" button:

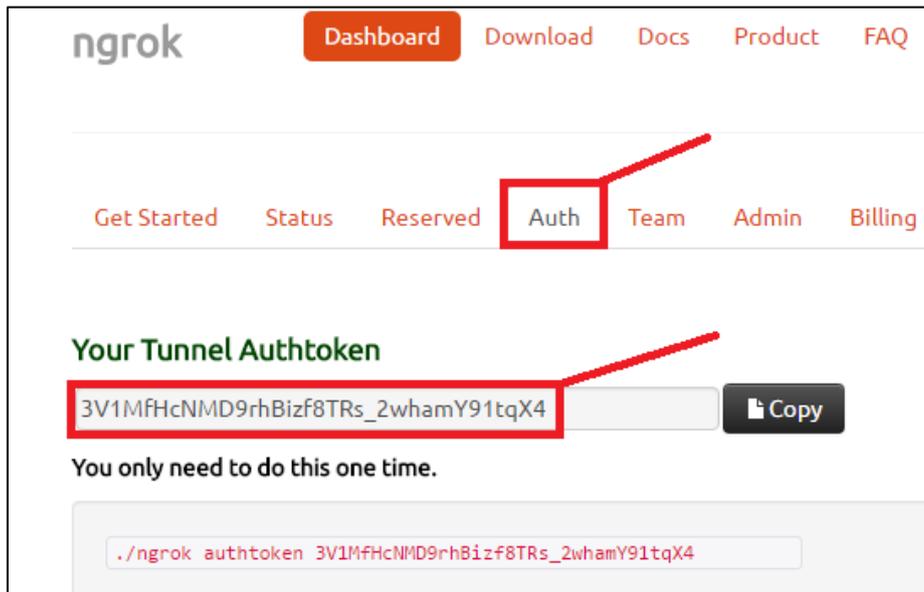


Enter your details in the fields.

The image shows the 'Sign up' form on the ngrok website. The form is titled 'Sign up' in a large, bold font. It contains four input fields: 'Your Name' with a placeholder 'Full Name', 'Your Email' with a placeholder 'Email address', 'Confirm Email' with a placeholder 'Email address', and 'Password' with a placeholder 'Password'. At the bottom of the form is a green 'Sign up' button.

After creating your account, login and go to the "**Auth**" tab to find Your Tunnel Authtoken.

Copy your unique **Your Tunnel Authtoken** to a safe place (you'll need it later in this Unit).



My Tunnel Authtoken is:

```
3V1MfHcNMD9rhBizf8TRs_2whamY91tqX4
```

Installing ngrok

Then, go to the navigation bar and select the “**Download**” tab:



Choose your operating system and download the **ngrok** software.

Download and Installation

ngrok is easy to install. Download a single binary with *zero run-time dependencies* for any major platform. Unzip it and then run it from the command line.

Step 1: Download ngrok

Mac OS X 64-Bit	Download
Windows 64-Bit	Download
Linux 64-Bit	Download
Linux ARM	Download
FreeBSD 64-Bit	Download

[32-bit platforms](#)

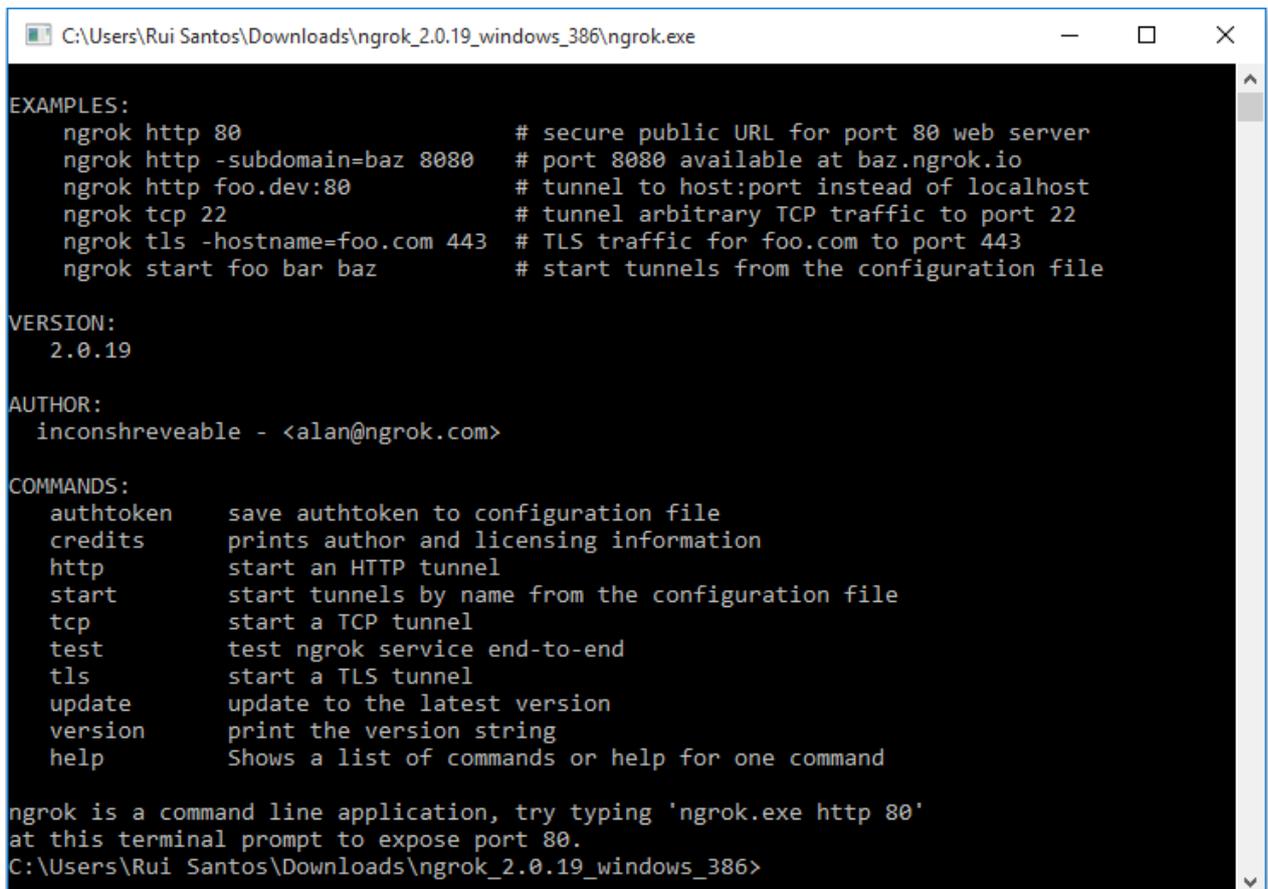
Note: if you're on Linux, open the terminal and type `./ngrok -help` to open the ngrok application.

Running ngrok

Unzip the folder that you have just downloaded and open the ngrok application. Double-click the ngrok application.



You should see a window similar to the one below:



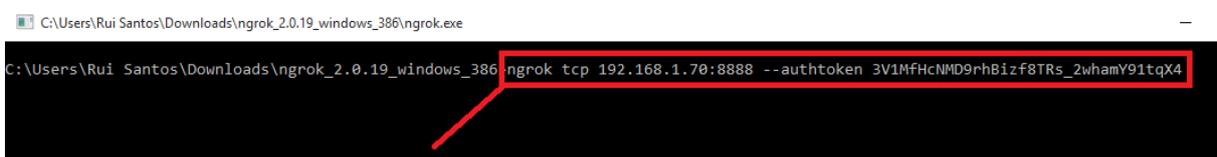
Now, in your terminal window, enter the following command and replace the red text with your own IP address and ngrok's tunnel Authtoken:

```
ngrok tcp YOUR_ESP_IP_ADDRESS:8888 --authtoken YOUR_TUNNEL_AUTHTOKEN
```

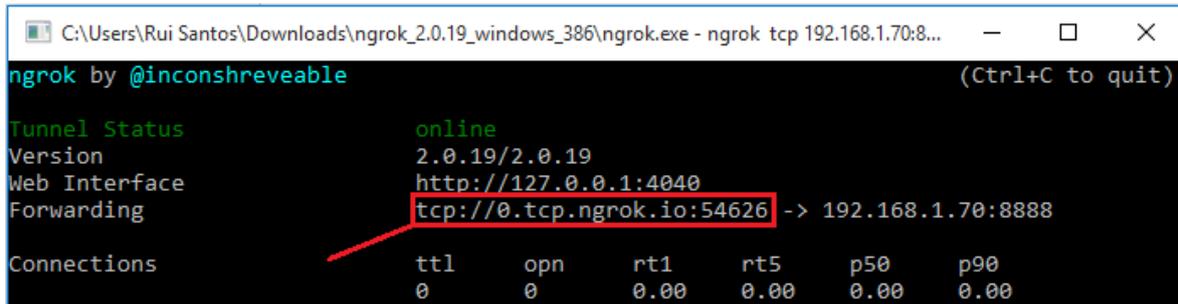
This is how it looks like for me:

```
ngrok tcp 192.168.1.70:8888 --authtoken 3V1MfHcNMD9rhBizf8TRs_2whamY91tqX4
```

The following figure shows how it should look like in the terminal window. Press Enter to run this command.



If everything runs smoothly, you should notice that your Tunnel is online and a URL should be in your terminal.



```
C:\Users\Rui Santos\Downloads\ngrok_2.0.19_windows_386\ngrok.exe - ngrok tcp 192.168.1.70:8...
ngrok by @inconshreveable (Ctrl+C to quit)
Tunnel Status      online
Version            2.0.19/2.0.19
Web Interface      http://127.0.0.1:4040
Forwarding         tcp://0.tcp.ngrok.io:54626 -> 192.168.1.70:8888
Connections
  ttl    opn    rt1    rt5    p50    p90
   0     0     0.00  0.00  0.00  0.00
```

Accessing your web server from anywhere

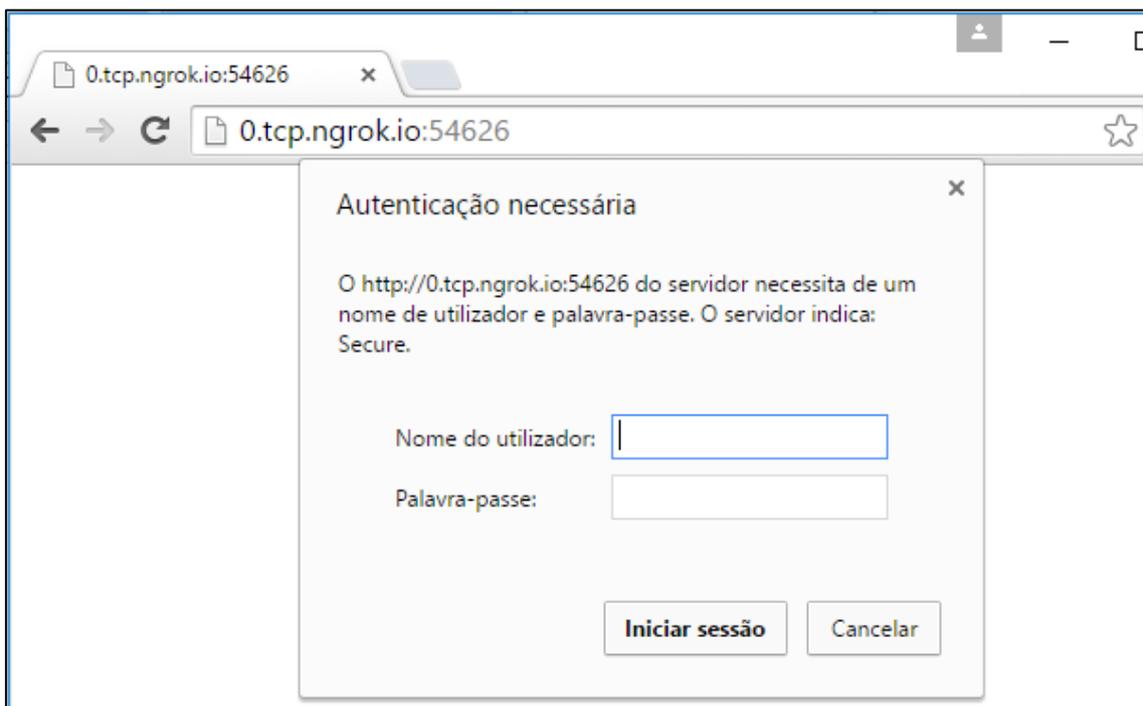
Now you can access your web server from anywhere in the world by typing your unique URL (in my case **http://0.tcp.ngrok.io:54626/**) in a browser.

Note: even though it's a tcp connection you type http in your web browser.

Important: you need to let your computer on and running ngrok in order to maintain the tunnel online.

Troubleshooting: if you go to your ngrok.io URL and nothing happens, open your ESP IP address in your browser to check if your web server is still running. If it's still running make sure you've entered the right IP address and authtoken on the ngrok command executed earlier.

You'll always be asked to enter your username and password to open your web server.

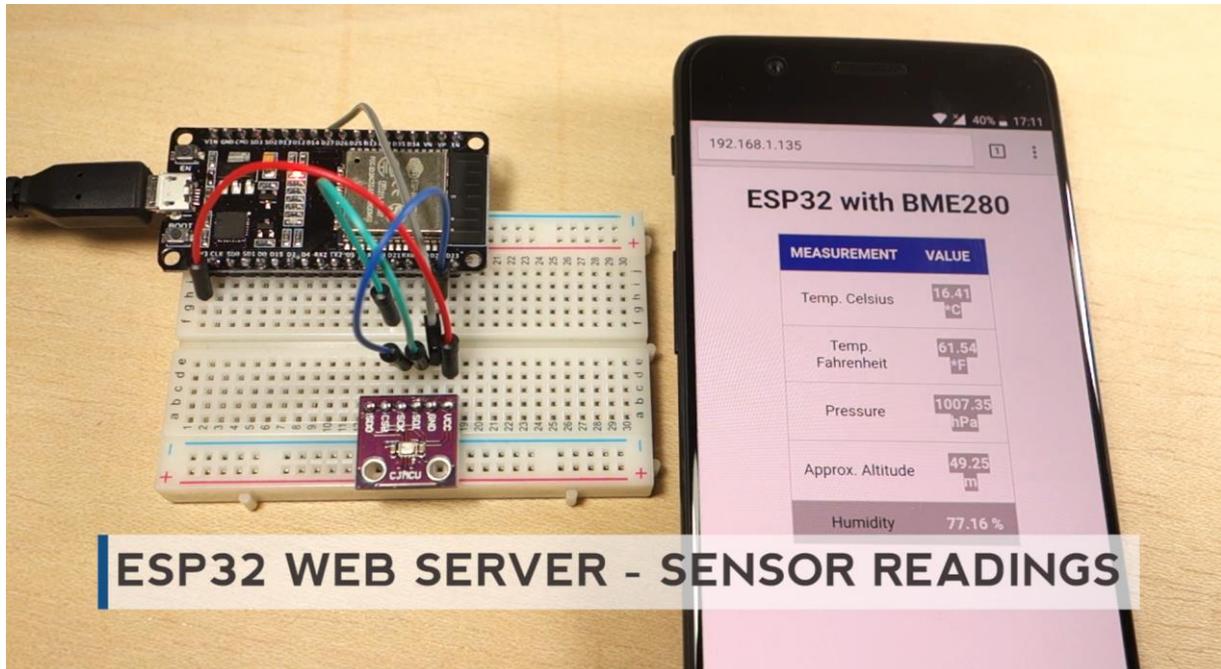


Now, you have full control over your ESP32 from anywhere!

Wrapping Up

In this Unit you've learned how to make your ESP32 accessible from anywhere. This is a very interesting feature as it allows you to control and monitor your house anytime anywhere. As always, you can apply this concept to any of your other web servers.

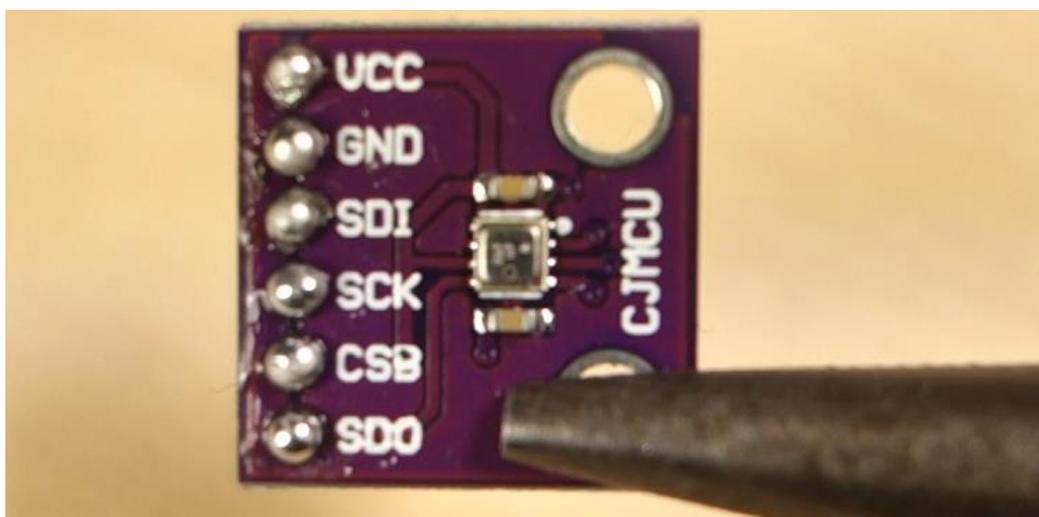
Unit 8 - ESP32 Web Server – Display Sensor Readings



In this Unit you're going to 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.



The sensor can communicate using either SPI or I2C communication protocols (there are modules of this sensor that just communicate with I2C, these just come with four pins).

To use SPI communication protocol, you use the following pins:

- SCK – this is the SPI Clock pin
- SDO – MISO
- SDI – MOSI
- CS – Chip Select

To use I2C communication protocol, the sensor uses the following pins:

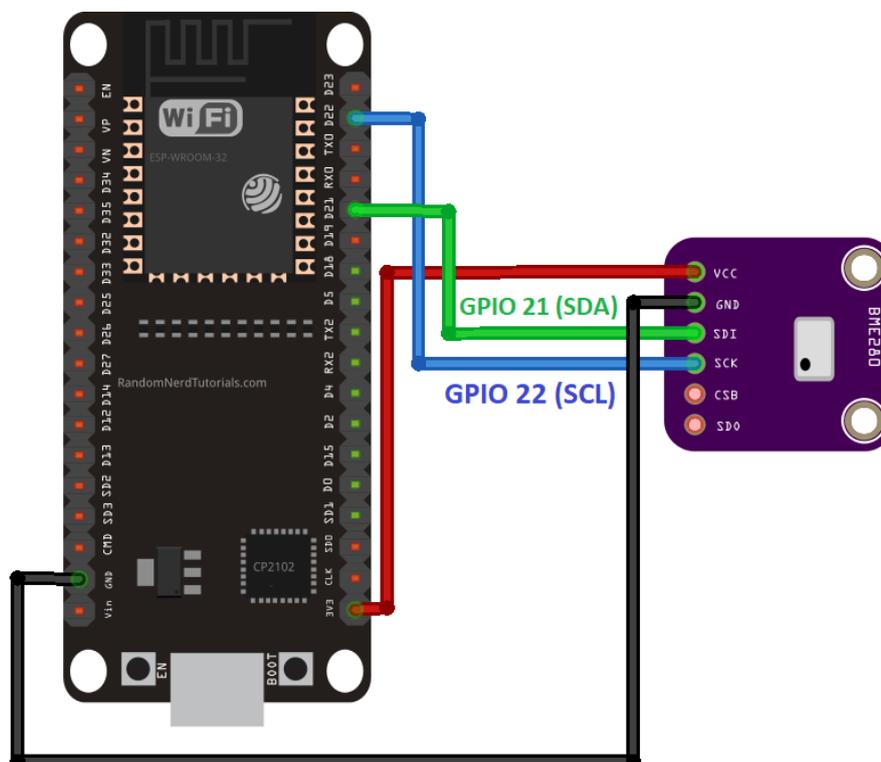
- SCK – this is also the SCL pin
- SDI – this is also the SDA pin

Schematic

We're going to use I2C communication with the BME280 sensor module. For that, wire the sensor to the ESP32 SDA and SCL pins, as shown in the following schematic diagram.

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [BME280 sensor module](#)
- [Breadboard](#)
- [Jumper wires](#)
- [Schematics](#)



(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.)

Installing the BME280 library

To take 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) [Click here to download](#) the Adafruit-BME280 library. You should have a .zip folder in your Downloads folder
- 2) Unzip the .zip folder and you should get Adafruit-BME280-Library-master folder
- 3) Rename your folder from ~~Adafruit-BME280-Library-master~~ to Adafruit_BME280_Library
- 4) Move the Adafruit_BME280_Library folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Alternatively, you can go to **Sketch ▶ Include Library ▶ Manage Libraries** and type **"adafruit bme280"** to search for the library. Then, click install.

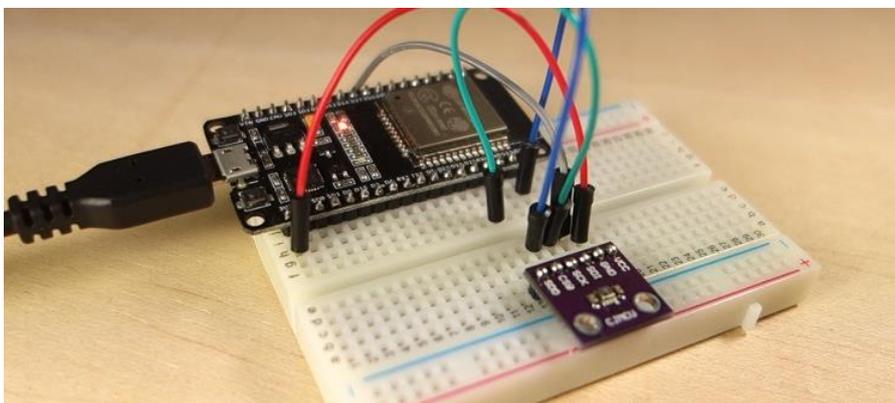
Installing the Adafruit_Sensor library

To use the BME280 library, you also need to install the [Adafruit_Sensor library](#). Follow the next steps to install the library in your Arduino IDE:

- 1) [Click here to download](#) the Adafruit_Sensor library. You should have a .zip folder in your Downloads folder
- 2) Unzip the .zip folder and you should get Adafruit_Sensor-master folder
- 3) Rename your folder from ~~Adafruit_Sensor-master~~ to Adafruit_Sensor
- 4) Move the Adafruit_Sensor folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Reading Temperature, Humidity, and Pressure

To get familiar with the BME280 sensor, we're going to use an example sketch from the library to see how to read temperature, humidity, and pressure.



After installing the BME280 library, and the Adafruit_Sensor library, open the Arduino IDE and, go to **File ▶ Examples ▶ Adafruit BME280 library ▶ bme280 test**.

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/bme280test/bme280test.ino>

```
/******  
  Complete project details at http://randomnerdtutorials.com  
*****/  
  
#include <Wire.h>  
#include <Adafruit_Sensor.h>  
#include <Adafruit_BME280.h>  
  
/*#include <SPI.h>  
#define BME_SCK 18  
#define BME_MISO 19  
#define BME_MOSI 23  
#define BME_CS 5*/  
  
#define SEALEVELPRESSURE_HPA (1013.25)  
  
Adafruit_BME280 bme; // I2C  
//Adafruit_BME280 bme(BME_CS); // hardware SPI  
//Adafruit_BME280 bme(BME_CS, BME_MOSI, BME_MISO, BME_SCK); // software SPI  
  
unsigned long delayTime;  
  
void setup() {  
  Serial.begin(9600);  
  Serial.println(F("BME280 test"));  
  
  bool status;  
  
  // default settings  
  // (you can also pass in a Wire library object like &Wire2)  
  status = bme.begin(0x76);  
  if (!status) {  
    Serial.println("Could not find a valid BME280 sensor, check  
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>
```

SPI communication

As we're going to use I2C communication you can comment the following lines:

```
/*#include <SPI.h>
#define BME_SCK 18
#define BME_MISO 19
#define BME_MOSI 23
#define BME_CS 5*/
```

Note: if you're using SPI communication, you need to change the pin definition to use the ESP32 GPIOs. For SPI communication on the ESP32 you can use either the HSPI or VSPI pins, as shown in the following table.

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

Sea level pressure

A variable called SEALEVELPRESSURE_HPA is created.

```
#define SEALEVELPRESSURE_HPA (1013.25)
```

This saves the pressure at the sea level in hectopascal (is equivalent to milibar). This variable is used to estimate the altitude for a given pressure by comparing it with the sea level pressure. This examples 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`.

```
Adafruit_BME280 bme; // I2C
```

If you would like to use SPI, you need to comment this previous line and uncomment one of the following lines depending on whether you're using hardware or software SPI.

```
//Adafruit_BME280 bme(BME_CS); // hardware SPI  
//Adafruit_BME280 bme(BME_CS, BME_MOSI, BME_MISO, BME_SCK); // software SPI
```

setup()

In the `setup()` you start a serial communication

```
Serial.begin(9600);
```

And the sensor is initialized:

```
status = bme.begin(0x76);  
if (!status) {  
    Serial.println("Could not find a valid BME280 sensor, check wiring!");  
    while (1);  
}
```

Printing values

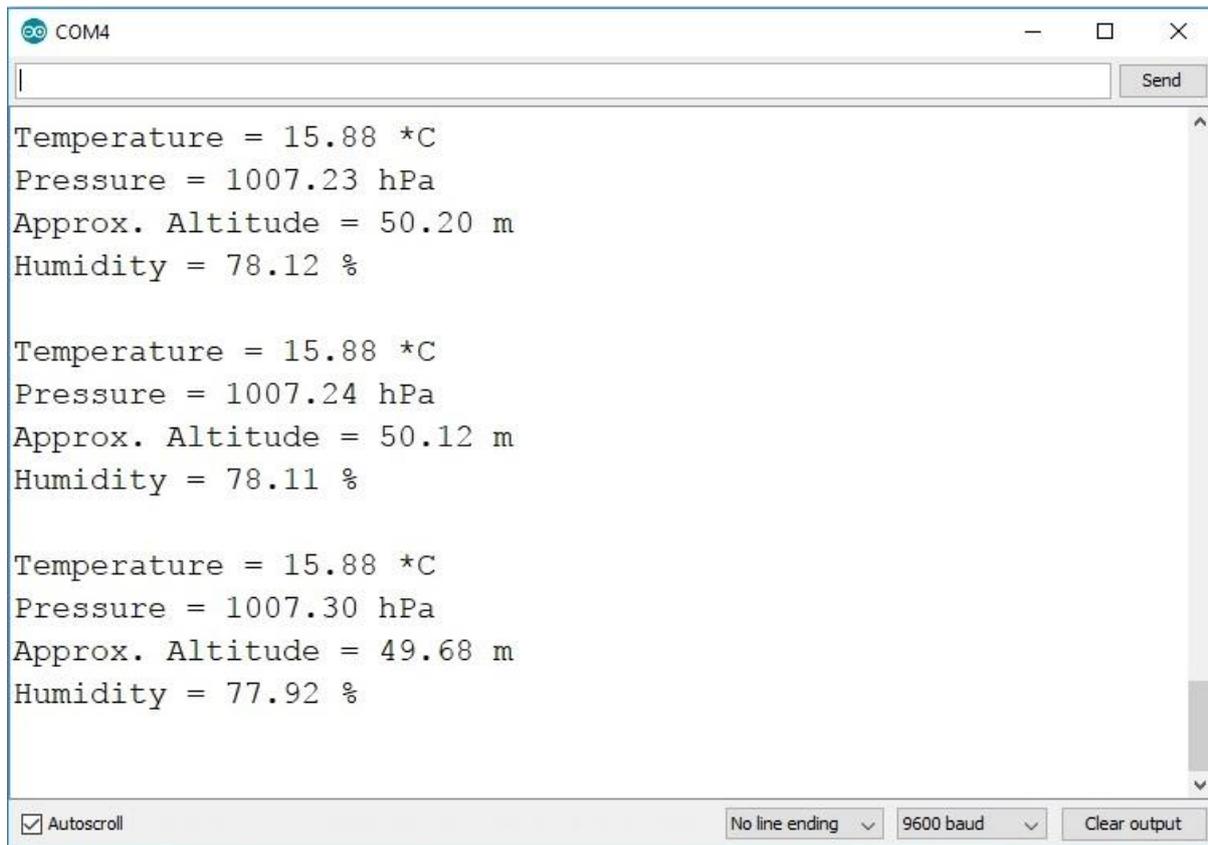
In the `loop()`, the `printValues()` function reads the values from the BME280 and prints the results in the Serial Monitor.

```
void loop() {  
    printValues();  
    delay(delayTime);  
}
```

Reading temperature, humidity, pressure, and estimate 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 9600. You should see the readings displayed on the Serial Monitor.



Creating a Table in HTML

As you've seen in the beginning of the video, we're displaying the readings in a web page with table served by the ESP32. 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>
```

```

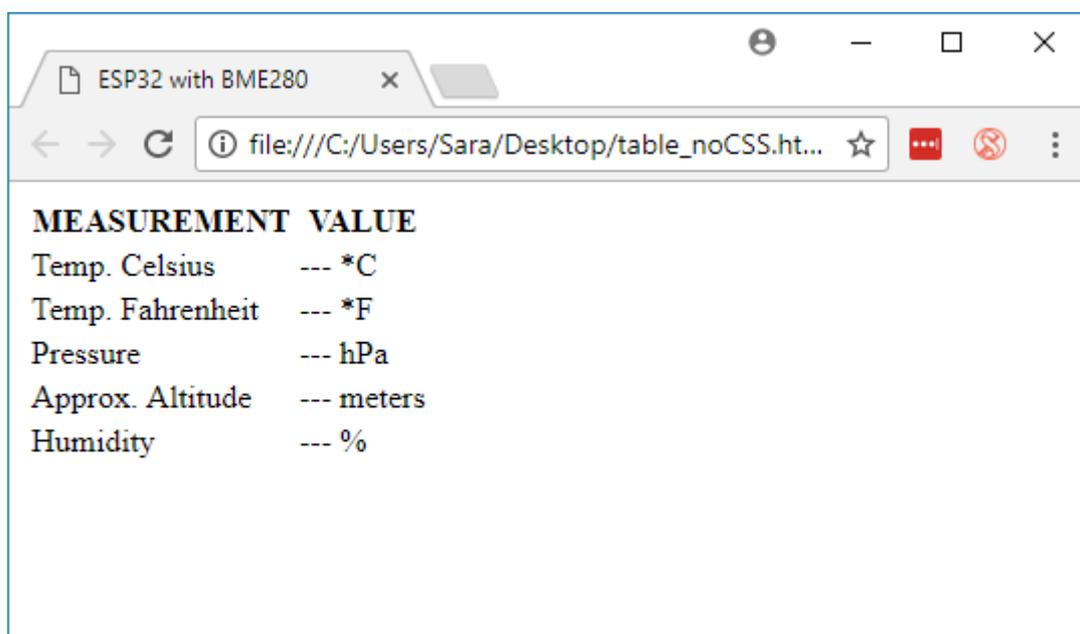
<td>Pressure</td>
<td>--- hPa</td>
</tr>
<tr>
<td>Approx. Altitude</td>
<td>--- meters</td></tr>
<tr>
<td>Humidity</td>
<td>--- %</td>
</tr>
</table>

```

We create the header of the table with a cell called MEASUREMENT, and another called 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 BME 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 with your own taste. You may find this link useful: [CSS Styling Tables](#).

Creating the Web Server

Now that you know how to take readings from the sensor, and how to build a table to display the results, it's time to build the web server. If you've followed the previous Units, you should be familiar with the majority of the code.

Copy the following code to your Arduino IDE. Don't upload it yet. First, you need to include your SSID and password.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi_Web_Server_Sensor_Readings/WiFi_Web_Server_Sensor_Readings.ino

```
// Load Wi-Fi library
#include <WiFi.h>
#include <Wire.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

//uncomment the following lines if you're using SPI
/*#include <SPI.h>
#define BME_SCK 18
#define BME_MISO 19
#define BME_MOSI 23
#define BME_CS 5*/

#define SEALEVELPRESSURE_HPA (1013.25)

Adafruit_BME280 bme; // I2C
//Adafruit_BME280 bme(BME_CS); // hardware SPI
//Adafruit_BME280 bme(BME_CS, BME_MOSI, BME_MISO, BME_SCK); // software SPI

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

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

// Variable to store the HTTP request
String header;

// 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() {
  Serial.begin(115200);
  bool status;

  // default settings
  // (you can also pass in a Wire library object like &Wire2)
  //status = bme.begin();
  if (!bme.begin(0x76)) {
    Serial.println("Could not find a valid BME280 sensor, check wiring!");
    while (1);
  }

  // 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
    while (client.connected() && currentTime - previousTime <= timeoutTime)
    { // loop while the client's connected
      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 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>ESP32 with
                BME280</h1>");
                client.println("<table><tr><th>MEASUREMENT</th><th>VALUE</th></
                tr>");

```

```

        client.println("<tr><td>Temp.           Celsius</td><td><span
class=\"sensor\\>");
        client.println(bme.readTemperature());
        client.println(" *C</span></td></tr>");
        client.println("<tr><td>Temp.           Fahrenheit</td><td><span
class=\"sensor\\>");
        client.println(1.8 * bme.readTemperature() + 32);
        client.println(" *F</span></td></tr>");
        client.println("<tr><td>Pressure</td><td><span
class=\"sensor\\>");
        client.println(bme.readPressure() / 100.0F);
        client.println(" hPa</span></td></tr>");
        client.println("<tr><td>Approx.           Altitude</td><td><span
class=\"sensor\\>");
        client.println(bme.readAltitude(SEALEVELPRESSURE_HPA));
        client.println(" m</span></td></tr>");
        client.println("<tr><td>Humidity</td><td><span
class=\"sensor\\>");
        client.println(bme.readHumidity());
        client.println(" %</span></td></tr>");
        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("");
}
}

```

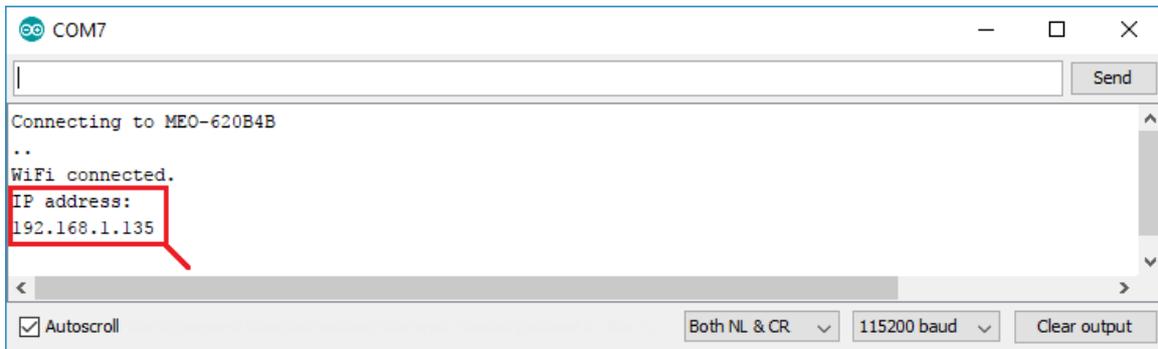
Modify the following lines to include your SSID and password.

```

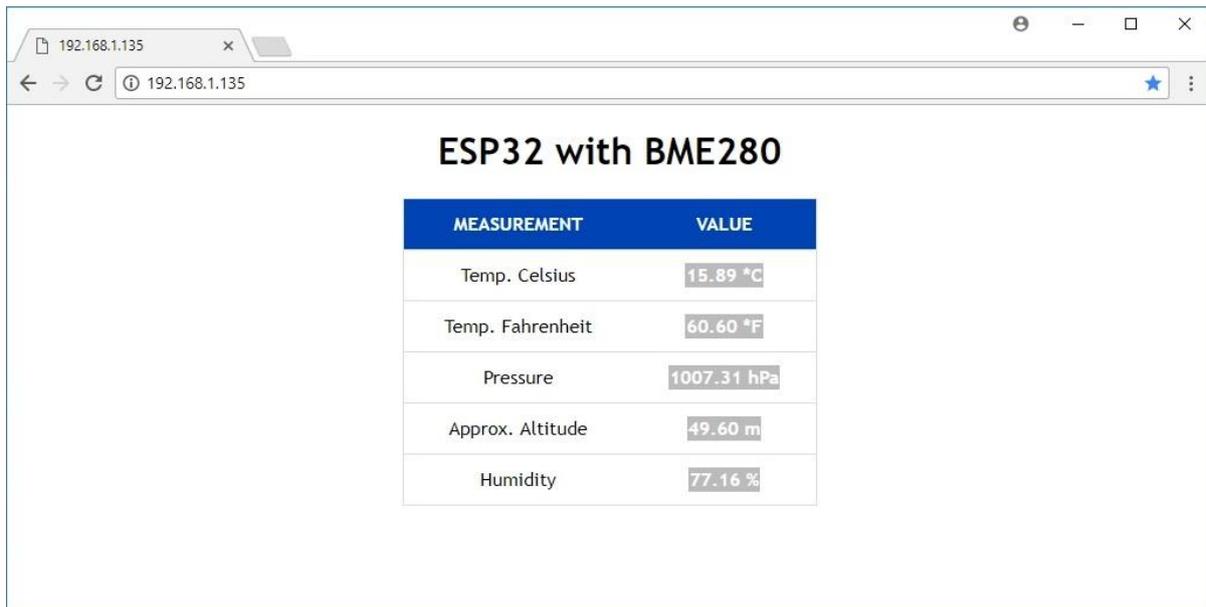
const char* ssid      = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

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.



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 with the sketch used in Unit 2. You must be familiar with most part of the code. First, you include the WiFi library and the needed libraries to read from the BME280 sensor.

```
#include <WiFi.h>
#include <Wire.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
```

The next 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, you set your web server to port 80.

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

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

```
String header;
```

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

loop()

In the `loop()`, we program what happens when a new client establishes a connection with the web server. The ESP is always listening for incoming clients with this line:

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

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

```
if (client) { // If a new client connects,
    Serial.println("New Client.");
    String currentLine = ""; //String to hold incoming data from 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();
                }
            }
        }
    }
}
```

Displaying the HTML web page

The next thing you need to is sending a response to the client with the HTML text to build the web page.

Note: you can read Unit 3 and Unit 4 of this Module to learn about HTML and CSS basics.

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

The following code snippet sends the web page to display the sensor readings in a table.

```
// 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:,\">");

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>ESP32 with BME280</h1>");
client.println("<table><tr><th>MEASUREMENT</th><th>VALUE</th></tr>");
client.println("<tr><td>Temp. Celsius</td><td><span class=\"sensor\">");
client.println(bme.readTemperature());
client.println(" *C</span></td></tr>");
client.println("<tr><td>Temp. Fahrenheit</td><td><span class=\"sensor\">");
client.println(1.8 * bme.readTemperature() + 32);
```

```

client.println(" *F</span></td></tr>");
client.println("<tr><td>Pressure</td><td><span class=\"sensor\">");
client.println(bme.readPressure() / 100.0F);
client.println(" hPa</span></td></tr>");
client.println("<tr><td>Approx. Altitude</td><td><span class=\"sensor\">");
client.println(bme.readAltitude(SEALEVELPRESSURE_HPA));
client.println(" m</span></td></tr>");
client.println("<tr><td>Humidity</td><td><span class=\"sensor\">");
client.println(bme.readHumidity());
client.println(" %</span></td></tr>");
client.println("</body></html>");

```

Displaying the Sensor Readings

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:

```

client.println("<tr><td>Temp. Celsius</td><td><span class=\"sensor\">");
client.println(bme.readTemperature());
client.println(" *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 is displaying the temperature readings in both Celsius degrees and Fahrenheit. You can comment the following three lines, if you only want to display the temperature only in Fahrenheit degrees.

```

/*client.println("<tr><td>Temp. Celsius</td><td><span class=\"sensor\">");
client.println(bme.readTemperature());
client.println(" *C</span></td></tr>");*/

```

Closing the Connection

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

```

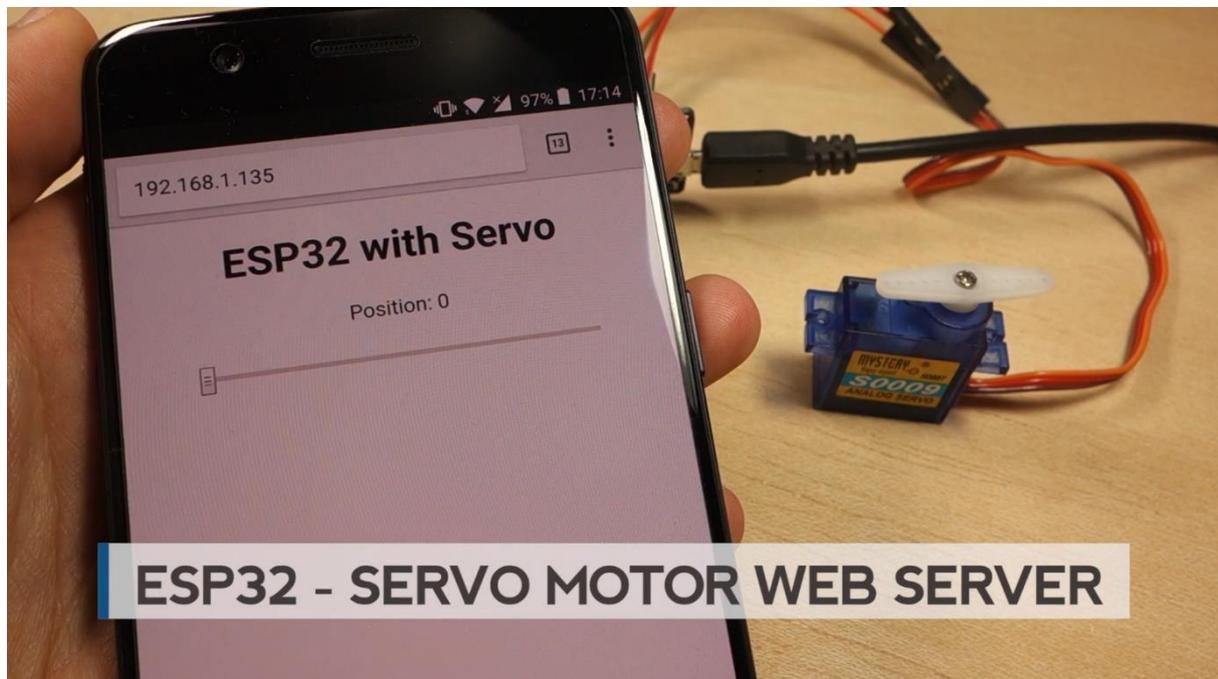
// Clear the header variable
header = "";
// Close the connection
client.stop();

```

Wrapping Up

In summary, in this project you've learned how to read temperature, humidity, 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.

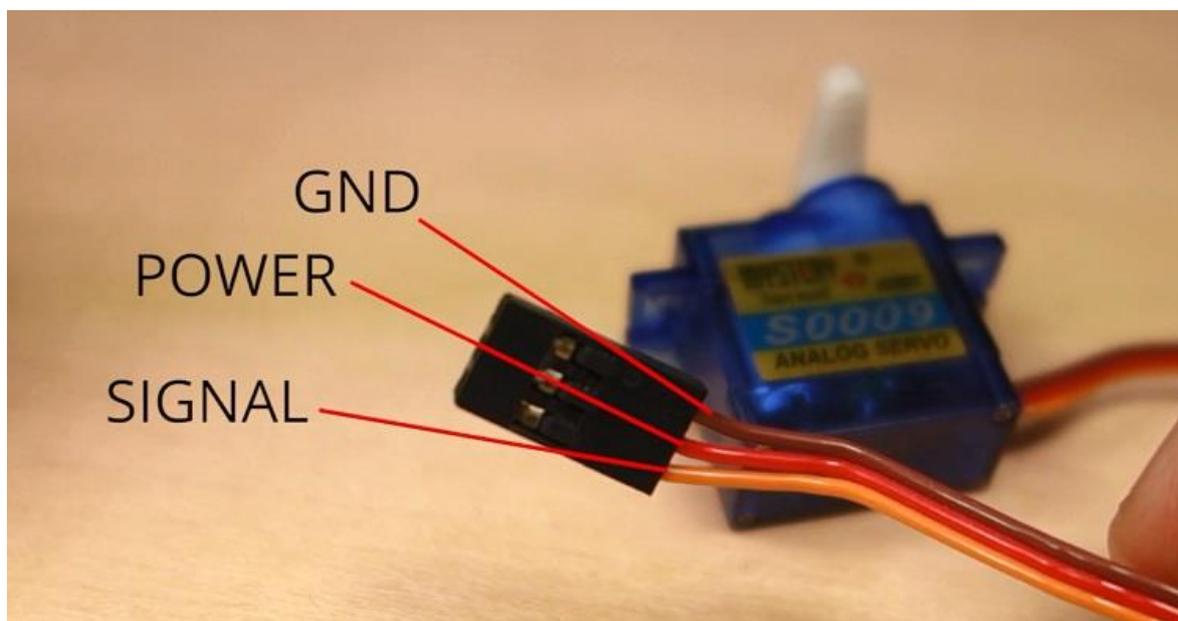
Unit 9 - ESP32 Control Servo Motor Remotely (Web Server)



In this Unit we're going to show you how to build a web server with the ESP32 that controls the shaft's position of a servo motor using a slider. First, we'll take a quick look on how to control a servo with the ESP32, and then we'll build the web server.

Connecting the servo motor to the ESP32

Servo motors have three wires: power, ground, and signal. The power is usually red, the GND is black or brown, and the signal wire is usually yellow, orange, or white.

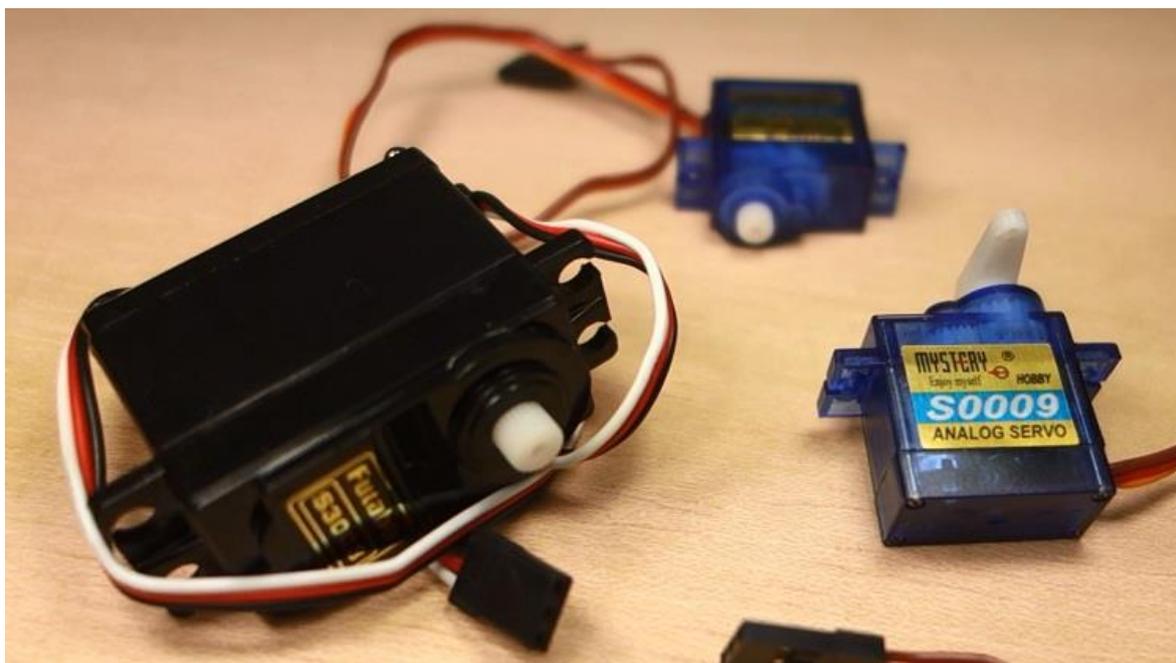


Wire	Color
Power	Red
GND	Black, or brown
Signal	Yellow, orange, or white

When using a small servo like the S0009 as shown in the figure below, you can power it directly from the ESP32.



But, if you're using more than one servo or other type, you'll probably need to power up your servos using an external power supply.



If you're using a small servo like the S0009, you need to connect:

- GND -> ESP32 GND pin
- Power -> ESP32 VIN pin
- Signal -> GPIO 13 (or any PWM pin)

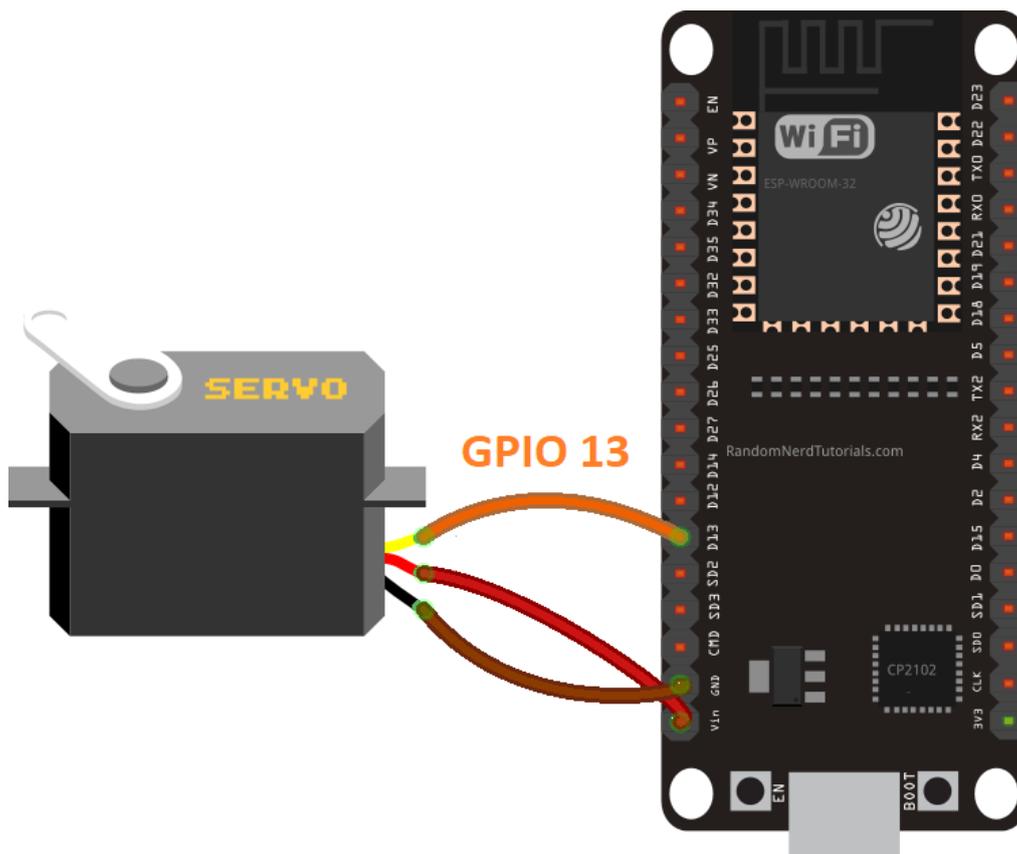
Note: In this case, you can use any ESP32 GPIO, because any GPIO is able to produce a PWM signal. However, we don't recommend using GPIOs 9, 10, and 11 that are connected to the integrated SPI flash and are not recommend for other uses.

Schematic

In our examples we'll connect the signal wire to GPIO 13. So, you can follow the next schematic diagram to wire your servo motor.

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

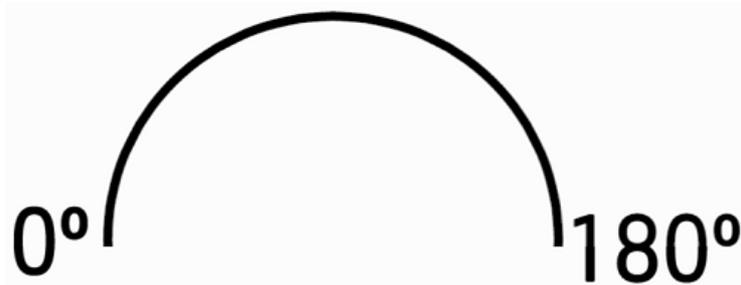
- [ESP32 DOIT DEVKIT V1 Board](#)
- [Micro Servo Motor – S0009](#) or [Servo Motor – S0003](#)
- [Jumper wires](#)



(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.)

How to Control a Servo?

You can position the servo's shaft in various angles from 0 to 180°. Servos are controlled using a pulse width modulation (PWM) signal. This means that the PWM signal sent to the motor will determine the shaft's position.



To control the motor you can simply use the PWM capabilities of the ESP32 by sending a 50Hz signal with the appropriate pulse width – check “ESP32 Pulse-Width Modulation (PWM)” to see how to generate PWM signals with the ESP32.

Or you can use a library to make your life much simpler.

Installing the ESP32_Arduino_Servo_Library

The [ESP32 Arduino Servo Library](#) makes it easier to control a servo motor with your ESP32, using the Arduino IDE. Follow the next steps to install the library in your Arduino IDE:

- 1) [Click here to download the ESP32_Arduino_Servo_Library](#). You should have a .zip folder in your Downloads folder
- 2) Unzip the .zip folder and you should get ESP32-Arduino-Servo-Library-Master folder
- 3) Rename your folder from ~~ESP32-Arduino-Servo-Library-Master~~ to ESP32_Arduino_Servo_Library
- 4) Move the ESP32_Arduino_Servo_Library folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE
- 6) Testing an Example

After installing the library, go to your Arduino IDE. Make sure you have the ESP32 board selected, and then, go to **File** ▶ **Examples** ▶ **ServoESP32** ▶ **Simple Servo**.

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Servo/Sweep/Sweep.ino>

```
/*  
Rui Santos  
Complete project details at http://randomnerdtutorials.com  
Written by BARRAGAN and modified by Scott Fitzgerald  
*/
```

```

#include <Servo.h>

Servo myservo; // create servo object to control a servo
// twelve servo objects can be created on most boards

int pos = 0;    // variable to store the servo position

void setup() {
  myservo.attach(13); // attaches the servo on pin 13 to the servo object
}

void loop() {
  for (pos = 0; pos <= 180; pos += 1) { // goes from 0 degrees to 180
degrees
    // in steps of 1 degree
    myservo.write(pos);           // tell servo to go to position in
variable 'pos'
    delay(15);                    // waits 15ms for the servo to reach
the position
  }
  for (pos = 180; pos >= 0; pos -= 1) { // goes from 180 degrees to 0
degrees
    myservo.write(pos);           // tell servo to go to position in
variable 'pos'
    delay(15);                    // waits 15ms for the servo to reach
the position
  }
}

```

Understanding the code

This sketch rotates the servo 180 degrees to one side, and 180 degrees to the other. Let's see how it works.

First, you need to include the Servo library:

```
#include <Servo.h>
```

Then, you need to create a servo object. In this case it is called `myservo`.

```
Servo myservo; // create servo object to control a servo
```

setup()

In the `setup()`, you attach GPIO 13 to the servo object.

```

void setup() {
  myservo.attach(13); // attaches the servo on pin 13 to the servo object
}

```

loop()

In the `loop()`, we change the motor's shaft position from 0 to 180 degrees, and then from 180 to 0 degrees. To set the shaft to a particular position, you just need to use the `.write()` method on the servo object. You pass as an argument, an integer number with the position in degrees.

```
myservo.write(pos)
```

Testing the Sketch

Upload the code to your ESP32. After uploading the code, you should see the motor's shaft rotating to one side and then, to the other.



Creating the Web Server

You should already be familiar on how to build a web server. We recommend checking the previous Units for a refresher:

- ESP32 Web Server – Control Outputs
- ESP32 Web Server – HTML and CSS Basics (Part 1/2)
- ESP32 Web Server – HTML in Arduino IDE (Part 2/2)

Now that you know how to control a servo with the ESP32, let's create the web server to control it. We'll use the same method used in previous units to create the web server.

The web server we'll build:

- Contains a slider from 0 to 180, that you can adjust to control the servo's shaft position;
- The current slider value is automatically updated in the web page, as well as the shaft position, without the need to refresh the web page. For this, we use AJAX to send HTTP requests to the ESP32 on the background;
- Refreshing the web page doesn't change the slider value, neither the shaft position.



Creating the HTML Page

Let's start by taking a look at the HTML text the ESP32 needs to send to your browser.

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Servo/slider.html>

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,">
  <style>
    body {
      text-align: center;
      font-family: "Trebuchet MS", Arial;
      margin-left:auto;
      margin-right:auto;
    }
    .slider {
      width: 300px;
    }
  </style>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></sc
ript>
</head>
<body>
  <h1>ESP32 with Servo</h1>
  <p>Position: <span id="servoPos"></span></p>
  <input type="range" min="0" max="180" class="slider" id="servoSlider"
onchange="servo(this.value)"/>
  <script>
    var slider = document.getElementById("servoSlider");
    var servoP = document.getElementById("servoPos");
    servoP.innerHTML = slider.value;
    slider.oninput = function() {
```

```
    slider.value = this.value;
    servoP.innerHTML = this.value;
  }
  $.ajaxSetup({timeout:1000});
  function servo(pos) {
    $.get("/?value=" + pos + "&");
    {Connection: close};
  }
</script>
</body>
</html>
```

Creating a Slider

The HTML page for this project involves creating a slider. To create a slider in HTML you use the `<input>` tag. The `<input>` tag specifies a field where the user can enter data.

There are a wide variety of input types. To define a slider, use the `"type"` attribute with the `"range"` value. In a slider, you also need to define the minimum and the maximum range using the `"min"` and `"max"` attributes.

```
<input type="range" min="0" max="180" class="slider" id="servoSlider"
onchange="servo(this.value)"/>
```

You also need define other attributes like:

- The **class** to style the slider;
- The **id** to update the current position displayed on the web page;
- And finally, the **onchange** attribute to call the servo function to send an HTTP request to the ESP32 when the slider moves.

Adding JavaScript to the HTML File

Next, you need to add some JavaScript code to your HTML file using the `<script>` and `</script>` tags. This snippet of the code updates the web page with the current slider position:

```
var slider = document.getElementById("servoSlider");
var servoP = document.getElementById("servoPos");
servoP.innerHTML = slider.value;
slider.oninput = function() {
  slider.value = this.value;
  servoP.innerHTML = this.value;
}
```

And the next lines make an HTTP GET request on the ESP IP address in this specific URL path `/?value=[SLIDER_POSITION]&`.

```
$.ajaxSetup({timeout:1000});
function servo(pos) {
  $.get("/?value=" + pos + "&");
```

```
{Connection: close};  
}
```

For example, when the slider is at 0, you make an HTTP GET request on the following URL:

```
http://192.168.1.135/?value=0&
```

And when the slider is at 180 degrees, you'll have something as follows:

```
http://192.168.1.135/?value=180&
```

This way, when the ESP32 receives the GET request, it can retrieve the value parameter in the URL and move the servo motor to the right position.

Note: if you don't know how to include HTML text in the Arduino IDE, follow this previous Unit: **"ESP32 Web Server - HTML in Arduino IDE"**.

Code

Now, we need to include the previous HTML text in the sketch and rotate the servo accordingly. This next sketch does precisely that.

Copy the following code to your Arduino IDE, but don't upload it yet. First, we'll take a quick look on how it works.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Servo/WiFi_Web_Server_Servo/WiFi_Web_Server_Servo.ino

```
/*  
  Rui Santos  
  Complete project details at http://randomnerdtutorials.com  
  */  
  
#include <WiFi.h>  
#include <Servo.h>  
  
Servo myservo; // create servo object to control a servo  
// twelve servo objects can be created on most boards  
  
// GPIO the servo is attached to  
static const int servoPin = 13;  
  
// Replace with your network credentials  
const char* ssid      = "REPLACE_WITH_YOUR_SSID";  
const char* password = "REPLACE_WITH_YOUR_PASSWORD";  
  
// Set web server port number to 80  
WiFiServer server(80);  
  
// Variable to store the HTTP request  
String header;
```

```

// Decode HTTP GET value
String valueString = String(5);
int pos1 = 0;
int pos2 = 0;

// 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() {
  Serial.begin(115200);

  myservo.attach(servoPin); // attaches the servo on the servoPin to the
servo object

  // 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
    while (client.connected() && currentTime - previousTime <= timeoutTime)
{ // loop while the client's connected
      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>body { text-align: center; font-family:
\"Trebuchet MS\", Arial; margin-left:auto; margin-right:auto;}");
                client.println(".slider { width: 300px; }</style>");
                client.println("<script
src=\"https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js\"></
script>");

                // Web Page
                client.println("</head><body><h1>ESP32 with Servo</h1>");
                client.println("<p>Position:                                <span
id=\"servoPos\"></span></p>");
                client.println("<input type=\"range\" min=\"0\" max=\"180\"
class=\"slider\" id=\"servoSlider\" onchange=\"servo(this.value)\"
value=\""+valueString+"\"/>");

                client.println("<script>var                                slider
document.getElementById(\"servoSlider\");");
                client.println("var                                servoP
document.getElementById(\"servoPos\"); servoP.innerHTML = slider.value;");
                client.println("slider.oninput = function() { slider.value =
this.value; servoP.innerHTML = this.value; }");
                client.println("$.ajaxSetup({timeout:1000}); function servo(pos)
{ ");
                client.println("$.get(\"/?value=\" + pos + \"%&\"); {Connection:
close};}</script>");
                client.println("</body></html>");

                //GET /?value=180& HTTP/1.1
                if(header.indexOf("GET /?value=")>=0) {
                    pos1 = header.indexOf('=');
                    pos2 = header.indexOf('&');
                    valueString = header.substring(pos1+1, pos2);

```

```

        //Rotate the servo
        myservo.write(valueString.toInt());
        Serial.println(valueString);
    }
    // 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("");
}
}

```

How the Code Works

We've covered how to build a Web Server in great detail in previous Units, so we'll just take a look at the parts that are relevant for this example.

First, we include the Servo library, and create a servo object called `myservo`.

```
#include <Servo.h>
```

```
Servo myservo;
```

We also create a variable to hold the GPIO number the servo is connected to. In this case, GPIO 13.

```
static const int servoPin = 13;
```

Don't forget that you need to modify the following two lines to include your network credentials.

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

Then, create a couple of variables that will be used to extract the slider position from the HTTP request.

```
// Decode HTTP GET value
String valueString = String(5);
int pos1 = 0;
int pos2 = 0;
```

setup()

In the `setup()`, you need to attach the servo to the GPIO it is connected to, with `myservo.attach()`.

```
myservo.attach(servoPin); // attaches the servo on the servoPin to the servo object
```

loop()

You should already be familiar with the part of the `loop()` that creates the web server and sends the HTML text to display the web page. So, we'll skip that part.

The following part of the code retrieves the slider value from the HTTP request.

```
//GET /?value=180& HTTP/1.1
if(header.indexOf("GET /?value=")>=0) {
  pos1 = header.indexOf('=');
  pos2 = header.indexOf('&');
  valueString = header.substring(pos1+1, pos2);
```

When you move the slider, you make an HTTP request on the following URL, that contains the slider position between the = and & signs.

```
http://your-esp-ip-address/?value=[SLIDER_POSITION]&
```

The slider position value is saved in the `valueString` variable. Then, we set the servo to that specific position using `myservo.write()` with the `valueString` variable as an argument. The `valueString` variable is a string, so we need to use the `.toInt()` method to convert it into an integer number – the data type accepted by the `write()` method.

```
myservo.write(valueString.toInt());
```

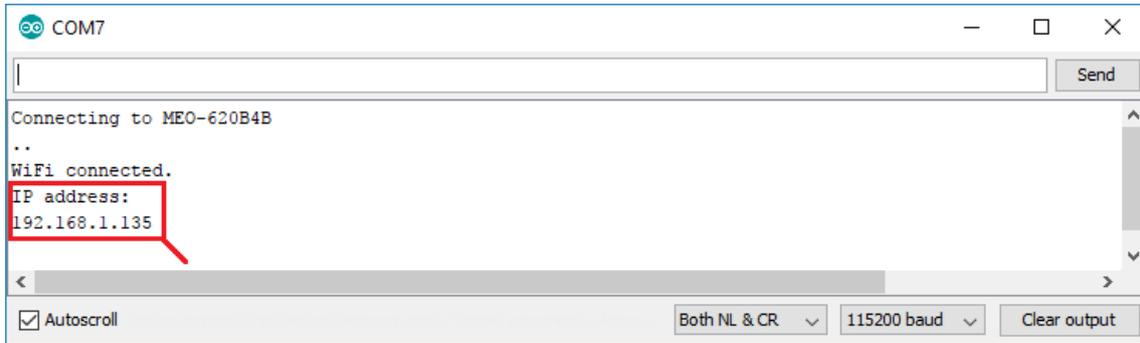
Testing the Web Server

Now you can upload the code to your ESP32 – make sure you have the right board and COM port selected. Also don't forget to modify the code to include your network credentials.

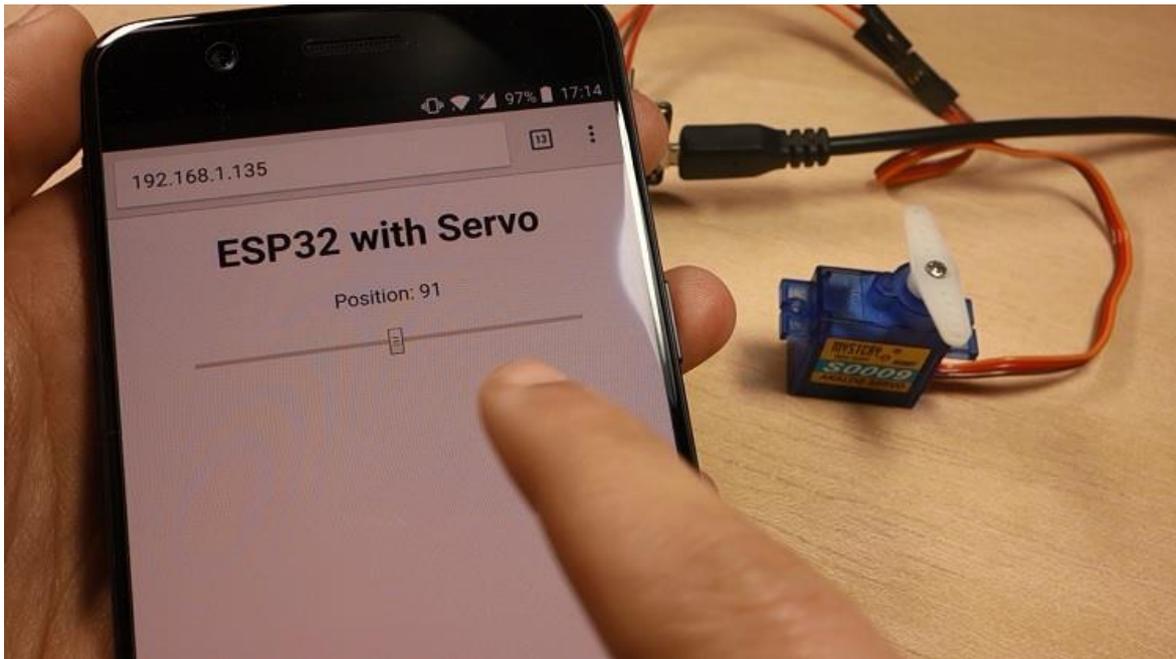
After uploading the code, open the Serial Monitor at a baud rate of 115200.



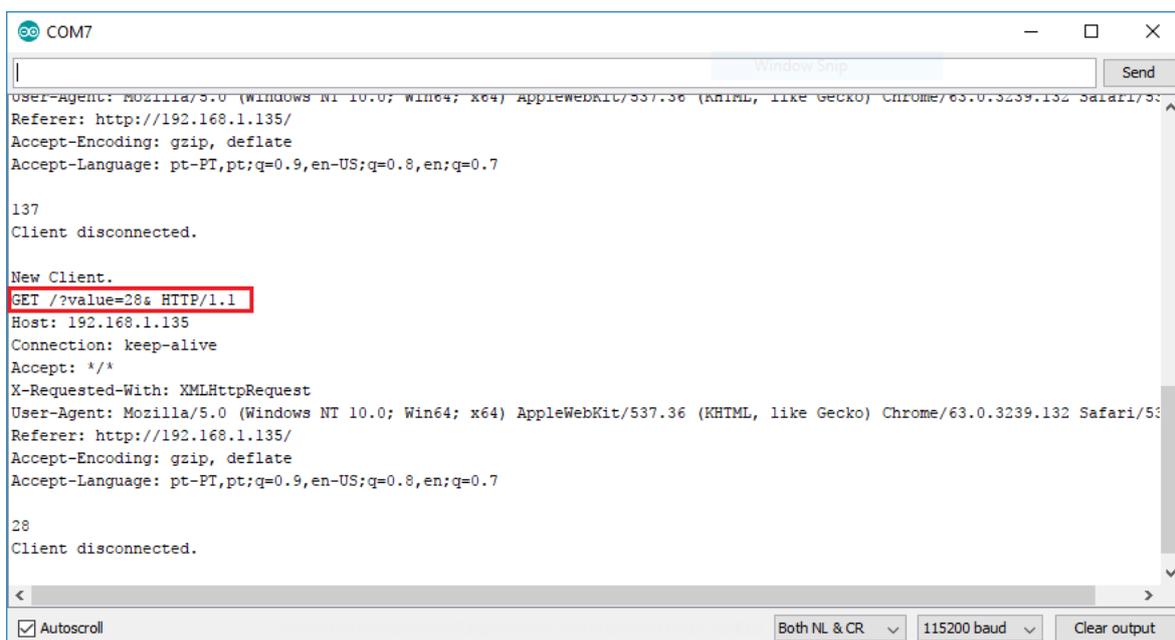
Press the ESP32 “Enable” button to restart the board, and copy the ESP32 IP address that shows up on the Serial Monitor.



Open your browser, paste the ESP IP address, and you should see the web page you've created previously. Move the slider to control the servo motor.



In the Serial Monitor, you can also see the HTTP requests you're sending to the ESP32 when you move the slider.



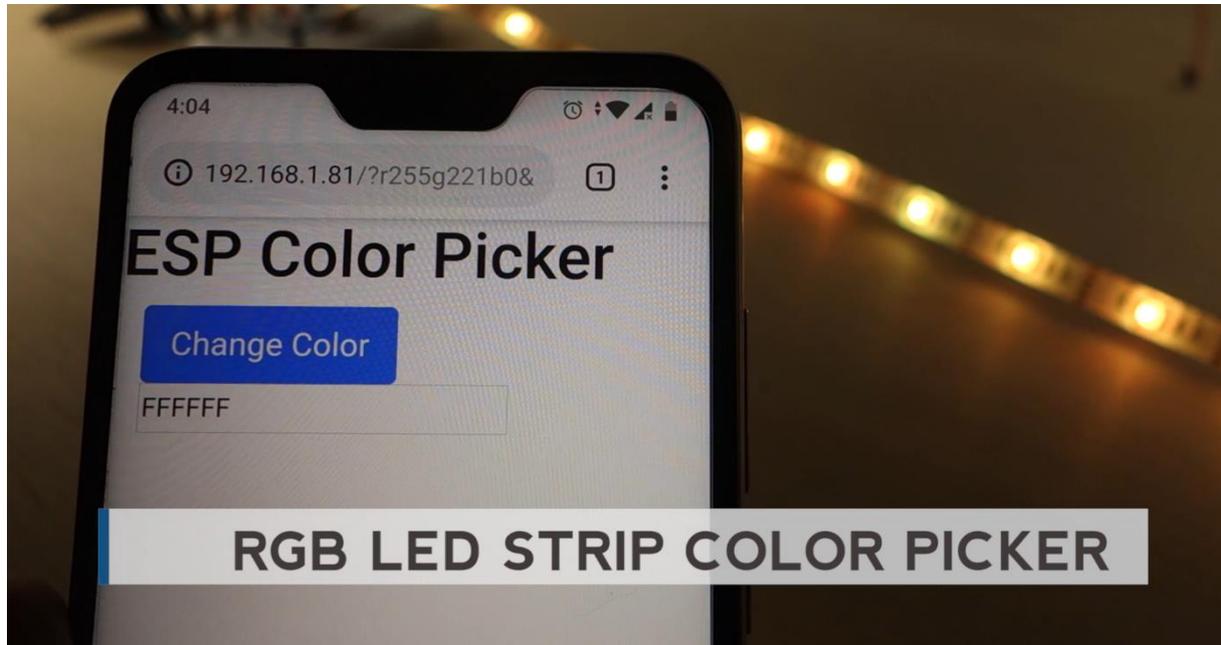
Experiment with your web server for a while to see if it's working properly.

Wrapping Up

In summary, in this Unit you've learned how to control a servo motor with the ESP32 and how to create a web server with a slider to control its position.

This is just an example on how to control a servo motor. Instead of a slider, you can use a text input field, several buttons with predefined angles, or any other suitable input fields.

Unit 10 - Color Picker Web Server for RGB LED Strip



In this project we'll show you how to remotely control an RGB LED strip with your ESP32 using a web server with a color picker.

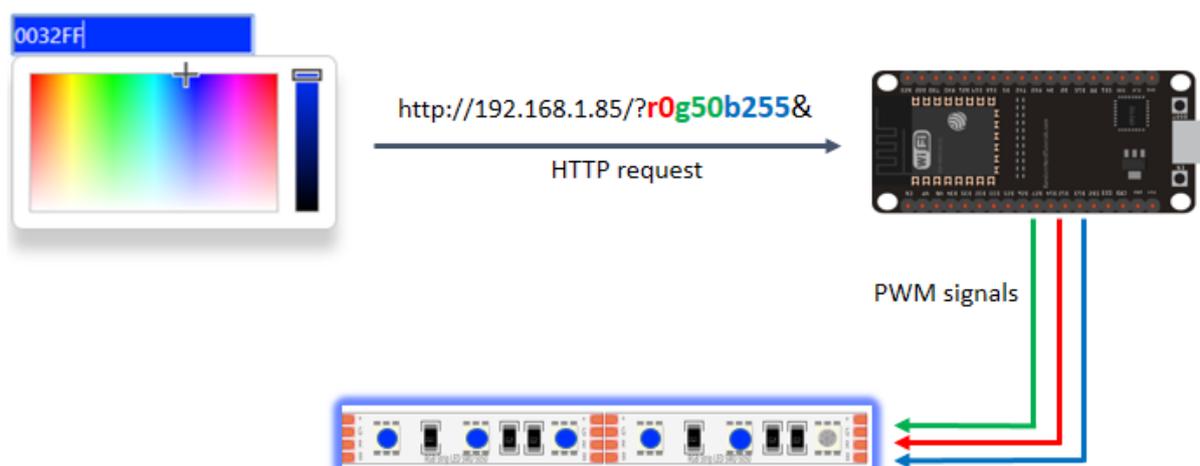
To better understand this project, we recommend following the [ESP32 PWM tutorial](#) as well as the [ESP32 Web Server tutorial](#):

- [ESP32 Pulse-Width Modulation \(PWM\)](#)
- [ESP32 Web Server - Control Outputs](#)

It may also be helpful to take a quick look at the blog post "[How RGB LEDs work](#)".

Project Overview

Before getting started, let's see how this project works:



- The ESP32 web server has a color picker.
- When you chose a color, your browser makes a request on a URL that contains the R, G, and B parameters of the selected color.
- Your ESP32 receives the request and splits the value for each color parameter.
- Then, it sends a PWM signal with the corresponding value to the GPIOs that are controlling the strip.

Schematic

For this project, we'll be using an RGB LED strip that can be powered with 5V.

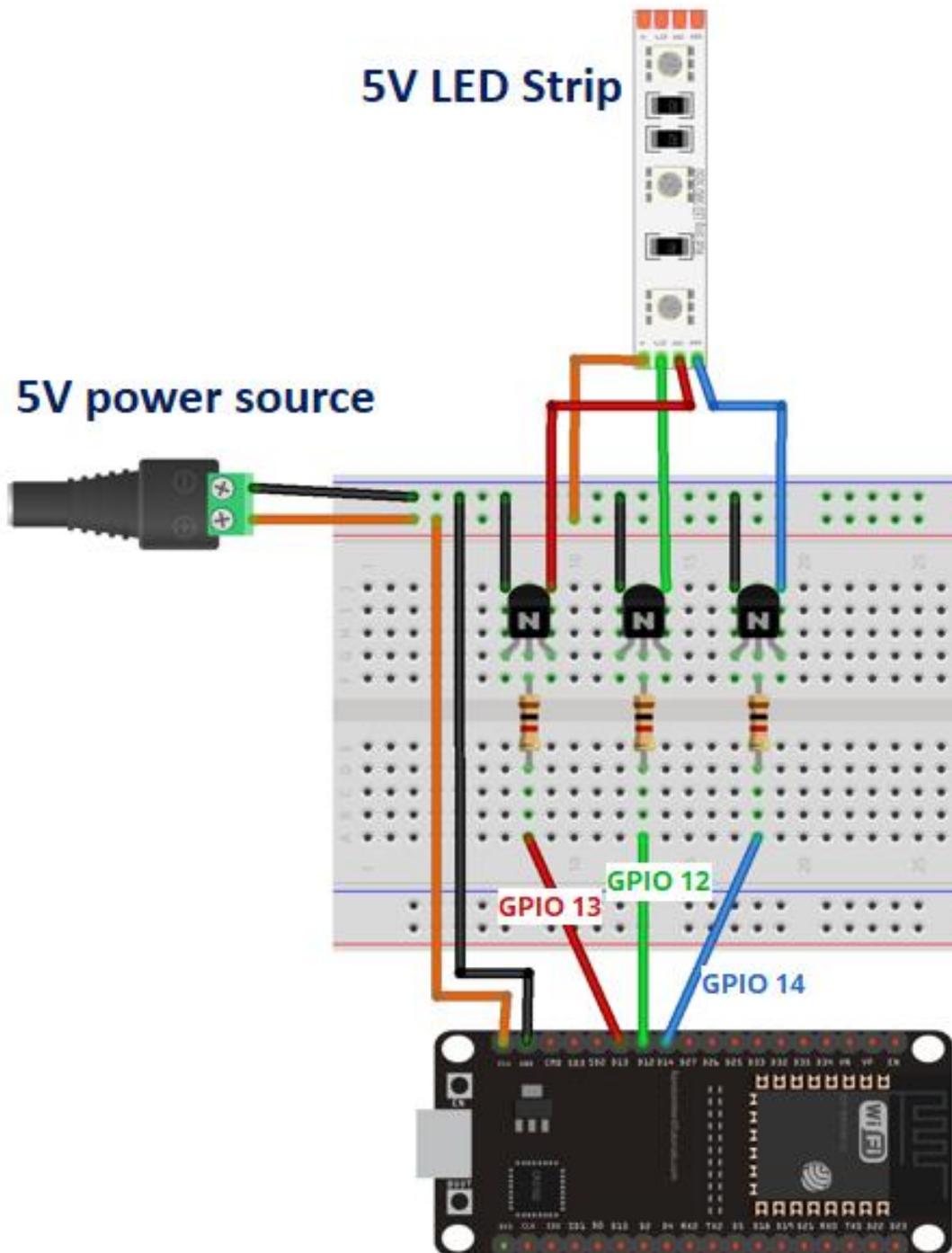


Note: there are other similar strips that require 12V to operate. You can still use them with the code provided, but you need a suitable power supply.

To follow this example you need the following parts:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [RGB LED Strip \(5V\)](#)
- 3x NPN transistors (see how to choose the proper transistor in the project description)
- 3x [1k ohm resistors](#)
- [Jumper wires](#)
- [Breadboard](#)

In this example, we'll be powering the LED strip and the ESP32 using the same 5V power supply. Follow the next schematic diagram to connect the strip to your ESP32.

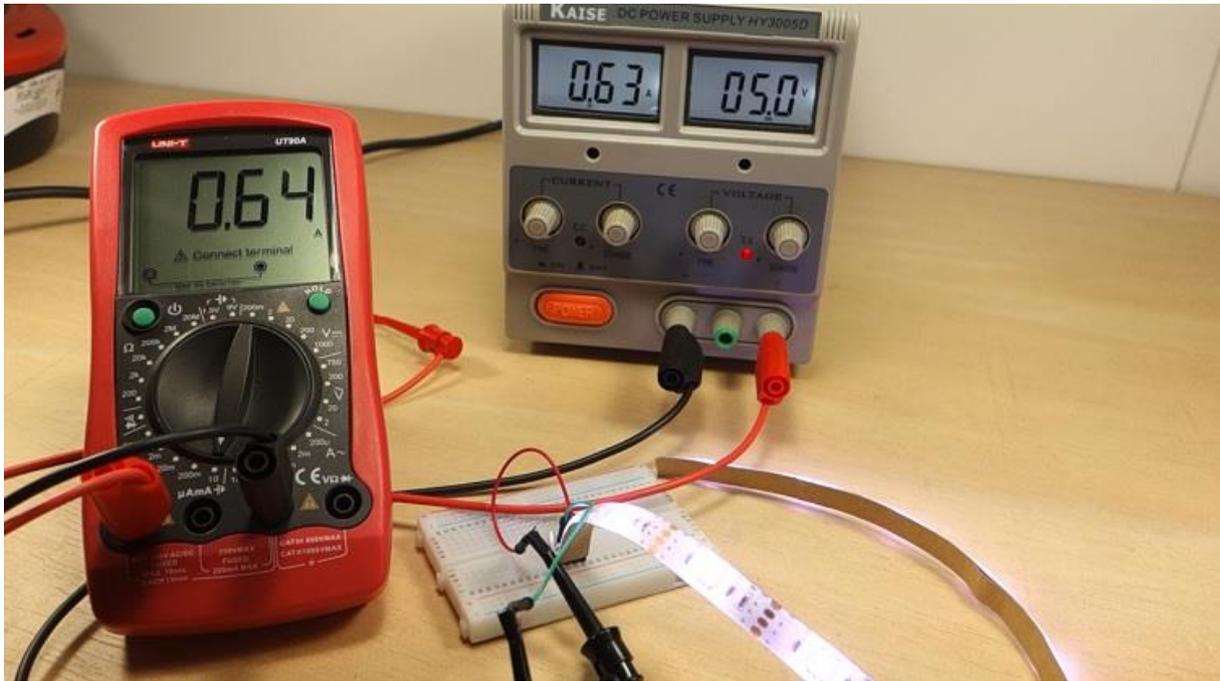


(This schematic uses the ESP32 DEVKIT V1 module version with 36 GPIOs – if you're using another model, please check the pinout for the board you're using.)

In this circuit, we're using the S8050 NPN transistor. However, depending on how many LEDs you have in your strip, you might need to use an NPN transistor that can handle more continuous current in the collector pin.

To determine the max current used by your LED strip, you can measure the current consumption when all the LEDs are at maximum brightness (when the color is white).

Since we're using 12 LEDs, the maximum current is approximately 630 mA at full brightness in white color. So, we can use the S8050 NPN transistor that can handle up to 700 mA.



Note: your strip consumes the maximum current when you set white color (this is the same as setting all three colors to the maximum brightness). Setting other colors draws less current, so you'll probably won't have your strip using the maximum current.

Code

After assembling the circuit, copy the following code to your Arduino IDE to program the ESP32.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/raw/master/code/WiFi_Web_Server_Color_Picker/WiFi_Web_Server_Color_Picker.ino

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

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

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

// Decode HTTP GET value
String redString = "0";
String greenString = "0";
String blueString = "0";
int pos1 = 0;
int pos2 = 0;
int pos3 = 0;
int pos4 = 0;

// Variable to store the HTTP request
String header;
```

```

// Red, green, and blue pins for PWM control
const int redPin = 13;    // 13 corresponds to GPIO13
const int greenPin = 12; // 12 corresponds to GPIO12
const int bluePin = 14;   // 14 corresponds to GPIO14

// Setting PWM frequency, channels and bit resolution
const int freq = 5000;
const int redChannel = 0;
const int greenChannel = 1;
const int blueChannel = 2;
// Bit resolution 2^8 = 256
const int resolution = 8;

// 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() {
  Serial.begin(115200);
  // configure LED PWM functionalites
  ledcSetup(redChannel, freq, resolution);
  ledcSetup(greenChannel, freq, resolution);
  ledcSetup(blueChannel, freq, resolution);

  // attach the channel to the GPIO to be controlled
  ledcAttachPin(redPin, redChannel);
  ledcAttachPin(greenPin, greenChannel);
  ledcAttachPin(bluePin, blueChannel);

  // 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) {
    currentTime = millis();
    previousTime = currentTime;
    Serial.println("New Client.");
    String currentLine = "";
    while (client.connected() && currentTime - previousTime <=
timeoutTime) {
      currentTime = millis();

```

```

if (client.available()) {
    char c = client.read();
    Serial.write(c);
    header += c;
    if (c == '\n') {
        if (currentLine.length() == 0) {
            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:,\">");
            client.println("<link rel=\"stylesheet\"
href=\"https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootst
rap.min.css\">");
            client.println("<script
src=\"https://cdnjs.cloudflare.com/ajax/libs/jscolor/2.0.4/jscolor.m
in.js\"></script>");
            client.println("</head><body><div
class=\"container\"><div class=\"row\"><h1>ESP Color
Picker</h1></div>");
            client.println("<a class=\"btn btn-primary btn-lg\"
href=\"#\" id=\"change_color\" role=\"button\">Change Color</a> ");
            client.println("<input class=\"jscolor
{onFineChange:'update(this)'}\" id=\"rgb\"></div>");
            client.println("<script>function update(picker)
{document.getElementById('rgb').innerHTML =
Math.round(picker.rgb[0]) + ', ' + Math.round(picker.rgb[1]) + ', '
+ Math.round(picker.rgb[2]);}");
            client.println("document.getElementById(\"change_color\"
).href=\"?r\" + Math.round(picker.rgb[0]) + \"g\"
+ Math.round(picker.rgb[1]) + \"b\" + Math.round(picker.rgb[2]) +
\"&\";}</script></body></html>");
            client.println();

            // Request sample: /?r201g32b255&
            // Red = 201 | Green = 32 | Blue = 255
            if(header.indexOf("GET /?r") >= 0) {
                pos1 = header.indexOf('r');
                pos2 = header.indexOf('g');
                pos3 = header.indexOf('b');
                pos4 = header.indexOf('&');
                redString = header.substring(pos1+1, pos2);
                greenString = header.substring(pos2+1, pos3);
                blueString = header.substring(pos3+1, pos4);
                /*Serial.println(redString.toInt());
                Serial.println(greenString.toInt());
                Serial.println(blueString.toInt());*/
                ledcWrite(redChannel, redString.toInt());
                ledcWrite(greenChannel, greenString.toInt());
                ledcWrite(blueChannel, blueString.toInt());
            }
            break;
        } else {
            currentLine = "";
        }
    }
}

```

```

        } else if (c != '\r') {
            currentLine += c;
        }
    }
}
// Clear the header variable
header = "";
// Close the connection
client.stop();
Serial.println("Client disconnected.");
Serial.println("");
}
}

```

Before uploading the code, don't forget to insert your network credentials so that the ESP can connect to your local network.

```

const char* ssid      = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

How the Code Works

If you've built a web server with the ESP32 before, this code is not much different. It adds the color picker to the web page and decodes the request to control the strip color. So, we'll just take a look at the relevant parts for this project.

The following lines define string variables to hold the R, G, and B parameters from the request.

```

String redString = "0";
String greenString = "0";
String blueString = "0";

```

The next four variables are used to decode the HTTP request later on.

```

int pos1 = 0;
int pos2 = 0;
int pos3 = 0;
int pos4 = 0;

```

Create three variables for the GPIOs that will control the strip R, G, and B parameters. In this case we're using GPIO 13, GPIO 12, and GPIO 14.

```

const int redPin = 13;
const int greenPin = 12;
const int bluePin = 14;

```

These GPIOs need to output PWM signals, so we need to configure the PWM properties first. Set the PWM signal frequency to 5000 Hz. Then, associate a PWM channel for each color.

```

const int freq = 5000;
const int redChannel = 0;

```

```
const int greenChannel = 1;
const int blueChannel = 2;
```

And finally, set the resolution of the PWM channels to 8-bits.

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

In the `setup()`, assign the PWM properties to the PWM channels.

```
ledcSetup(redChannel, freq, resolution);
ledcSetup(greenChannel, freq, resolution);
ledcSetup(blueChannel, freq, resolution);
```

Attach the PWM channels to the corresponding GPIOs

```
ledcAttachPin(redPin, redChannel);
ledcAttachPin(greenPin, greenChannel);
ledcAttachPin(bluePin, blueChannel);
```

The following code section displays the color picker in your web page and makes a request based on the color you've picked.

```
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:,\">");
client.println("<link rel=\"stylesheet\"
href=\"https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootst
rap.min.css\">");
client.println("<script
src=\"https://cdnjs.cloudflare.com/ajax/libs/jscolor/2.0.4/jscolor.m
in.js\"></script>");
client.println("</head><body><div class=\"container\"><div
class=\"row\"><h1>ESP Color Picker</h1></div>");
client.println("<a class=\"btn btn-primary btn-lg\" href=\"#\"
id=\"change_color\" role=\"button\">Change Color</a> ");
client.println("<input class=\"jscolor
{onFineChange:'update(this)'}\" id=\"rgb\"></div>");
client.println("<script>function update(picker)
{document.getElementById('rgb').innerHTML =
Math.round(picker.rgb[0]) + ', ' + Math.round(picker.rgb[1]) + ', '
+ Math.round(picker.rgb[2]);}");
client.println("document.getElementById(\"change_color\").href=\"?r\"
" + Math.round(picker.rgb[0]) + \"g\" + Math.round(picker.rgb[1]) +
\"b\" + Math.round(picker.rgb[2]) +
\"&\";}</script></body></html>");
client.println();
```

When you pick a color, you receive a request with the following format.

```
/?r201g32b255&
```

So, we need to split this string to get the R, G, and B parameters. The parameters are saved in `redString`, `greenString`, and `blueString` variables and can have values between 0 and 255.

```
pos1 = header.indexOf('r');
pos2 = header.indexOf('g');
pos3 = header.indexOf('b');
pos4 = header.indexOf('&');
redString = header.substring(pos1+1, pos2);
greenString = header.substring(pos2+1, pos3);
blueString = header.substring(pos3+1, pos4);
```

To control the strip, use the `ledcWrite()` function to generate PWM signals with the values decoded from the HTTP request.

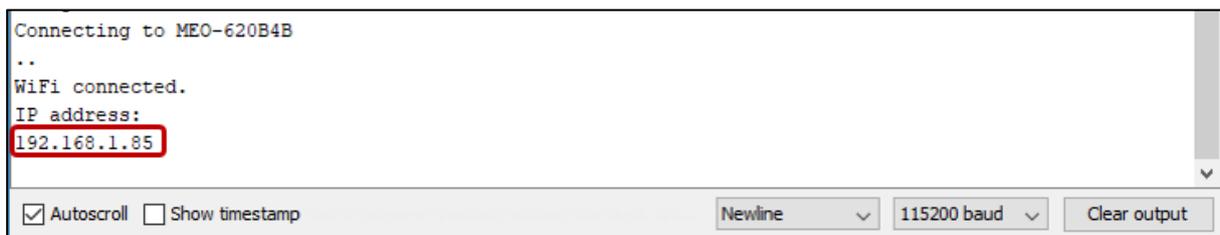
```
ledcWrite(redChannel, redString.toInt());
ledcWrite(greenChannel, greenString.toInt());
ledcWrite(blueChannel, blueString.toInt());
```

Because we get the values in a string variable, we need to convert them to integers using the `toInt()` method.

Demonstration

After inserting your network credentials, select the right board and COM port and upload the code to your ESP32.

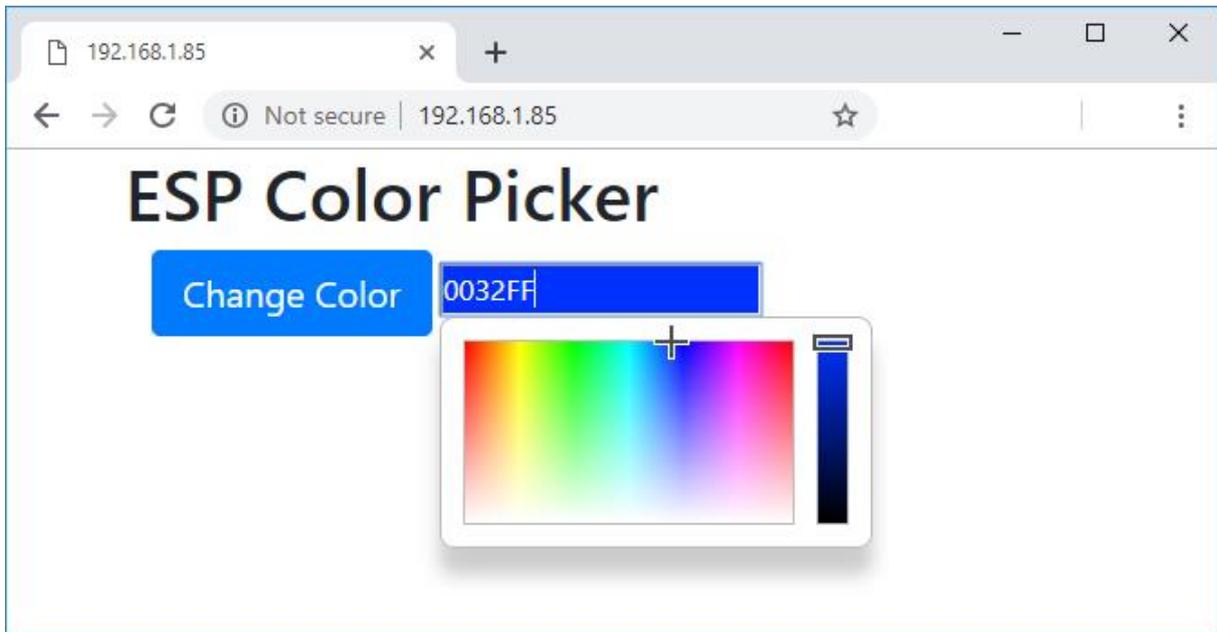
After uploading, open the Serial Monitor at a baud rate of 115200 and press the ESP32 Enable/Reset button. You should get the ESP32 IP address.



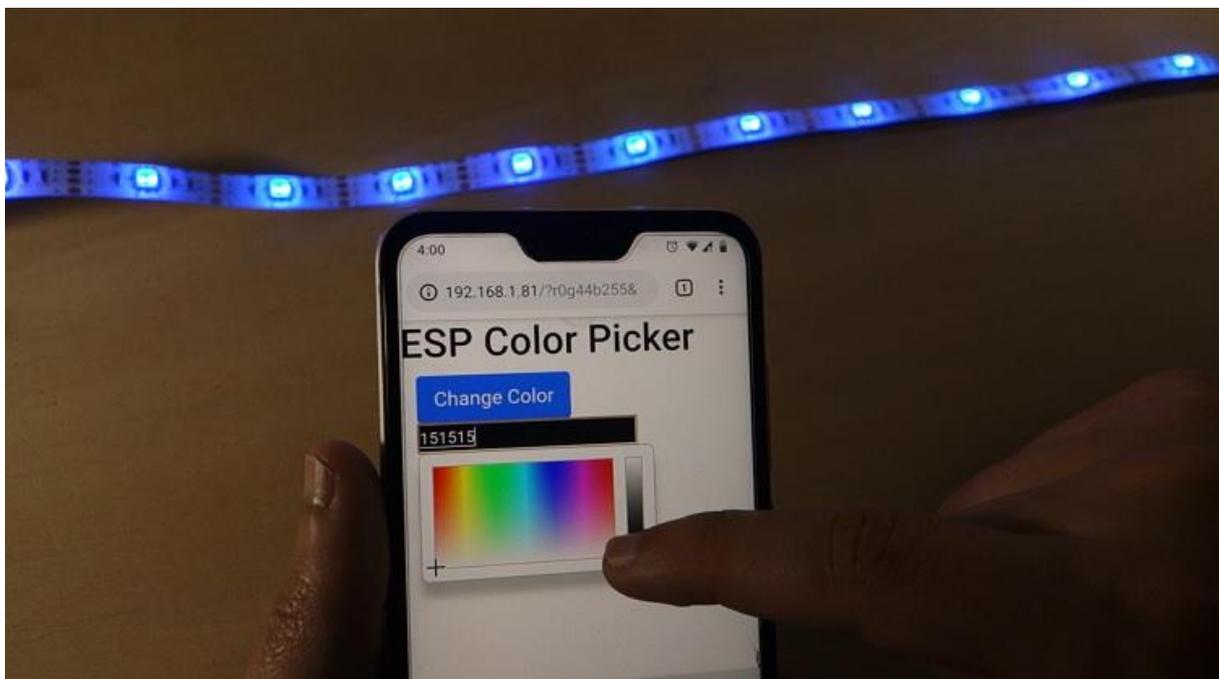
```
Connecting to MEO-620B4B
..
WiFi connected.
IP address:
192.168.1.85
```

Autoscroll Show timestamp Newline 115200 baud Clear output

Open your browser and insert the ESP32 IP address. Now, use the color picker to choose a color for the strip.

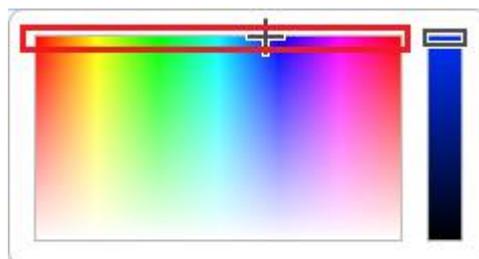


Then, you need to press the **“Change Color”** button for the color to take effect.



To turn off the RGB LED strip, select the black color.

The colors at the top, are the ones that will produce better results.



Now, you use the strip to decorate your house: under the bed, behind a TV, under the kitchen cabinet, and much more.

Unit 11 – 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 build 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](#).

Schematic

Before proceeding to the web server, you need to wire the DHT11 or DHT22 sensor to the ESP32.

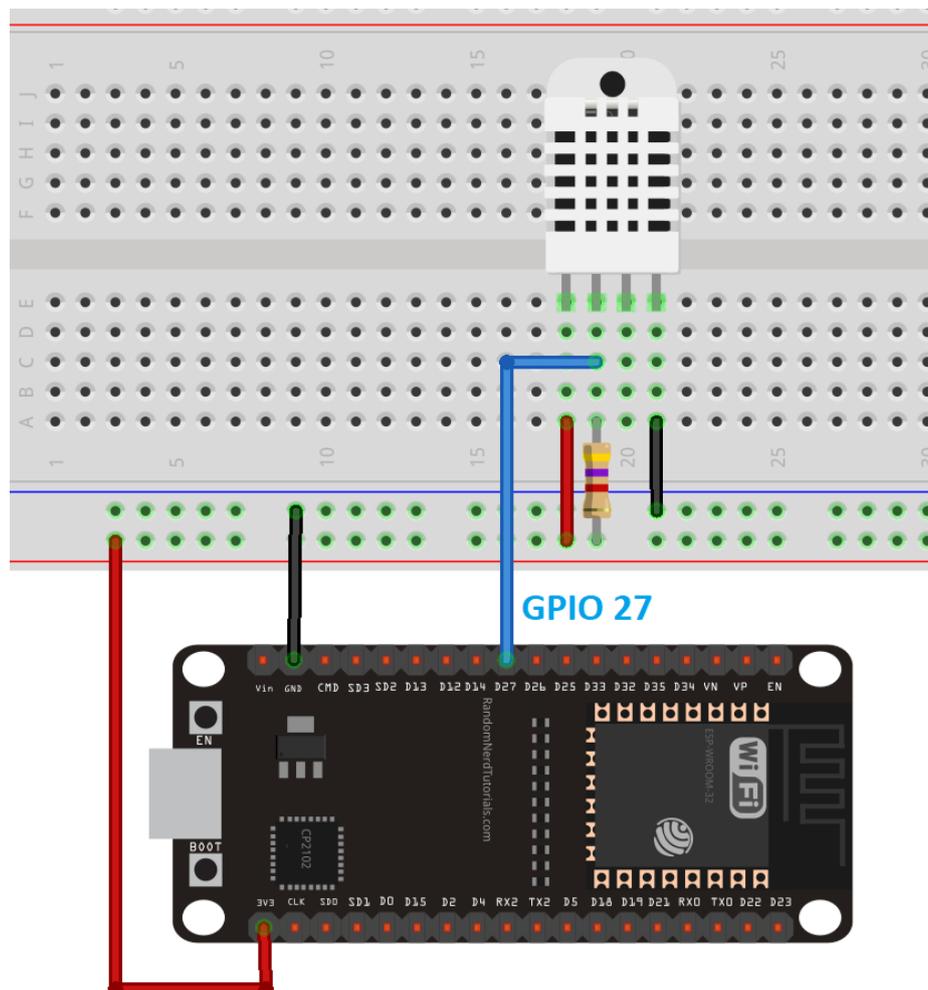
Parts required

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.



(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 can follow the same schematic diagram for the DHT11 sensor.

Installing Libraries

You need to install a couple of 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:

Installing the DHT Sensor Library

1. [Click here to download the DHT Sensor library](#). You should have a .zip folder in your Downloads folder
2. Unzip the .zip folder and you should get DHT-sensor-library-master folder
3. Rename your folder from ~~DHT-sensor-library-master~~ to DHT_sensor
4. Move the DHT_sensor folder to your Arduino IDE installation libraries folder

Installing the Adafruit Unified Sensor Driver

1. [Click here to download the Adafruit Unified Sensor library](#). You should have a .zip folder in your Downloads folder
2. Unzip the .zip folder and you should get Adafruit_sensor-master folder
3. Rename your folder from ~~Adafruit_sensor-master~~ to Adafruit_sensor
4. Move the Adafruit_sensor folder to your Arduino IDE installation libraries folder

Installing the ESPAsyncWebServer library

1. [Click here to download the ESPAsyncWebServer library](#). You should have a .zip folder in your Downloads folder
2. Unzip the .zip folder and you should get ESPAsyncWebServer-master folder
3. Rename your folder from ~~ESPAsyncWebServer-master~~ to ESPAsyncWebServer
4. Move the ESPAsyncWebServer folder to your Arduino IDE installation libraries folder

Installing the Async TCP Library for ESP32

1. [Click here to download the Async TCP](#) library. You should have a .zip folder in your Downloads folder
2. Unzip the .zip folder and you should get AsyncTCP-master folder
3. Rename your folder from AsyncTCP-master to AsyncTCP
4. Move the AsyncTCP folder to your Arduino IDE installation libraries folder
5. Finally, re-open your Arduino IDE

Code

Open your Arduino IDE and copy the following code.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/WiFi_Web_Server_DHT/WiFi_Web_Server_DHT.ino

```
// 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 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 "--";
    }
    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">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/
v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7
mOUStjsKC4pOpQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
    <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">%</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 the Code Works

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 (--) in case 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.

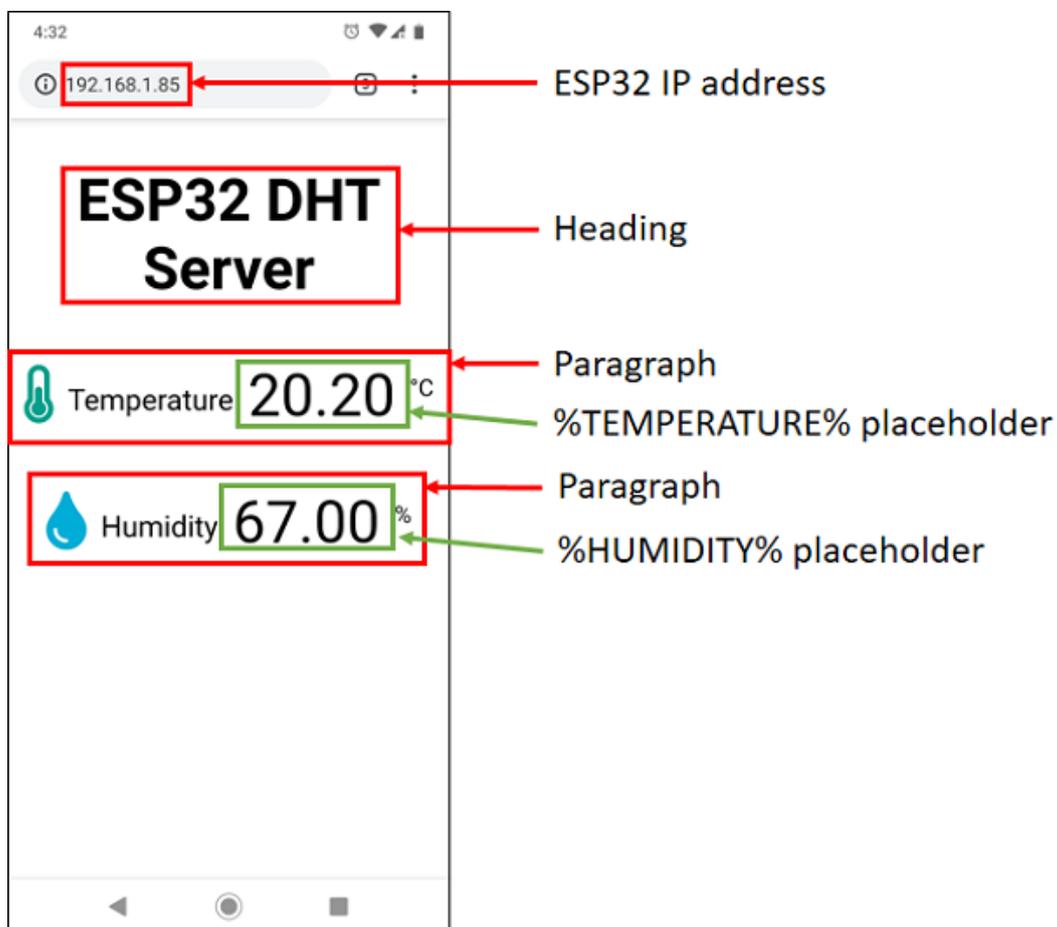
```
return String(t);
```

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 server 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 `<link>` tag is needed to load the icons from the fontawesome website.

```
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7mOUSTjsKC4pOpQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
```

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 all the 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 `<link>` 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 the other 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>
```

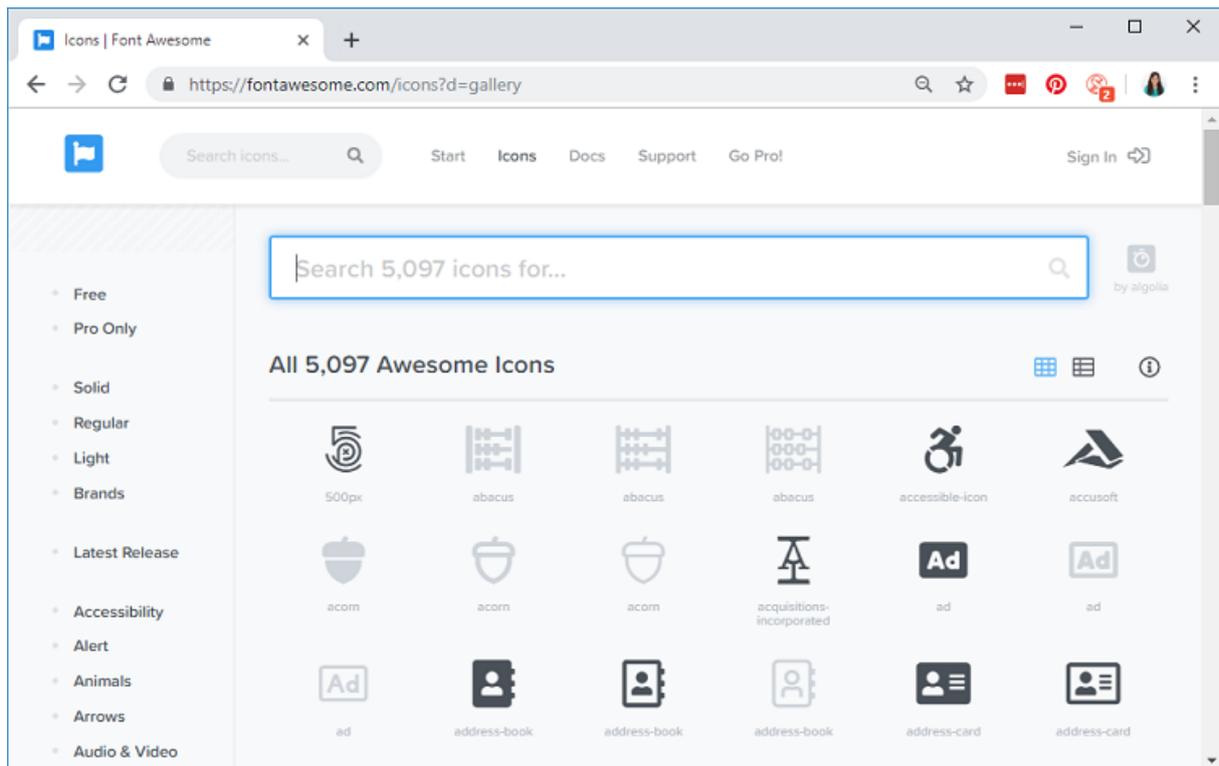
And the paragraph for the humidity is on the following snippet:

```
<p>
  <i class="fas fa-tint" style="color:#00add6;"></i>
  <span class="dht-labels">Humidity</span>
  <span id="humidity">%HUMIDITY%</span>
  <sup class="units">%</sup>
</p>
```

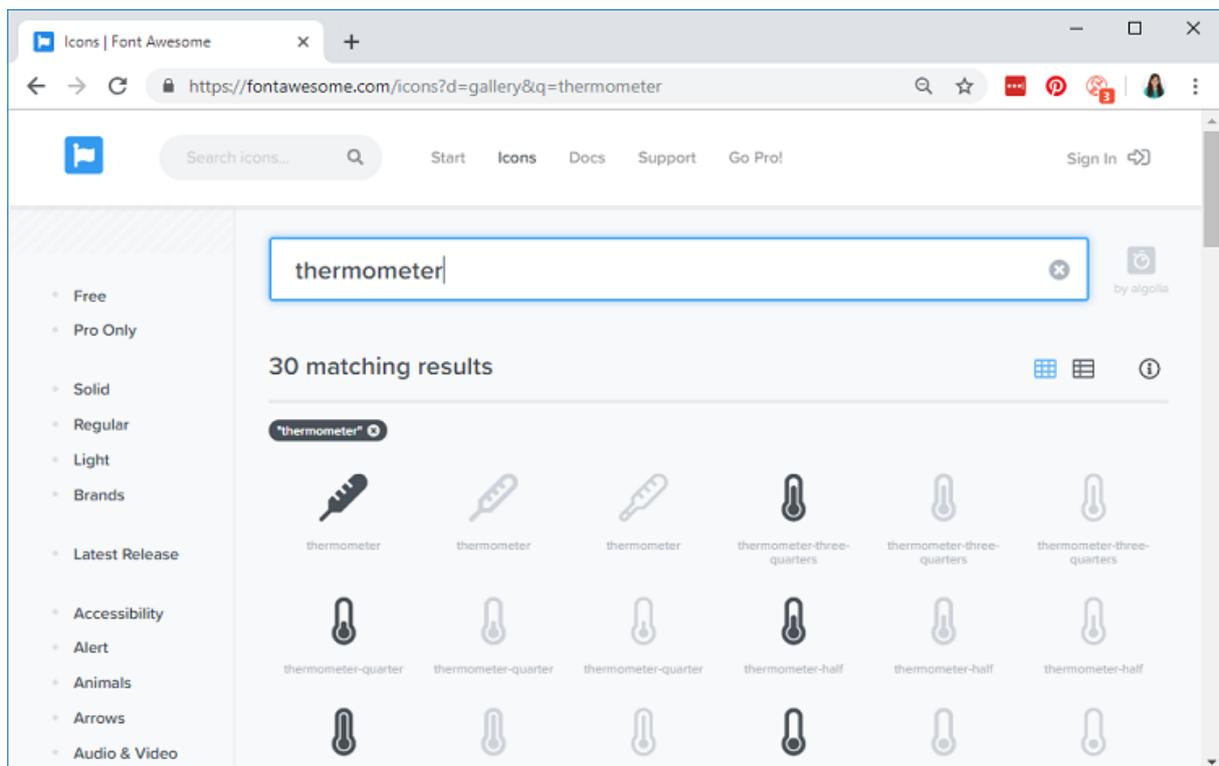
The `<i>` tags display the fontawesome icons.

How to display icons

To chose the icons, go to the [Font Awesome Icons website](https://fontawesome.com/icons).

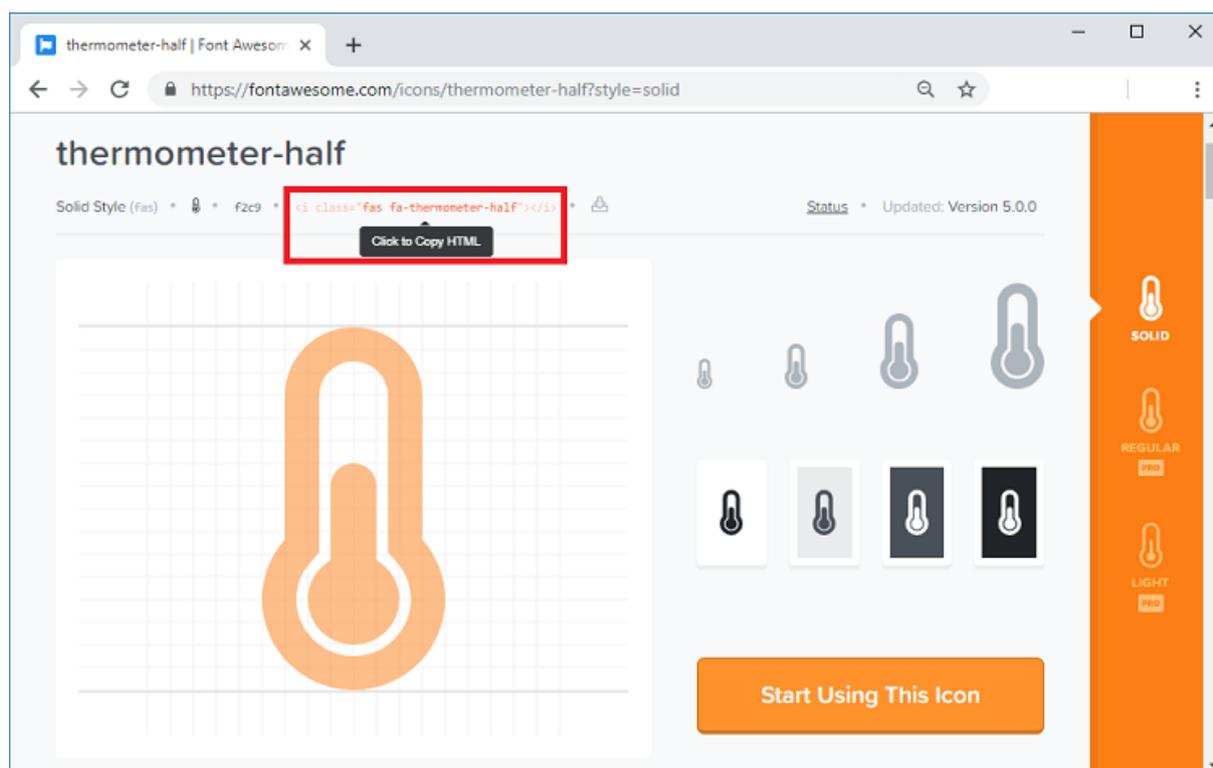


Search the icon you're looking for. For example, "thermometer".



Click the desired icon. Then, you just need to copy the HTML text provided.

```
<i class="fas fa-thermometer-half">
```



To choose the color, you just need to pass the `style` parameter with the color in hexadecimal, as follows:

```
<i class="fas fa-tint" style="color:#00add6;"></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">%</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 on 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", "/humidity", true);
xhttp.send();
}, 10000 ) ;
```

When it receives that value, 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 remove 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.

```
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.

```
WiFi.begin(ssid, password);  
while (WiFi.status() != WL_CONNECTED) {  
    delay(1000);  
    Serial.println("Connecting to WiFi..");  
}
```

Finally, add the next lines of code to handle the web server.

```
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()`.

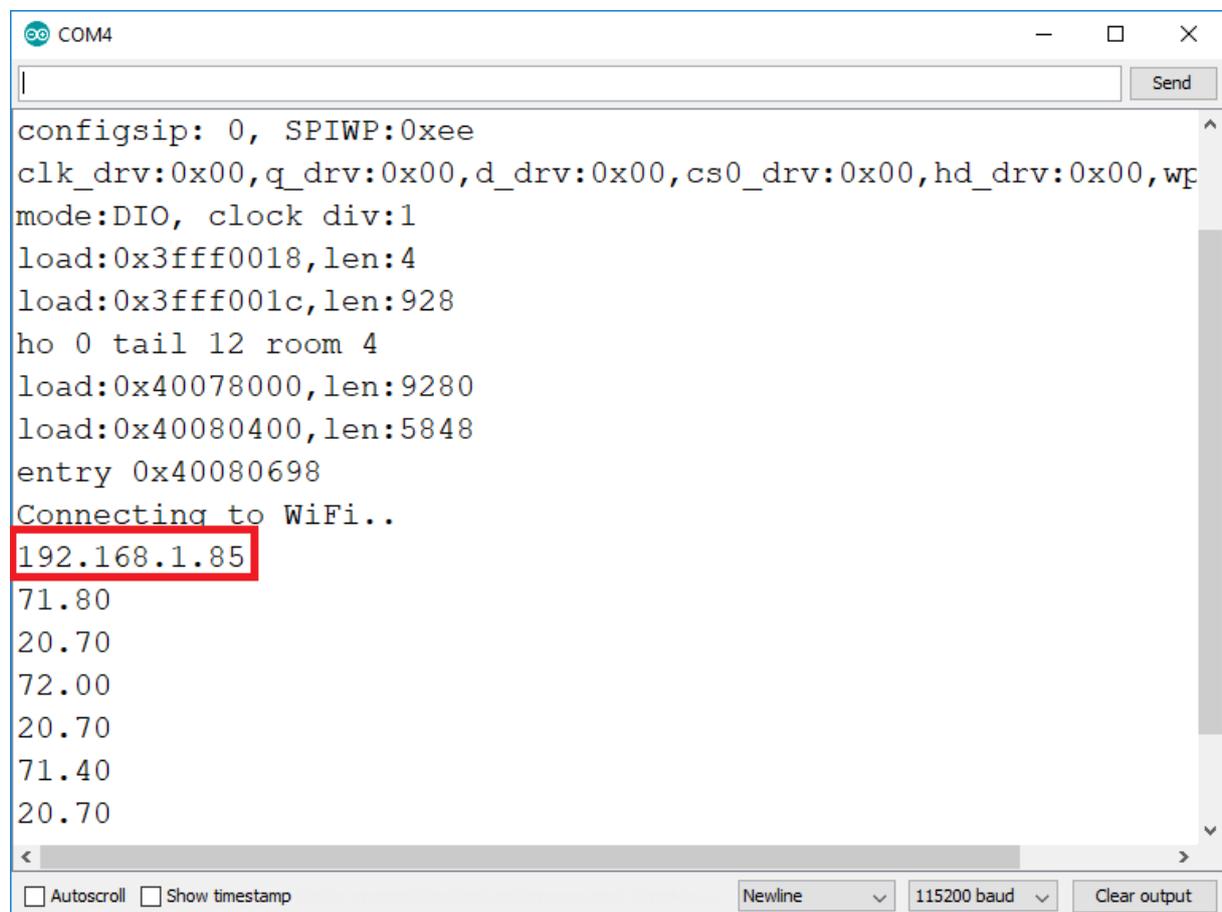
```
void loop() {  
}
```

That's pretty much how the code works.

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 in the serial monitor.



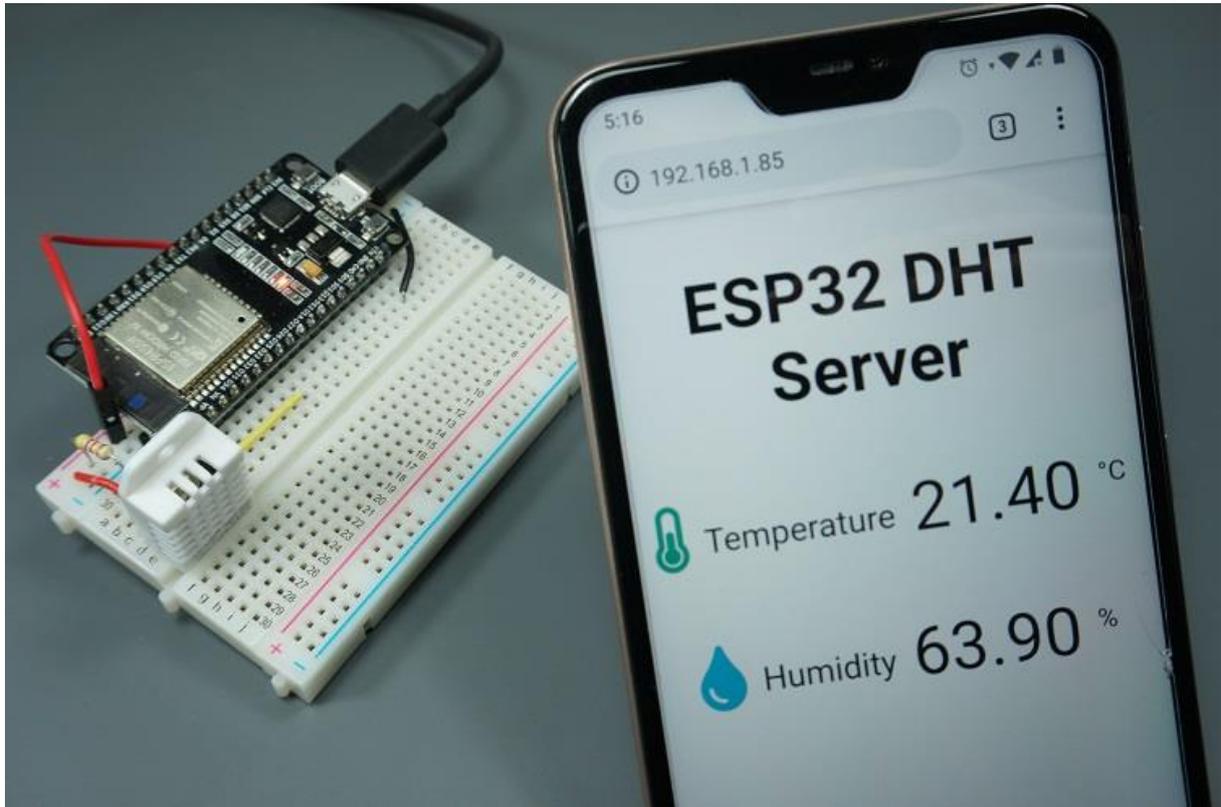
The screenshot shows the Serial Monitor window for COM4. The output text is as follows:

```
configsip: 0, SPIWP:0xee  
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp  
mode:DIO, clock div:1  
load:0x3fff0018,len:4  
load:0x3fff001c,len:928  
ho 0 tail 12 room 4  
load:0x40078000,len:9280  
load:0x40080400,len:5848  
entry 0x40080698  
Connecting to WiFi..  
192.168.1.85  
71.80  
20.70  
72.00  
20.70  
71.40  
20.70
```

The IP address `192.168.1.85` is highlighted with a red box. At the bottom of the window, the settings are: Autoscroll (unchecked), Show timestamp (unchecked), Newline (dropdown), 115200 baud (dropdown), and Clear output (button).

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

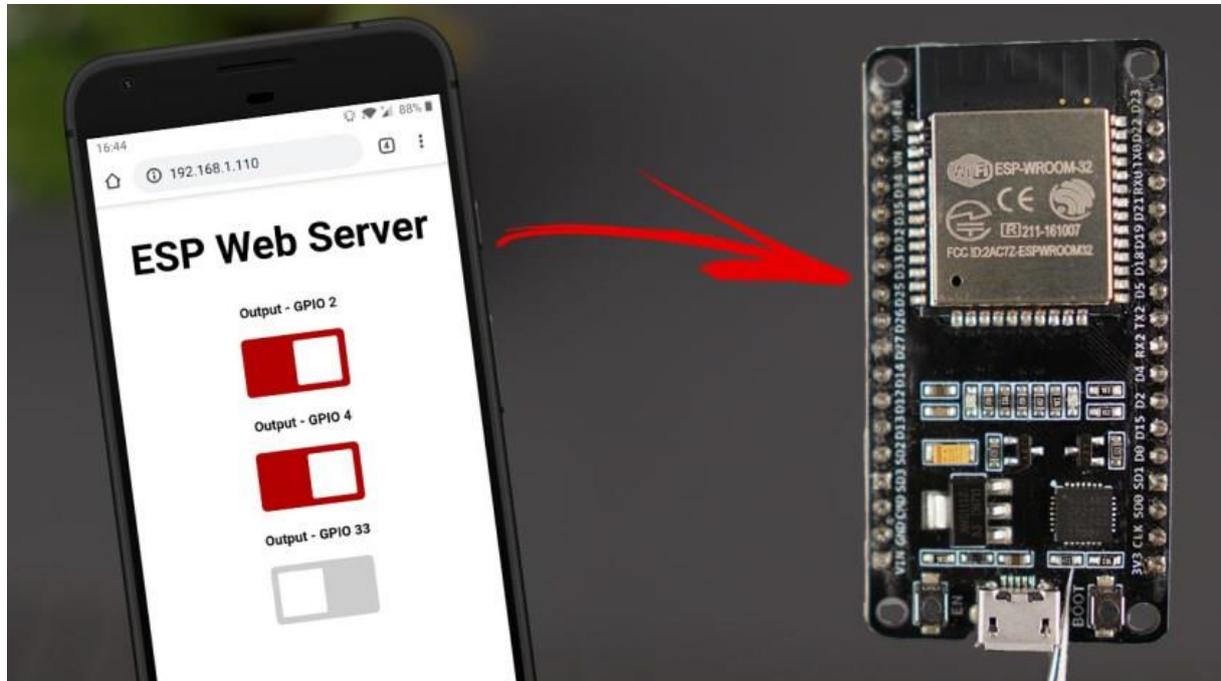
In this unit, we've provided a different way to build a web server using the [ESPAsyncWebServer](#) library.

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 the 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 a separate file that you then refer on the code. See this unit: [Build an ESP32 Web Server using Files from Filesystem \(SPIFFS\)](#)

Unit 12 – 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.

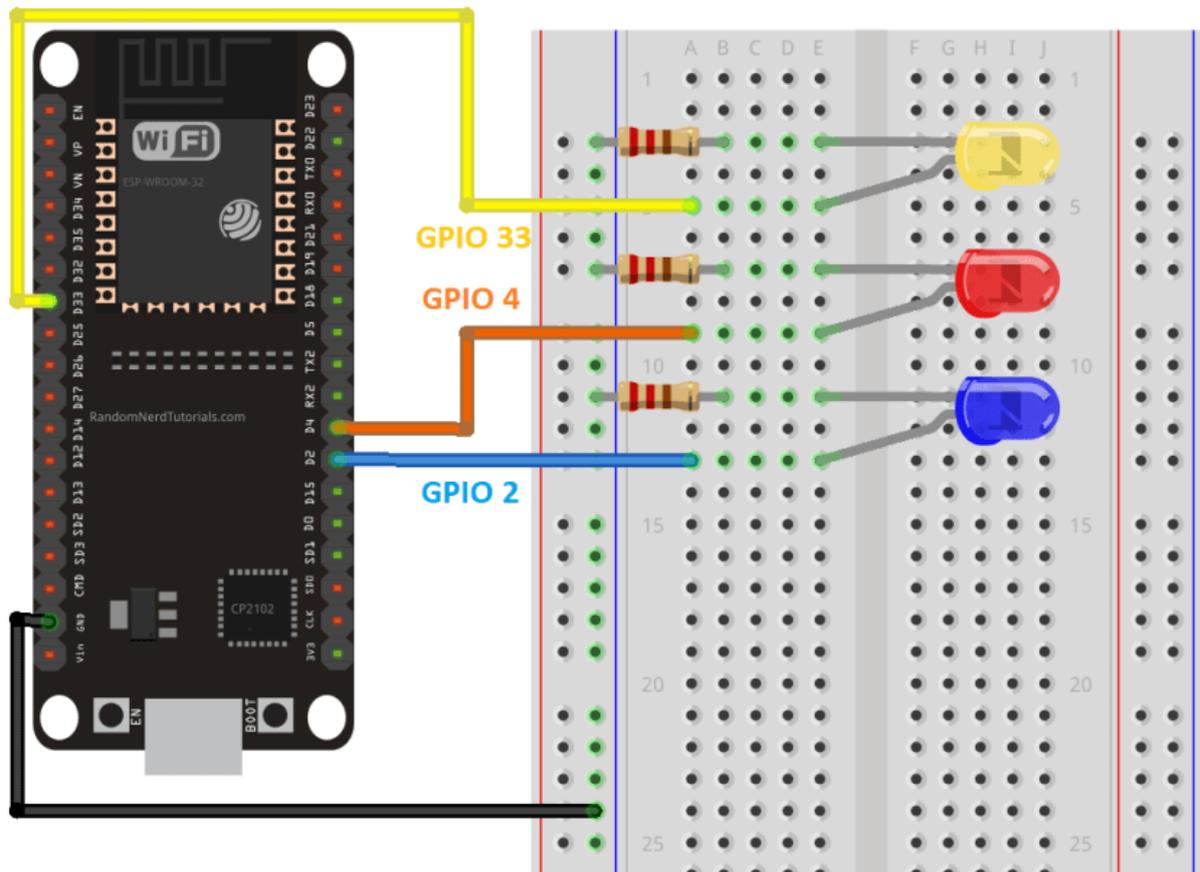
Parts Required

In this tutorial we'll control three outputs. As an example, we'll control LEDs. So, you need the following parts:

- [ESP32](#)
- 3x [LEDs](#)
- 3x [220 Ohm Resistor](#)
- [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

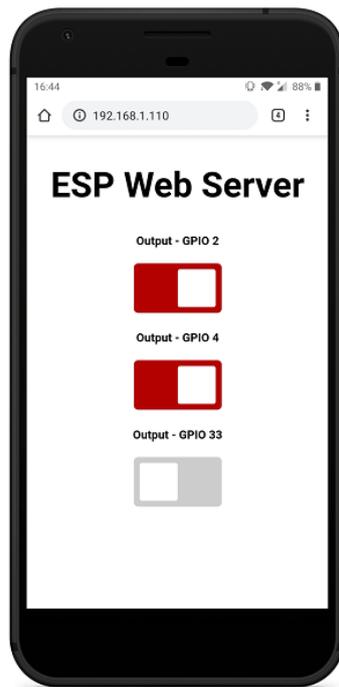
To build the web server you need to install the following libraries. If you've followed previous Units, you should have already installed them. Otherwise, click the links below to download the libraries.

- [ESPAsyncWebServer](#)
- [AsyncTCP](#)

These libraries aren't available to install through the Arduino Library Manager, so you need to copy the library files to the Arduino Installation Libraries folder. Alternatively, in your Arduino IDE, you can 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.

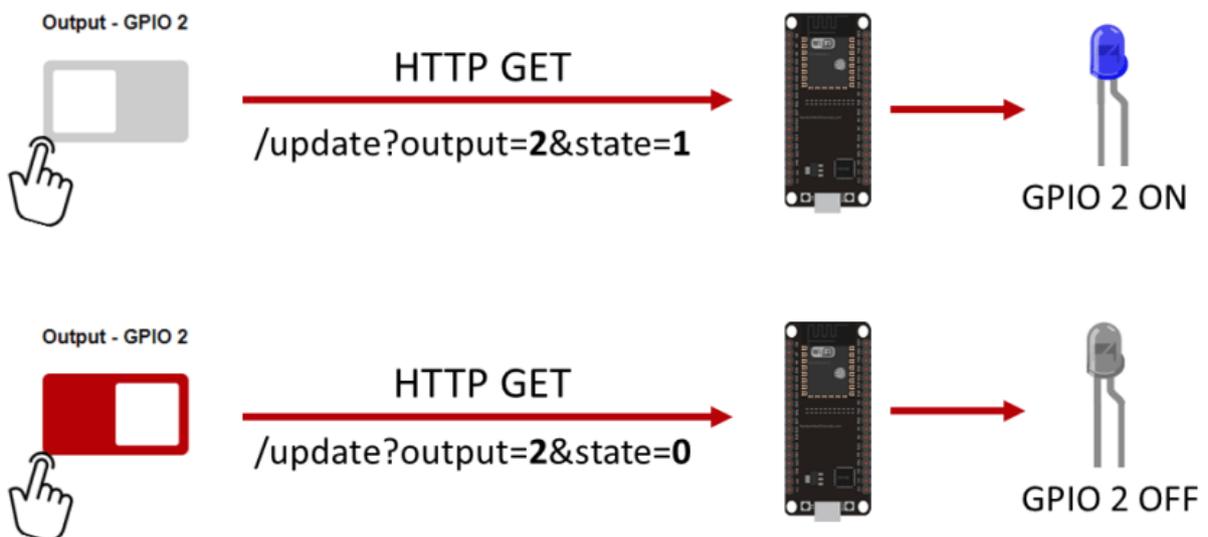


The web server contains one heading "ESP Web Server" and three buttons (toggle switches) to control three outputs. Each slider button has a label indicating the GPIO output pin. You can easily remove/add more outputs.

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.

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP Async Web Server/ESP Async Web Server.ino>

```
// 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 the Code Works

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.

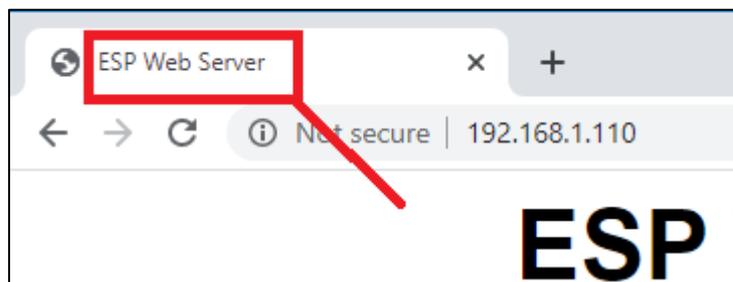
```
AsyncWebServer server(80);
```

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%`. This placeholder will then be replaced with the actual HTML text to build the buttons with the right states, when the web page is loaded.

```
%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.

```
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;
  }
}
```

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\"><input
type=\"checkbox\"      onchange=\"toggleCheckbox(this)\"      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 and empty field `""` if the GPIO is off.

```
String outputState(int output){
  if(digitalRead(output)){
    return "checked";
  }
  else {
    return "";
  }
}
```

So, the HTML text for GPIO 2 when it is on, would be:

```
<h4>Output - GPIO 2</h4>
<label class="switch">
<input type="checkbox"      onchange="toggleCheckbox(this)"      id="2"
checked><span class="slider"></span>
</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:

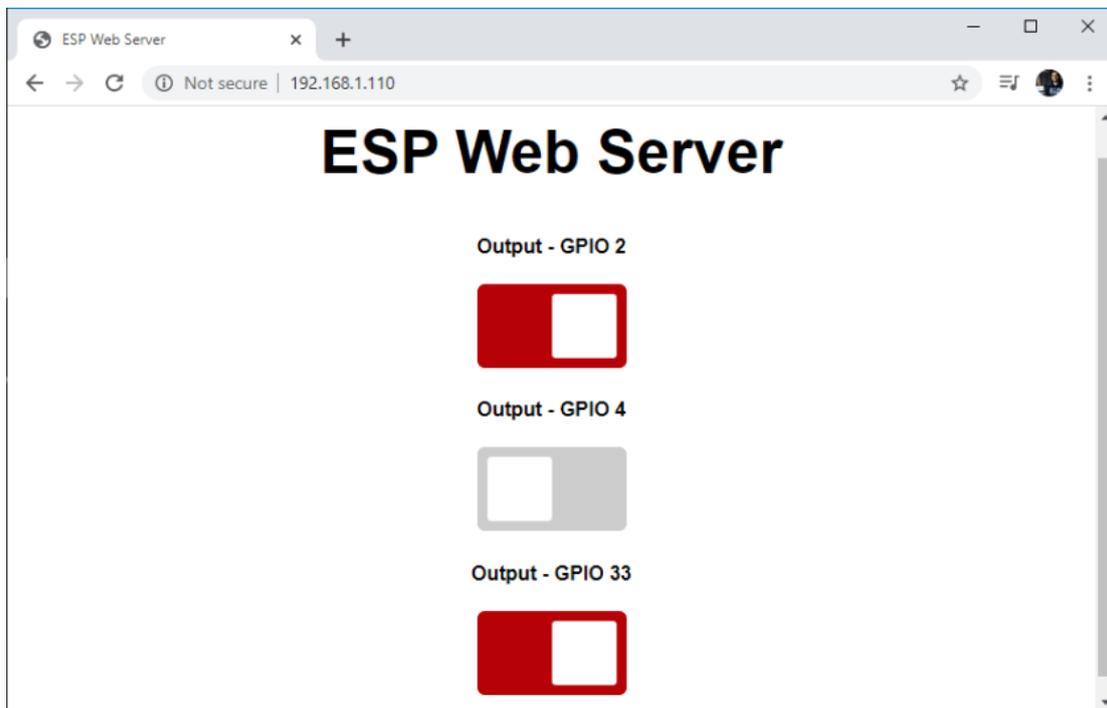
```
server.on("/update", HTTP_GET, [] (AsyncWebServerRequest *request) {
  String inputMessage1;
  String inputMessage2;
  // GET input1 value on
  <ESP_IP>/update?output=<inputMessage1>&state=<inputMessage2>
  if (request->hasParam(PARAM_INPUT_1) && request->hasParam(PARAM_INPUT_2))
  {
    inputMessage1 = 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

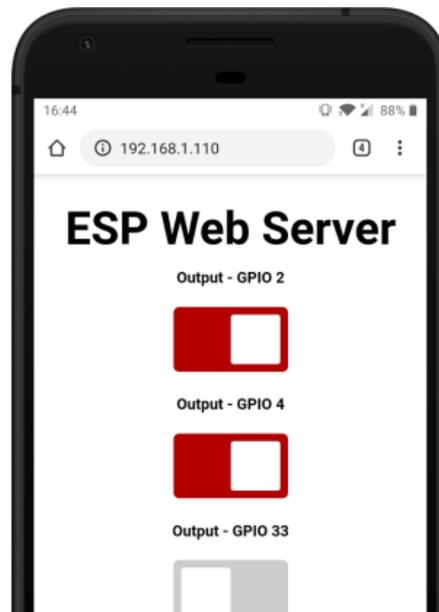
After uploading the code to your ESP32, open the Serial Monitor at a baud rate of 115200. Press the on-board RST/EN button. You should get its IP address. Open a browser and type the ESP IP address. You'll get access to a similar web page.



Press the toggle buttons to control the ESP32 GPIOs. At the same time, you should get the following messages in the Serial Monitor to help you debug your code and see what is happening on the background.

```
COM3
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
entry 0x400806b8
Connecting to WiFi..
192.168.1.110
GPIO: 2 - Set to: 1
GPIO: 33 - Set to: 1
GPIO: 33 - Set to: 0
GPIO: 33 - Set to: 1
GPIO: 33 - Set to: 0
GPIO: 4 - Set to: 1
GPIO: 4 - Set to: 0
GPIO: 2 - Set to: 0
```

You can also access the web server from a browser in 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

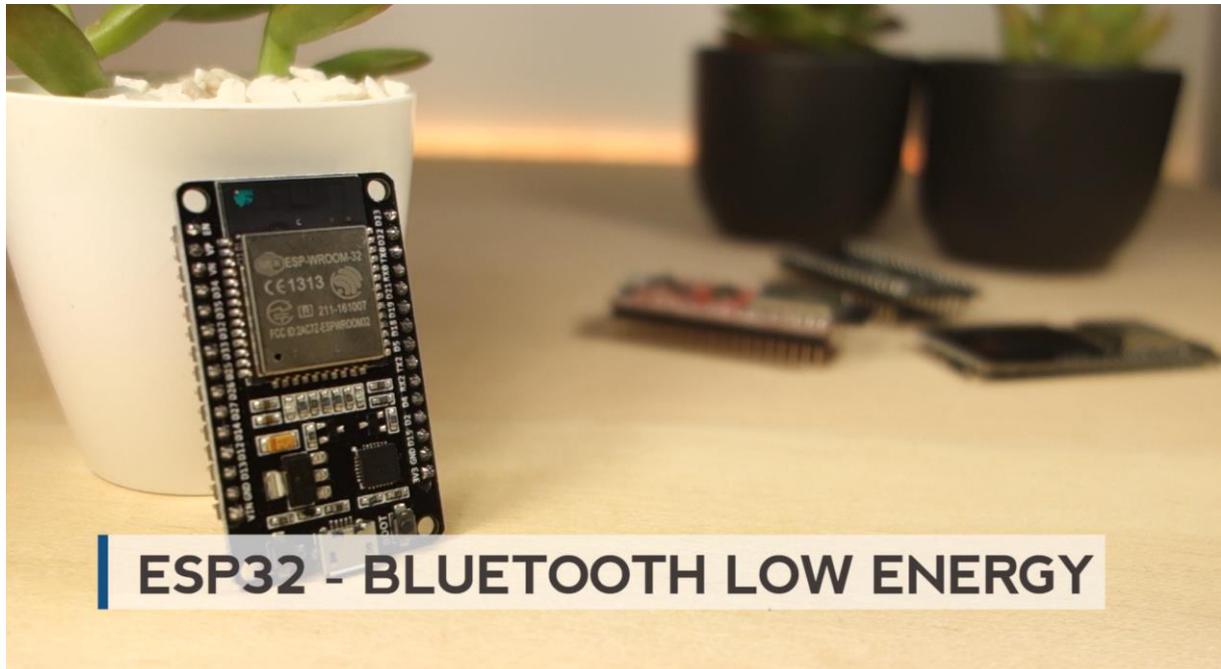
In this tutorial you've learned how to create an asynchronous web server with the ESP32 to control its outputs using toggle switches. Whenever you open the web page, it shows the updated GPIO states.



MODULE 5

ESP32 Bluetooth

Unit 1 - ESP32 Bluetooth Low Energy (BLE) – Introduction



In this Unit, we're going to explore what's BLE (which stands for Bluetooth Low Energy) and its applications. The ESP32 chip comes not only with Wi-Fi, but it also has both Bluetooth and Bluetooth Low Energy (BLE).

In this Unit we'll cover:

- The Basics of Bluetooth Low Energy (BLE)
- BLE Terminology: UUID, Service, Characteristic, and Properties
- Server and Client interaction

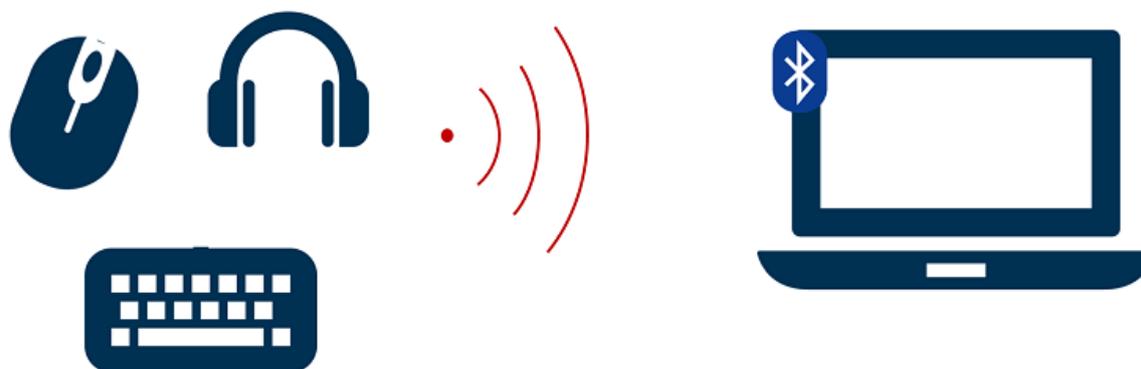
What is Bluetooth?

Bluetooth is a wireless technology standard for exchanging data over short distances. Just like Wi-Fi, Bluetooth also operates at 2.4GHz.



Bluetooth is used in many different applications that require wireless communication and control, like:

- Transmitting audio to headphones or car
- Communication between peripherals and a PC
- Transferring data between bluetooth enable devices
- And many other applications



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, is a power-conserving variant of Bluetooth. BLE's primary application is short distance transmission of small amounts of data and it's aimed at very low power applications running off a coin cell.



Unlike Bluetooth that is always on, BLE remains in sleep mode constantly except for when a connection is initiated. This makes it consume very low power. 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 exchange small amounts of data periodically, like in healthcare, fitness, tracking, beacons, security, and home automation industries.



You can compare in more detail the differences between [Bluetooth and Bluetooth Low Energy](#) or simply take a look at the table below.

	Bluetooth Low Energy (LE)	Bluetooth Basic Rate/ Enhanced Data Rate (BR/EDR)
Optimized For...	Short burst data transmission	Continuous data streaming
Frequency Band	2.4 GHz (2.402 GHz to 2.480 GHz)	2.4 GHz (2.402 GHz to 2.480 GHz)
Channels	40 channels with 2 MHz spacing (3 advertising channels/37 data channels)	79 channels with 1 MHz spacing
Channel Usage	Adaptive Frequency Hopping (AFH) 1600 hops/sec	Adaptive Frequency Hopping (AFH) 1600 hops/sec
Modulation	GFSK	GFSK, $\pi/4$ DQPSK, 8DPSK
Power Consumption	~0.01x to 0.5x of reference (depending on use case)	1 (reference value)
Data Rate	LE 2M PHY: 2 Mb/s LE 1M PHY: 1 Mb/s LE Coded PHY (S=2): 500 Kb/s LE Coded PHY (S=8): 125 Kb/s	EDR PHY (8DPSK): 3 Mb/s EDR PHY ($\pi/4$ DQPSK): 2 Mb/s BR PHY (GFSK): 1 Mb/s
Max Tx Power*	Class 1: 100 mW (+20 dBm) Class 1.5: 10 mW (+10 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)	Class 1: 100 mW (+20 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)
Network Topologies	Point-to-Point (including piconet) Broadcast Mesh	Point-to-Point (including piconet)

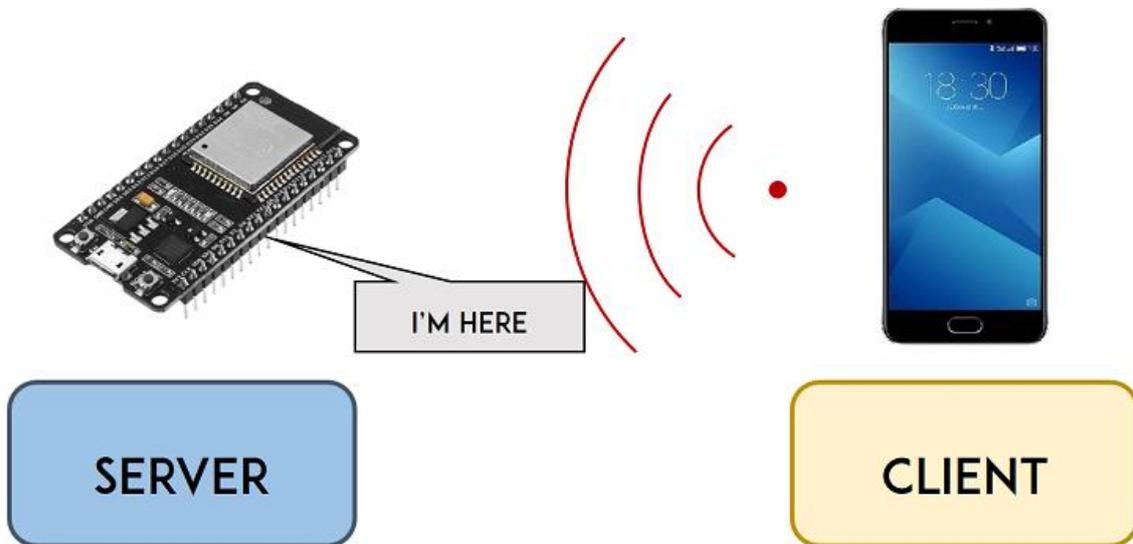
Source: [Bluetooth.com](https://www.bluetooth.com)

Bluetooth Server and Client

With Bluetooth Low Energy, there are two types of devices: the server and the client. For this example, the ESP32 acts as a server and the smartphone as a client.

Note: the ESP32 can act as a client or a server. The same can be said of the smartphone.

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



There are also other possible scenarios in which BLE can be used:

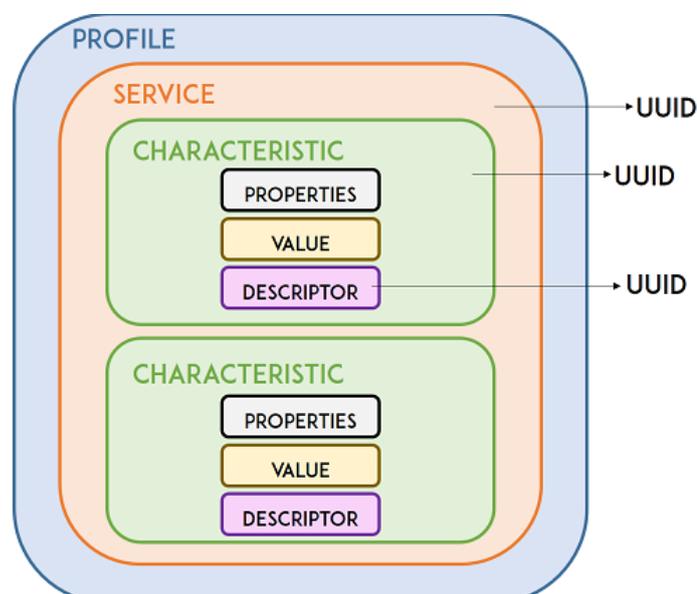
- **Broadcast mode:** the server transmits data to many clients that are connected;
- **Mesh network:** all the devices are connected, this is a many to many connection.

Even though the broadcast and mesh network setups are possible to implement, they were developed very recently, so there aren't many examples implemented for the ESP32 at this moment. So, we'll be using the point-to-point communication for most of our examples.

Now, let's take a look at some important terms when it comes to BLE.

GATT

GATT stands for Generic Attributes and it defines an hierarchical data structure that is exposed to connected BLE devices. This means that GATT defines the way that two BLE devices send and receive standard messages.



The top level of the hierarchy is a profile, which is composed of one or more services. Every service contains at least one characteristic, or can also reference other services.

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

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

UUID

Each service, characteristic and descriptor have an UUID (Universally Unique Identifier). An UUID is a unique 128-bit (16 bytes) number. For example:

```
55072829-bc9e-4c53-938a-74a6d4c78776
```

There are shortened UUIDs for all types, services, and profiles specified in the [SIG \(Bluetooth Special Interest Group\)](#).

GATT Services

Services are collections of characteristics and relationships to other services that encapsulate the behavior of part of a device.

All Service Assigned Numbers values on this page are normative. All other materials contained on this page is informative only. Authoritative compliance information is contained in the [applicable Bluetooth® specification](#).

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

Source: [Bluetooth.com](https://www.bluetooth.com)

But if your application needs its own UUID, you can generate them using this [UUID generator website](#).

In summary, the UUID is used for uniquely identifying information. For example, it can identify a particular service provided by a Bluetooth device

Battery Service

For example, let's take a look at the Battery service. The battery service is used in most battery powered BLE devices to indicate the current battery level in percentage. If the BLE device was developed properly, it uses a standardized service called "Battery Level".

Service Characteristics

Overview	Properties	Security	Descriptors																																														
<p>Name: Battery Level</p> <p>Description: The Battery Level characteristic is read using the GATT Read Characteristic Value sub-procedure and returns the current battery level as a percentage from 0% to 100%; 0% represents a battery that is fully discharged, 100% represents a battery that is fully charged.</p> <p>Type: org.bluetooth.characteristic.battery_level</p> <p>Requirement: Mandatory</p>	<table border="1"> <thead> <tr> <th>Property</th> <th>Requirement</th> </tr> </thead> <tbody> <tr><td>Read</td><td>Mandatory</td></tr> <tr><td>Write</td><td>Excluded</td></tr> <tr><td>WriteWithoutResponse</td><td>Excluded</td></tr> <tr><td>SignedWrite</td><td>Excluded</td></tr> <tr><td>Notify</td><td>Optional</td></tr> <tr><td>Indicate</td><td>Excluded</td></tr> <tr><td>WritableAuxiliaries</td><td>Excluded</td></tr> <tr><td>Broadcast</td><td>Excluded</td></tr> <tr><td>ExtendedProperties</td><td></td></tr> </tbody> </table>	Property	Requirement	Read	Mandatory	Write	Excluded	WriteWithoutResponse	Excluded	SignedWrite	Excluded	Notify	Optional	Indicate	Excluded	WritableAuxiliaries	Excluded	Broadcast	Excluded	ExtendedProperties		None	<table border="1"> <thead> <tr> <th>Overview</th> <th>Permissions</th> </tr> </thead> <tbody> <tr> <td>Name: Characteristic Presentation Format</td> <td><table border="1"><thead><tr><th>Permission</th><th>Requirement</th></tr></thead><tbody><tr><td>Read</td><td>Mandatory</td></tr><tr><td>Write</td><td>Excluded</td></tr></tbody></table></td> </tr> <tr> <td>Type: org.bluetooth.descriptor.gatt.characteristic_presentation_format</td> <td></td> </tr> <tr> <td>Requirement: if_multiple_service_instances</td> <td></td> </tr> <tr> <td>Name: Client Characteristic Configuration</td> <td><table border="1"><thead><tr><th>Permission</th><th>Requirement</th></tr></thead><tbody><tr><td>Read</td><td>Mandatory</td></tr><tr><td>Write</td><td>Mandatory</td></tr></tbody></table></td> </tr> <tr> <td>Type: org.bluetooth.descriptor.gatt.client_characteristic_configuration</td> <td></td> </tr> <tr> <td>Requirement: if_notify_or_indicate_supported</td> <td></td> </tr> </tbody> </table>	Overview	Permissions	Name: Characteristic Presentation Format	<table border="1"><thead><tr><th>Permission</th><th>Requirement</th></tr></thead><tbody><tr><td>Read</td><td>Mandatory</td></tr><tr><td>Write</td><td>Excluded</td></tr></tbody></table>	Permission	Requirement	Read	Mandatory	Write	Excluded	Type: org.bluetooth.descriptor.gatt.characteristic_presentation_format		Requirement: if_multiple_service_instances		Name: Client Characteristic Configuration	<table border="1"><thead><tr><th>Permission</th><th>Requirement</th></tr></thead><tbody><tr><td>Read</td><td>Mandatory</td></tr><tr><td>Write</td><td>Mandatory</td></tr></tbody></table>	Permission	Requirement	Read	Mandatory	Write	Mandatory	Type: org.bluetooth.descriptor.gatt.client_characteristic_configuration		Requirement: if_notify_or_indicate_supported	
Property	Requirement																																																
Read	Mandatory																																																
Write	Excluded																																																
WriteWithoutResponse	Excluded																																																
SignedWrite	Excluded																																																
Notify	Optional																																																
Indicate	Excluded																																																
WritableAuxiliaries	Excluded																																																
Broadcast	Excluded																																																
ExtendedProperties																																																	
Overview	Permissions																																																
Name: Characteristic Presentation Format	<table border="1"><thead><tr><th>Permission</th><th>Requirement</th></tr></thead><tbody><tr><td>Read</td><td>Mandatory</td></tr><tr><td>Write</td><td>Excluded</td></tr></tbody></table>	Permission	Requirement	Read	Mandatory	Write	Excluded																																										
Permission	Requirement																																																
Read	Mandatory																																																
Write	Excluded																																																
Type: org.bluetooth.descriptor.gatt.characteristic_presentation_format																																																	
Requirement: if_multiple_service_instances																																																	
Name: Client Characteristic Configuration	<table border="1"><thead><tr><th>Permission</th><th>Requirement</th></tr></thead><tbody><tr><td>Read</td><td>Mandatory</td></tr><tr><td>Write</td><td>Mandatory</td></tr></tbody></table>	Permission	Requirement	Read	Mandatory	Write	Mandatory																																										
Permission	Requirement																																																
Read	Mandatory																																																
Write	Mandatory																																																
Type: org.bluetooth.descriptor.gatt.client_characteristic_configuration																																																	
Requirement: if_notify_or_indicate_supported																																																	

Source: [Bluetooth.com](#)

This enables the applications that are connected to that BLE device to instantly know the battery level regardless of the manufacturer, because they are using the unique service ID that refers to the battery level.

Battery Level characteristic

You can explore all the other possible [characteristics that the Battery Service](#) can have. In this case, the battery service provides the "Battery Level" characteristic that returns the current battery level as a percentage from 0% to 100%.

So, if you have a BLE device with battery – that is a server with the battery service being advertised with the characteristic battery level – the client, your smartphone that is connected to that device, can see the battery level in percentage, because it's a well known service with a standard characteristic.

Battery Level characteristic properties

As we've seen before, each service can have one or more characteristics. The characteristic is a data value transferred between client and server. A characteristic can have different properties. For example the battery level is READ only and that's a mandatory requirement.

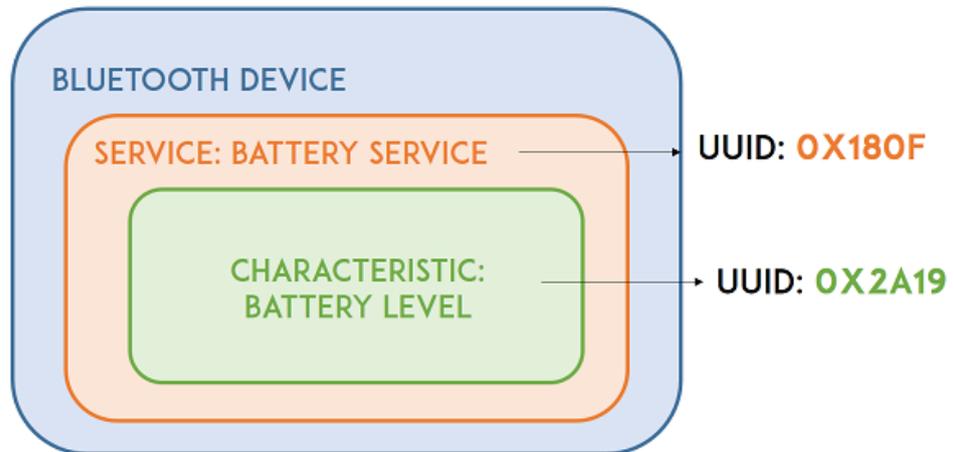
You can also have NOTIFY activated or not (it is optional), so that the BLE devices reports the battery level every X number of seconds, for example.

Property	Requirement
Read	Mandatory
Write	Excluded
WriteWithoutResponse	Excluded
SignedWrite	Excluded
Notify	Optional
Indicate	Excluded
WritableAuxiliaries	Excluded
Broadcast	Excluded
ExtendedProperties	

Depending on the characteristic type, it will have different properties and we'll explore that subject in future Units.

Bluetooth device overview

Here's a representation of a diagram for a battery service with the "Battery Level" characteristic.



Understanding this hierarchy is important, because it will make it easy to understand how to use the BLE and write your applications.

You can explore [this page](#) that provides additional information about standard services. It's also possible to create other services that are not contained in this list.

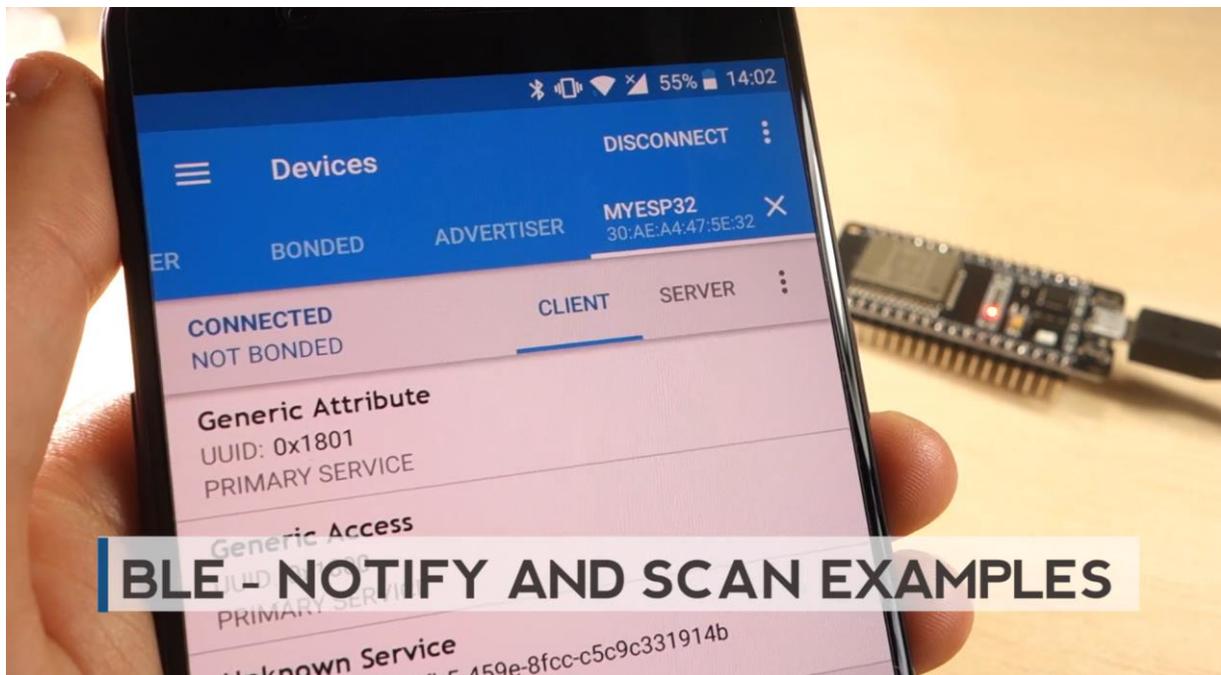
Wrapping Up

That's it for now, this is just the basic theory behind a BLE device and the interactions between a client and server and some of the standards that you need to pay attention when it comes to BLE. It is important to refer that we've just scratched the surface of BLE and there are many other concepts to explore. We just covered some of the features that are relevant for our projects.

We strongly encourage you to read the [BLE Wikipedia page](#) that contains more information about BLE. Don't forget to scroll down to the bottom of the page and take a look at the references section for a more in-depth reading.

You should also use the [Bluetooth.com website](#) as a reference, because it contains the standard services and UUIDs that you should use in BLE devices and applications.

Unit 2 - Bluetooth Low Energy – Notify and Scan



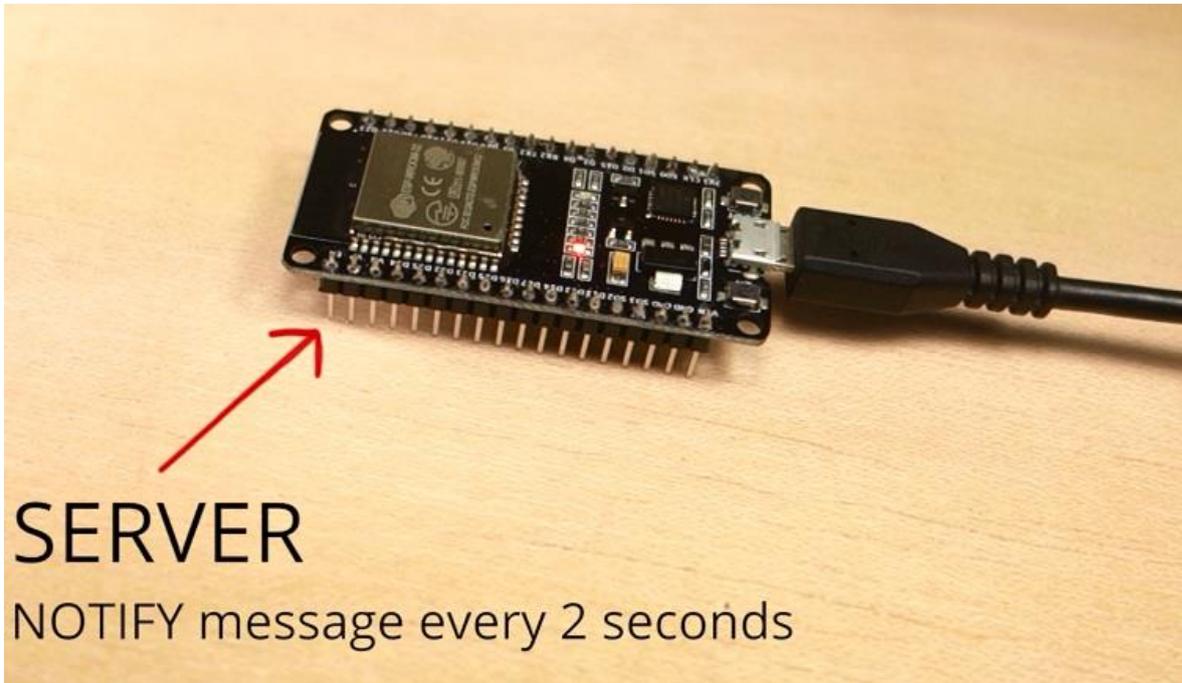
In this unit, we're going to take a look at a few examples that come with the ESP32 add-on for the Arduino IDE and explore some Bluetooth Low Energy examples.

Having the ESP32 add-on installed, open the Arduino IDE. Let's test an example that comes with the ESP32 BLE Arduino library.

Note: if you haven't already, you can read the "[Installing the ESP32 Board in Arduino IDE](#)" to learn how to install the ESP32 add-on.

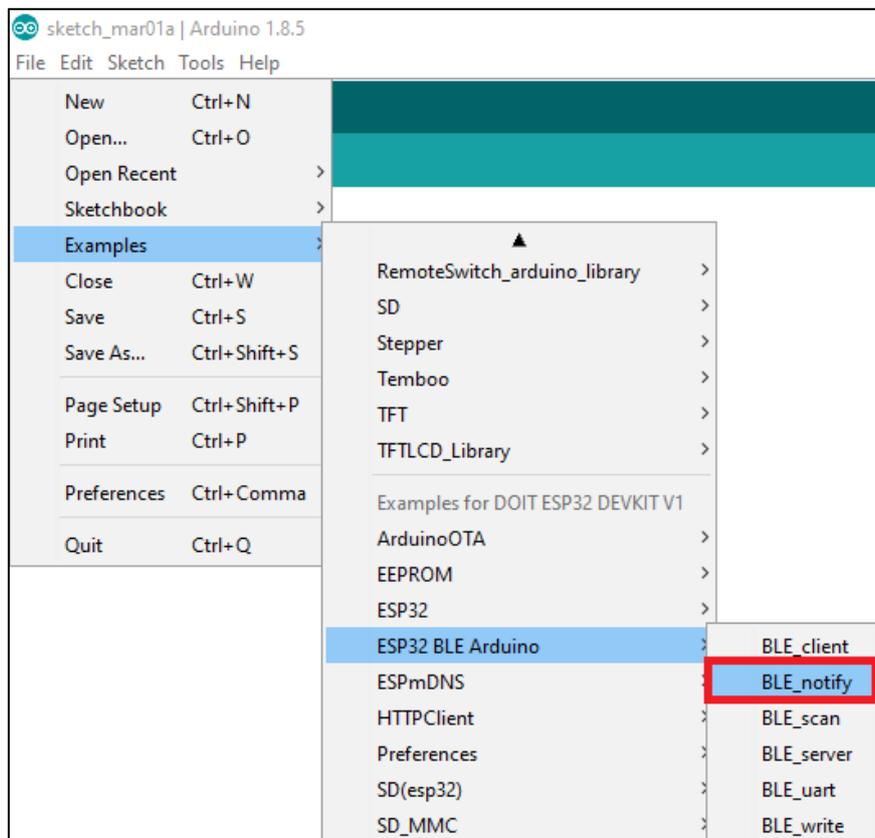
Notify Example

For this first example, you'll set the ESP32 as a server that sends a notify message every 2 seconds when a client is connected. We'll be using the examples that come with the [ESP32 BLE library for Arduino IDE](#).



Note: the ESP32 can be configured as a server or as a client.

Go to **Tools** ▶ **Board** and select your ESP32 board. Then, go to **File** ▶ **Examples** ▶ **ESP32 BLE Arduino** ▶ **BLE_notify**.



This new sketch should open:

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/BLE_Examples/BLE_notify/BLE_notify.ino

```

/*
  Video: https://www.youtube.com/watch?v=oCMOYS71NIU
  Based on Neil Kolban example for IDF: https://github.com/nkolban/esp32-snippets/blob/master/cpp\_utils/tests/BLE%20Tests/SampleNotify.cpp
  Ported to Arduino ESP32 by Evandro Copercini
  Create a BLE server that, once we receive a connection, will send
  periodic notifications.
  The service advertises itself as: 4fafc201-1fb5-459e-8fcc-c5c9c331914b
  And has a characteristic of: beb5483e-36e1-4688-b7f5-ea07361b26a8
  The design of creating the BLE server is:
  1. Create a BLE Server
  2. Create a BLE Service
  3. Create a BLE Characteristic on the Service
  4. Create a BLE Descriptor on the characteristic
  5. Start the service.
  6. Start advertising.
  A connect handler associated with the server starts a background task
  that performs notification
  every couple of seconds.
*/
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>

BLECharacteristic *pCharacteristic;
bool deviceConnected = false;
uint8_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("MyESP32");

  // 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,

```

```

        BLECharacteristic::PROPERTY_READ |
        BLECharacteristic::PROPERTY_WRITE |
        BLECharacteristic::PROPERTY_NOTIFY |
        BLECharacteristic::PROPERTY_INDICATE
    );

    //
    https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.descriptor.gatt.client_characteristic_configuration.xml
    // Create a BLE Descriptor
    pCharacteristic->addDescriptor(new BLE2902());

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

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

void loop() {

    if (deviceConnected) {
        Serial.printf("*** NOTIFY: %d ***\n", value);
        pCharacteristic->setValue(&value, 1);
        pCharacteristic->notify();
        //pCharacteristic->indicate();
        value++;
    }
    delay(2000);
}

```

How the Code Works

Let's take a look at this example.

It starts by importing the necessary libraries for the BLE capabilities.

```

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

```

The following line creates a pointer to a BLECharacteristic.

```
BLECharacteristic *pCharacteristic;
```

Next, you create a boolean variable to store if a client is connected to a server or not.

```
bool deviceConnected = false;
```

The value variable is used to set the Characteristic value.

```
uint8_t value = 0;
```

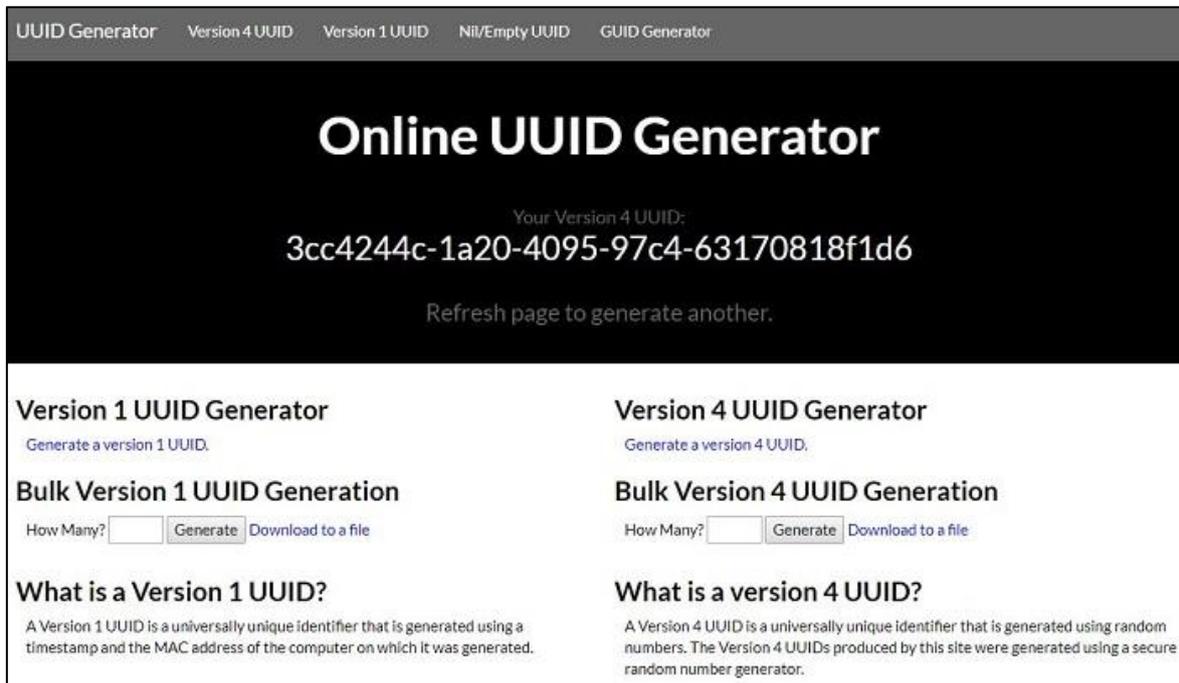
Then, you need to define a new service and characteristic with a generated UUID.

```

#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 "MyESP32". You can change this name to whatever you like.

```
// Create the BLE Device  
BLEDevice::init("MyESP32");
```

In the following line, you set the BLE device as a server.

```
BLEServer *pServer = BLEDevice::createServer();
```

The server has two callback functions assigned to it.

```
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 service for the server that has the UUID defined earlier.

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

Then, you set the characteristic for that service. As you can see, you also use the UUID defined earlier, and you need to pass as arguments the properties that this characteristic has. 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.

```
// Create a BLE Descriptor
pCharacteristic->addDescriptor(new BLE2902());
```

Finally, you can start the service, and the advertising, so other BLE devices can scan and find this BLE device.

```
// Start the service
pService->start();
// Start advertising
pServer->getAdvertising()->start();
```

loop()

In the `loop()`, you basically check if a device is connected or not. If a device is connected:

- It prints a message in the Serial Monitor with the current value;
- You set a new value to the characteristic;
- You notify the connected client;
- Finally, the value is incremented by one.

```
if (deviceConnected) {
    Serial.printf("*** NOTIFY: %d ***\n", value);
    pCharacteristic->setValue(&value, 1);
    pCharacteristic->notify();
    //pCharacteristic->indicate();
    value++;
}
```

This process is repeated every 2 seconds. So, in the first loop it sets the value to 1, then to 2, 3, and so on...

```
delay(2000);
```

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...

Prepare your Smartphone

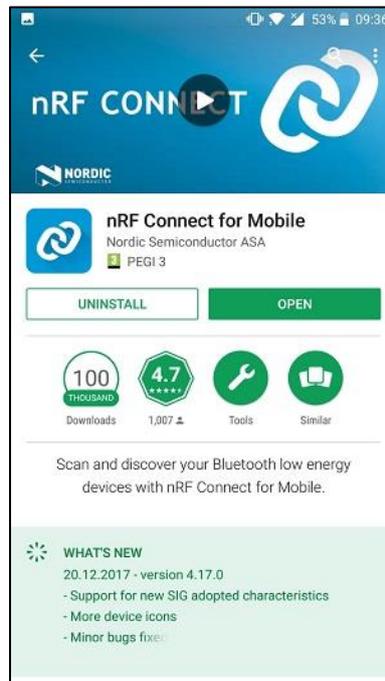
Most modern smartphones should have BLE capabilities. I'm currently using a [OnePlus 5](#), but most smartphones should also work. You can search for your smartphone specifications to check if it has BLE or not.



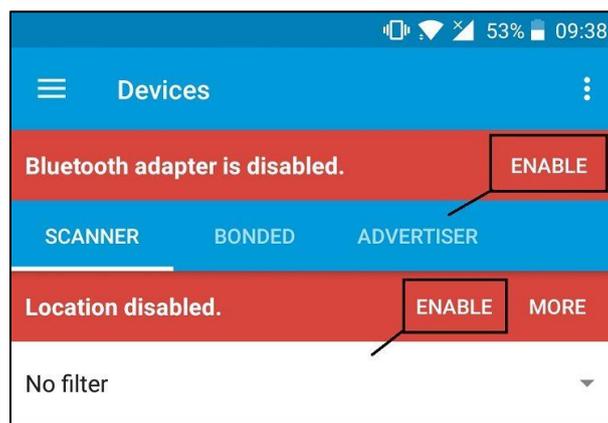
Note: the smartphone can act as a client or as a server. In this case, it will be the client that connects to the ESP32 BLE server.

Grab your smartphone...

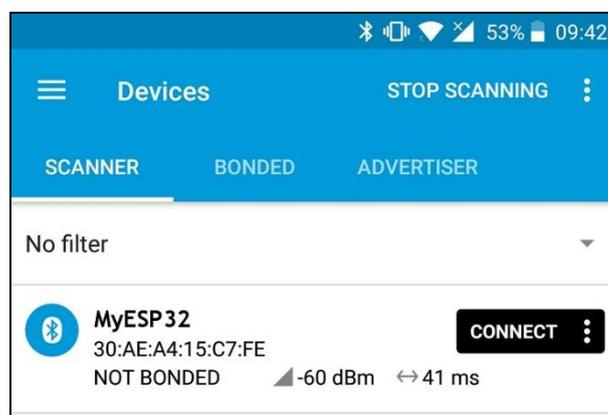
For our tests will be using a free app called nRF Connect for Mobile from Nordic, it works on [Android \(Google Play Store\)](#) and [iOS \(App Store\)](#). Go to Google Play Store or App Store and search for "nRF Connect for Mobile". Install the app and open it.



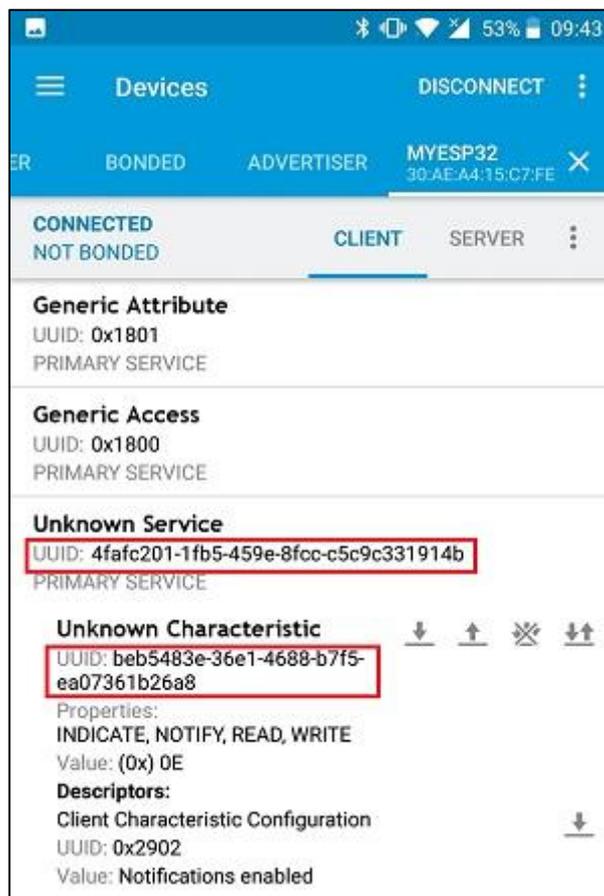
Don't forget go to the Bluetooth settings and enable Bluetooth adapter in your smartphone. You may also want to make it visible to other devices to test other sketches later on.



Once everything is ready in your smartphone and the ESP32 is running the BLE notify sketch, in the app, tap the scan button to scan for nearby devices, you should find an ESP32 with the name "MyESP32".



Click the “Connect” button. As you can see in the figure below, the ESP32 has a service with the UUID that you’ve defined earlier. If you tap the service, it expands the menu and shows the Characteristic with the UUID that you’ve also defined.



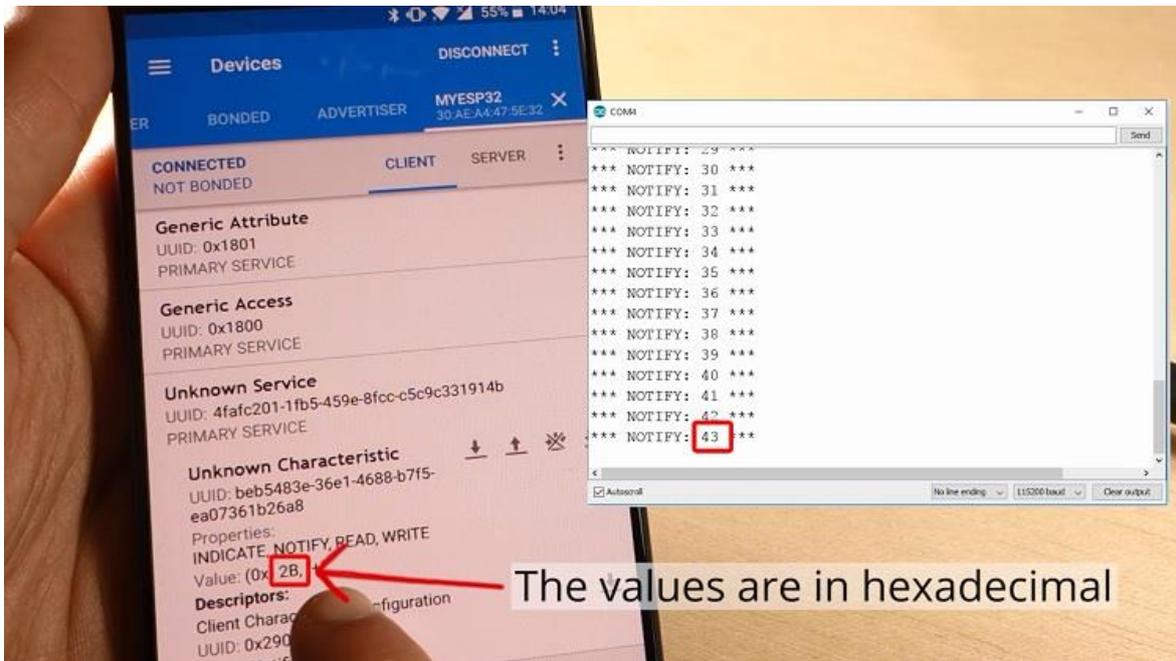
This characteristic has 4 properties: INDICATE, NOTIFY, READ, and WRITE. The four buttons highlighted in the figure below, allow you to read the current characteristic value, write a new characteristic value, enable the notify property or enable the indicate property.



In this example, we’re going to leave the notify property enabled, so the smartphone receives the new characteristic value every 2 seconds.

The message that is being printed in the Serial Monitor is also being received in the app. So, it’s working.

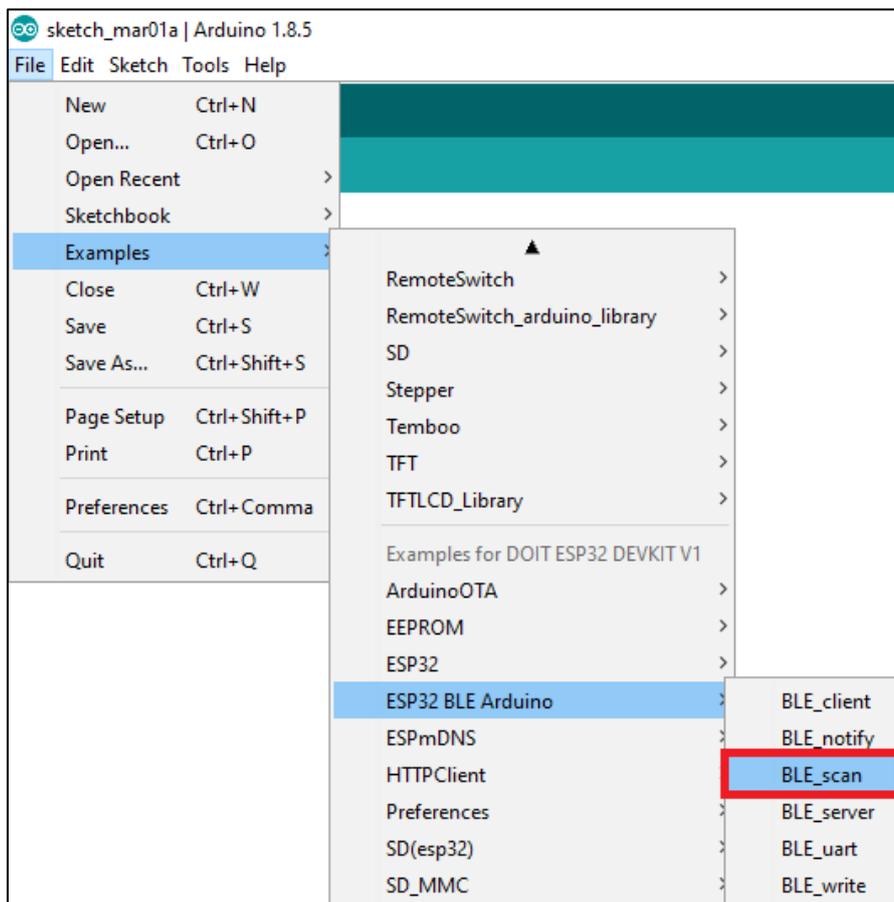
Note: the values received in the app are in hexadecimal.



Now, you can turn off your smartphone's Bluetooth and close the Nordic app.

Scan Example

Keep the ESP32 board running the BLE notify sketch. Grab another ESP32 board and connect it to your computer. Go to **File** ▶ **Examples** ▶ **ESP32 BLE Arduino** ▶ **BLE_scan**.



The following sketch scans for nearby devices and when it finds them, it displays their info in the Serial Monitor.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/BLE_Examples/BLE_scan/BLE_scan.ino

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

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

int scanTime = 30; //In seconds

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...");

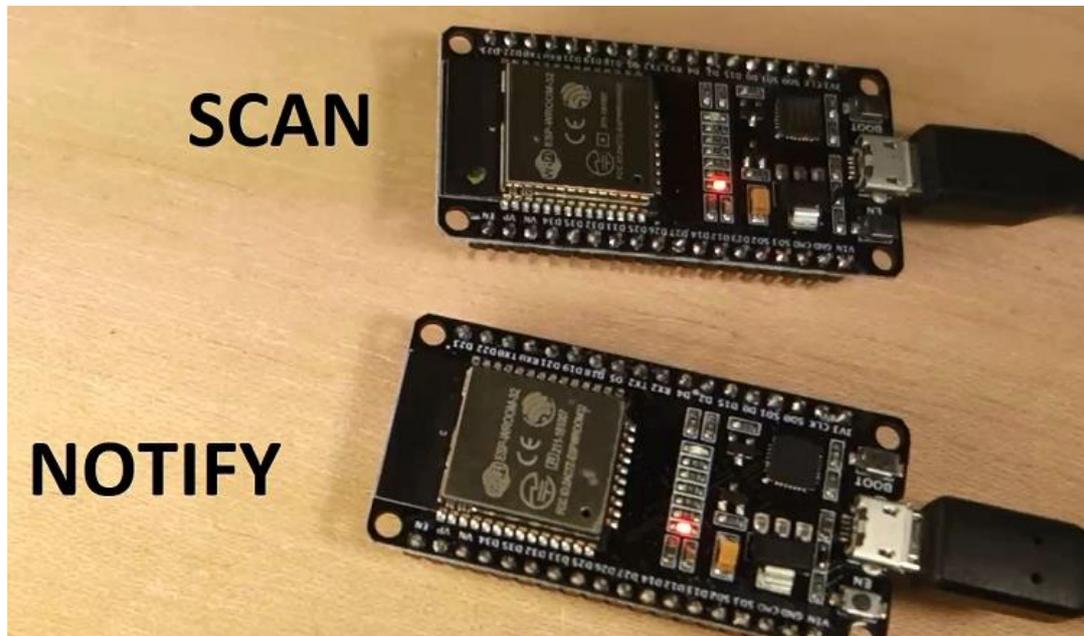
  BLEDevice::init("");
  BLEScan* pBLEScan = BLEDevice::getScan(); //create new scan
  pBLEScan->setAdvertisedDeviceCallbacks(new
MyAdvertisedDeviceCallbacks());
  pBLEScan->setActiveScan(true); //active scan uses more power, but get
results faster
  BLEScanResults foundDevices = pBLEScan->start(scanTime);
  Serial.print("Devices found: ");
  Serial.println(foundDevices.getCount());
  Serial.println("Scan done!");
}

void loop() {
  // put your main code here, to run repeatedly:
  delay(2000);
}
```

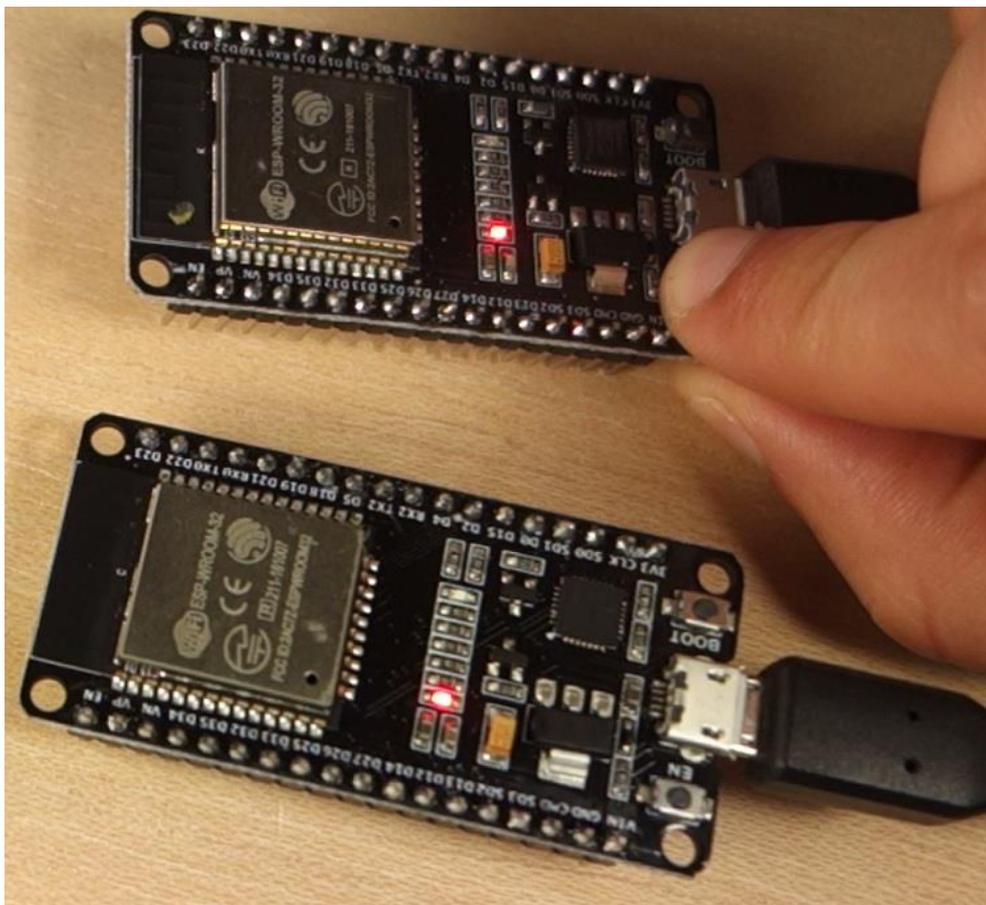
Make sure you have the right board and COM port selected for this new ESP32 (you might want to temporarily disconnect the other ESP32 from your computer, so you're sure that you're uploading the code to the right ESP32 board).

Once the code is uploaded and you should have the two ESP32 boards powered on:

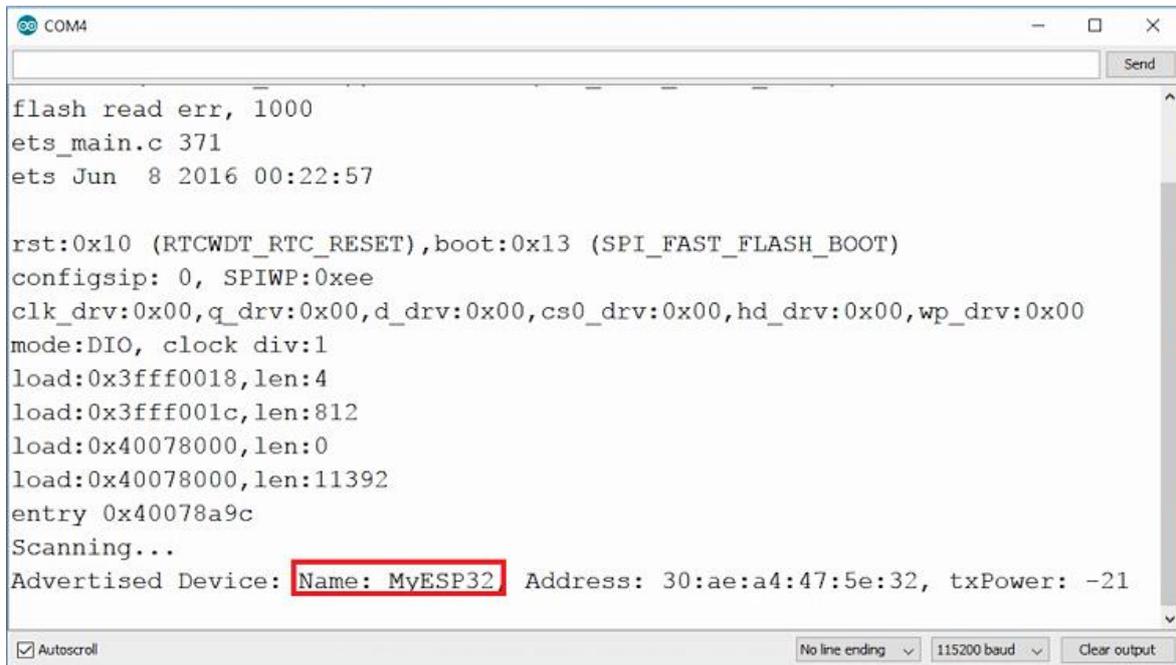
- 1) One ESP32 with the "Notify" sketch
- 2) Other with ESP32 "Scan" sketch



Go to the Serial Monitor with the ESP32 running the “Scan” example, press the ESP32 (with the “Scan” sketch) ENABLE button to restart and wait a few seconds while it scans.



After a few seconds, it should find a device called “MyESP32” – which is the ESP32 that is running the “Notify” sketch.



```
COM4
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

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
Scanning...
Advertised Device: Name: MyESP32 Address: 30:ae:a4:47:5e:32, txPower: -21

Autoscroll No line ending 115200 baud Clear output
```

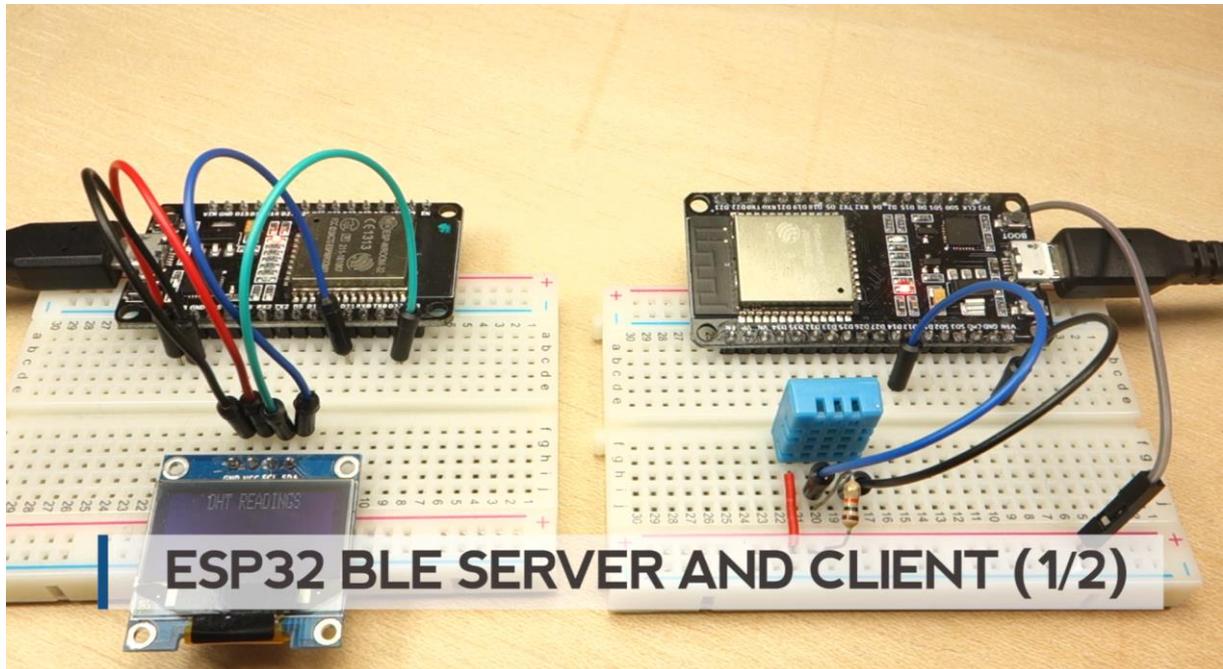
The “Scan” sketch also provides additional information, such as address and transmit power level (txPower).

Wrapping Up

That’s it for now, in the next Unit, you’re going to learn how to setup an ESP32 as a client and another ESP32 as a server, so they can exchange useful data via Bluetooth.

Unit 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. In this project the server is connected to a DHT temperature and humidity sensor that reports the latest readings every 10 seconds to the client. The ESP32 client has an OLED display that prints the current results.

Let's see what you'll achieve by the end of this project. Having the ESP32 running the server sketch, power the other ESP32 with the client sketch. The client starts scanning nearby devices and when it finds the other ESP32, it establishes a connection.

After a few seconds it should start receiving the latest temperature and humidity readings.

This project is divided in two parts:

- **Part 1** – Prepare the BLE server
- **Part 2** – Prepare the BLE client

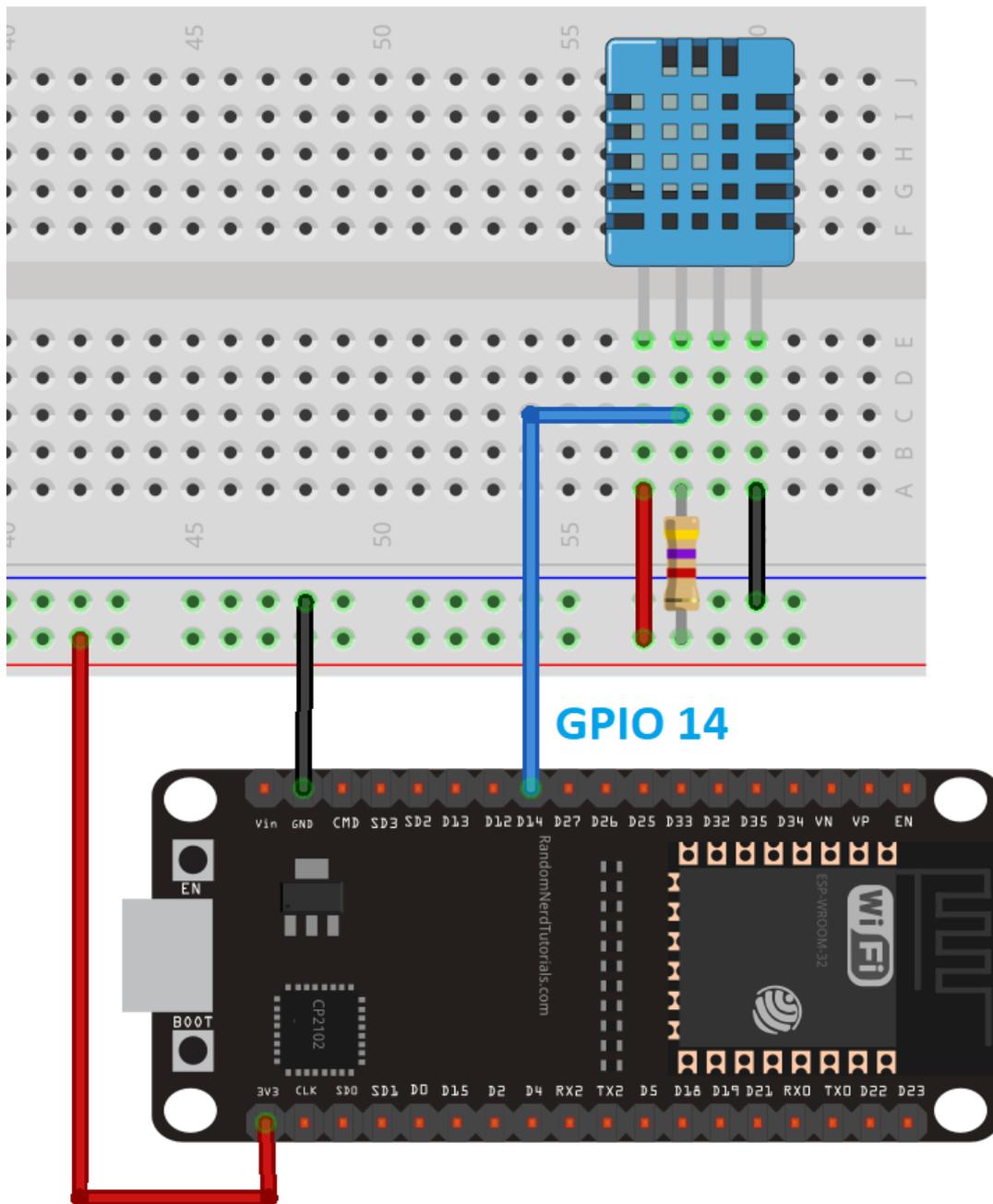
Schematic

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

- [ESP32 DOIT DEVKIT V1 Board](#)
- [DHT11/DHT22](#) temperature and humidity sensor
- [4.7k Ohm resistor](#)
- [Jumper wires](#)

- [Breadboard](#)
- Smartphone with Bluetooth (Example: [OnePlus 5](#))

Start this project by wiring the circuit. Wire the DHT sensor to the ESP32 using a 4.7k Ohm resistor as shown in the following 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 connect the DHT to any other ESP32 digital pin, you simply need to change the pin assignment in the sketch.

Installing the DHT Sensor Library

To read from the DHT sensor using Arduino IDE, you need to install the DHT sensor library. Follow the next steps to install the library.

- 1) [Click here to download the DHT Sensor library](#). You should have a .zip folder in your Downloads folder
- 2) Unzip the .zip folder and you should get DHT-sensor-library-master folder
- 3) Rename your folder from ~~DHT-sensor-library-master~~ to DHT_sensor
- 4) Move the DHT_sensor folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Installing the Adafruit Unified Sensor Driver

You also need to install the Adafruit Unified Sensor Driver Library. Follow the next steps to install the library.

- 1) Click here to download the [Adafruit Unified Sensor library](#). You should have a .zip folder in your Downloads folder
- 2) Unzip the .zip folder and you should get Adafruit_sensor-master folder
- 3) Rename your folder from ~~Adafruit_sensor-master~~ to Adafruit_Sensor
- 4) Move the DHT_sensor folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Server Sketch

With the circuit ready, and the needed libraries installed, open your Arduino IDE and copy the code provided below.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/BLE_Server_DHT/BLE_Server_DHT.ino

```
/*  
  Rui Santos  
  Complete project details at http://randomnerdtutorials.com  
*/  
  
#include <BLEDevice.h>  
#include <BLEServer.h>  
#include <BLEUtils.h>  
#include <BLE2902.h>  
#include "DHT.h"  
  
//Default Temperature is in Celsius  
//Comment the next line for Temperature in Fahrenheit  
#define temperatureCelsius  
  
//BLE server name  
#define bleServerName "dhtESP32"  
  
// 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
```

```

// See the following for generating UUIDs:
// https://www.uuidgenerator.net/
#define SERVICE_UUID "91bad492-b950-4226-aa2b-4ede9fa42f59"

#ifdef temperatureCelsius
    BLECharacteristic dhtTemperatureCelsiusCharacteristics("cba1d466-344c-4be3-ab3f-189f80dd7518", BLECharacteristic::PROPERTY_NOTIFY);
    BLEDescriptor dhtTemperatureCelsiusDescriptor(BLEUUID((uint16_t)0x2902));
#else
    BLECharacteristic dhtTemperatureFahrenheitCharacteristics("f78ebbf-c8b7-4107-93de-889a6a06d408", BLECharacteristic::PROPERTY_NOTIFY);
    BLEDescriptor
dhtTemperatureFahrenheitDescriptor(BLEUUID((uint16_t)0x2901));
#endif

BLECharacteristic dhtHumidityCharacteristics("ca73b3ba-39f6-4ab3-91ae-186dc9577d99", BLECharacteristic::PROPERTY_NOTIFY);
BLEDescriptor dhtHumidityDescriptor(BLEUUID((uint16_t)0x2903));

// DHT Sensor
const int DHTPin = 14;

// Initialize DHT sensor.
DHT dht(DHTPin, DHTTYPE);

bool deviceConnected = false;

//Setup callbacks onConnect and onDisconnect
class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
        deviceConnected = true;
    };
    void onDisconnect(BLEServer* pServer) {
        deviceConnected = false;
    }
};

void setup() {
    // Start DHT sensor
    dht.begin();

    // Start serial communication
    Serial.begin(115200);

    // Create the BLE Device
    BLEDevice::init(bleServerName);

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

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

    // Create BLE Characteristics and Create a BLE Descriptor
    // Descriptor
bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.descriptor.gatt.client_characteristic_configuration.xml

#ifdef temperatureCelsius
    dhtService->addCharacteristic(&dhtTemperatureCelsiusCharacteristics);
    dhtTemperatureCelsiusDescriptor.setValue("DHT temperature Celsius");
    dhtTemperatureCelsiusCharacteristics.addDescriptor(new BLE2902());
#else
    dhtService-
>addCharacteristic(&dhtTemperatureFahrenheitCharacteristics);

```

```

    dhtTemperatureFahrenheitDescriptor.setValue("DHT temperature
Fahrenheit");
    dhtTemperatureFahrenheitCharacteristics.addDescriptor(new BLE2902());
#endif

dhtService->addCharacteristic(&dhtHumidityCharacteristics);
dhtHumidityDescriptor.setValue("DHT humidity");
dhtHumidityCharacteristics.addDescriptor(new BLE2902());

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

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

void loop() {
    if (deviceConnected) {
        // Read temperature as Celsius (the default)
        float t = dht.readTemperature();
        // Read temperature as Fahrenheit (isFahrenheit = true)
        float f = dht.readTemperature(true);
        // Read humidity
        float h = dht.readHumidity();

        // 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!");
            return;
        }
        //Notify temperature reading from DHT sensor
#ifdef temperatureCelsius
        static char temperatureCTemp[7];
        dtostrf(t, 6, 2, temperatureCTemp);
        //Set temperature Characteristic value and notify connected client
        dhtTemperatureCelsiusCharacteristics.setValue(temperatureCTemp);
        dhtTemperatureCelsiusCharacteristics.notify();
        Serial.print("Temperature Celsius: ");
        Serial.print(t);
        Serial.print(" *C");
#else
        static char temperatureFTemp[7];
        dtostrf(f, 6, 2, temperatureFTemp);
        //Set temperature Characteristic value and notify connected client
        dhtTemperatureFahrenheitCharacteristics.setValue(temperatureFTemp);
        dhtTemperatureFahrenheitCharacteristics.notify();
        Serial.print("Temperature Fahrenheit: ");
        Serial.print(f);
        Serial.print(" *F");
#endif

        //Notify humidity reading from DHT
        static char humidityTemp[7];
        dtostrf(h, 6, 2, humidityTemp);
        //Set humidity Characteristic value and notify connected client
        dhtHumidityCharacteristics.setValue(humidityTemp);
        dhtHumidityCharacteristics.notify();
        Serial.print(" - Humidity: ");
        Serial.print(h);
        Serial.println(" %");

        delay(10000);
    }
}

```

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.

Note: You need to have the [ESP32 BLE library](#), the [DHT library](#), and [Adafruit Unified Sensor library](#) installed in your Arduino IDE.

Importing libraries

You start by importing the required libraries.

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include "DHT.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
```

So, it's up to you, you can either leave the Celsius degrees or comment the line to use Fahrenheit degrees. For this example, we'll be using Celsius degrees.

BLE server name

Leave the default BLE server name, otherwise the server name in the client code also needs to be changed, because they have to match.

```
#define bleServerName "dhtESP32"
```

Selecting the temperature sensor

In the following section, you can select the DHT sensor type that you're using. In this case, we'll be using the DHT11.

```
#define DHTTYPE DHT11 // DHT 11
//#define DHTTYPE DHT21 // DHT 21 (AM2301)
//#define DHTTYPE DHT22 // DHT 22 (AM2302), AM2321
```

Bluetooth UUIDs

I also recommend leaving all the default UUIDs. Otherwise, you also need to change the code in the client side, so the client can find the service and retrieve the characteristic values. The UUIDs are highlighted in the following lines in bold.

```
#define SERVICE_UUID "91bad492-b950-4226-aa2b-4ede9fa42f59"

#ifdef temperatureCelsius
    BLECharacteristic dhtTemperatureCelsiusCharacteristics ("cba1d466-344c-4be3-ab3f-189f80dd7518", BLECharacteristic::PROPERTY_NOTIFY);
    BLEDescriptor dhtTemperatureCelsiusDescriptor (BLEUUID((uint16_t)0x2902));
#else
```

```

BLECharacteristic dhtTemperatureFahrenheitCharacteristics("f78ebbf-f-c8b7-4107-93de-889a6a06d408", BLECharacteristic::PROPERTY_NOTIFY);
BLEDescriptor
dhtTemperatureFahrenheitDescriptor(BLEUUID((uint16_t)0x2901));
#endif

BLECharacteristic dhtHumidityCharacteristics("ca73b3ba-39f6-4ab3-91ae-186dc9577d99", BLECharacteristic::PROPERTY_NOTIFY);
BLEDescriptor dhtHumidityDescriptor(BLEUUID((uint16_t)0x2903));

```

Pin assignment

In the following line we define the DHT sensor pin. In this example we're connecting the digital pin of the DHT to GPIO 14. If you're using a different pin assignment, you should change it in the following line.

```
const int DHTPin = 14;
```

setup()

In the `setup()`, you start the DHT sensor:

```
// Start DHT sensor
dht.begin();
```

Start the serial port at baud rate of 115200:

```
// Start serial communication
Serial.begin(115200);
```

And 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 the 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.

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

Creating the BLE Service and Characteristics

Continuing to the `setup()`, start a BLE service with the service UUID defined earlier.

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

Then, create the temperature BLE characteristic. If you're using Celsius degrees it starts the following characteristic:

```
#ifdef temperatureCelsius
    dhtService->addCharacteristic(&dhtTemperatureCelsiusCharacteristics);
    dhtTemperatureCelsiusDescriptor.setValue("DHT temperature Celsius");
    dhtTemperatureCelsiusCharacteristics.addDescriptor(new BLE2902());
```

Otherwise it uses the Fahrenheit characteristic:

```
#else
    dhtService->addCharacteristic(&dhtTemperatureFahrenheitCharacteristics);
    dhtTemperatureFahrenheitDescriptor.setValue("DHT temperature
Fahrenheit");
    dhtTemperatureFahrenheitCharacteristics.addDescriptor(new BLE2902());
#endif
```

After that, it starts the humidity characteristic:

```
dhtService->addCharacteristic(&dhtHumidityCharacteristics);
dhtHumidityDescriptor.setValue("DHT humidity");
dhtHumidityCharacteristics.addDescriptor(new BLE2902());
```

Starting the Service and the Advertising

Finally, you start the service and the server starts the advertising, so other devices can find it.

```
// Start the service
dhtService->start();

// Start advertising
pServer->getAdvertising()->start();
```

loop()

The `loop()` function is fairly straightforward. You constantly check if the device is connected or not. If it's connected, it reads the current temperature and humidity.

```
if (deviceConnected) {
    // Read temperature as Celsius (the default)
    float t = dht.readTemperature();
    // Read temperature as Fahrenheit (isFahrenheit = true)
    float f = dht.readTemperature(true);
    // Read humidity
    float h = dht.readHumidity();
```

There's also a condition that checks if the DHT readings are valid or not.

```
if (isnan(h) || isnan(t) || isnan(f)) {  
    Serial.println("Failed to read from DHT sensor!");  
    return;  
}
```

If they are valid, the code continues.

If you're using temperature in Celsius it runs the following code section, where it stores the temperature in a temporary variable.

```
#ifndef temperatureCelsius  
    static char temperatureCTemp[7];  
    dtostrf(t, 6, 2, temperatureCTemp);
```

The following two lines update the current characteristic value (using `.setValue()`) and send it to the connected client (using `.notify()`).

```
dhtTemperatureCelsiusCharacteristics.setValue(temperatureCTemp);  
dhtTemperatureCelsiusCharacteristics.notify();
```

There are also three lines to print the temperature in the Serial Monitor for debugging purposes.

```
Serial.print("Temperature Celsius: ");  
Serial.print(t);  
Serial.print(" *C");
```

The same process is used to send the temperature in Fahrenheit.

```
#else  
    static char temperatureFTemp[7];  
    dtostrf(f, 6, 2, temperatureFTemp);  
    //Set temperature Characteristic value and notify connected client  
    dhtTemperatureFahrenheitCharacteristics.setValue(temperatureFTemp);  
    dhtTemperatureFahrenheitCharacteristics.notify();  
    Serial.print("Temperature Fahrenheit: ");  
    Serial.print(f);  
    Serial.print(" *F");  
#endif
```

Sending the humidity also uses the same process.

```
//Notify humidity reading from DHT  
static char humidityTemp[7];  
tostrf(h, 6, 2, humidityTemp);  
//Set humidity Characteristic value and notify connected client  
dhtHumidityCharacteristics.setValue(humidityTemp);  
dhtHumidityCharacteristics.notify();  
Serial.print(" - Humidity: ");  
Serial.print(h);  
Serial.println(" %");
```

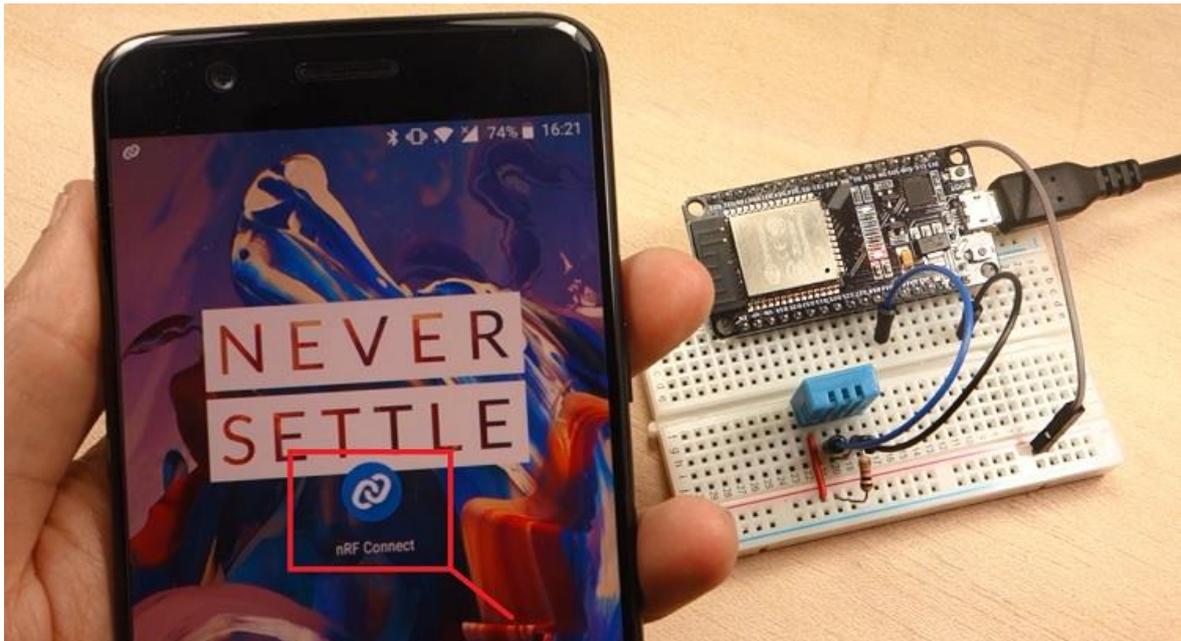
The delay function waits 10 seconds between readings.

```
delay(10000);
```

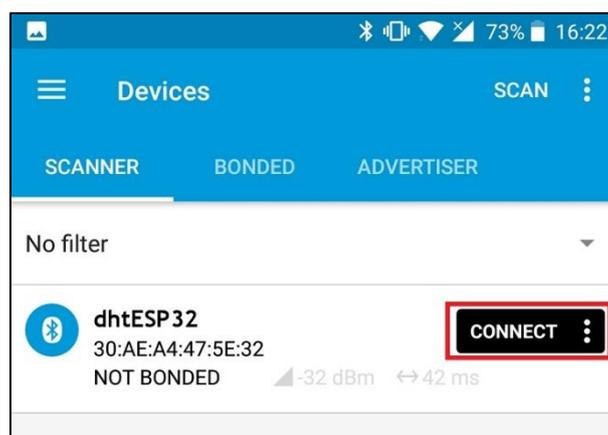
You could have used a timer for a more efficient way to send readings, but we've used this method to keep the project simpler.

Testing the ESP32 BLE Server

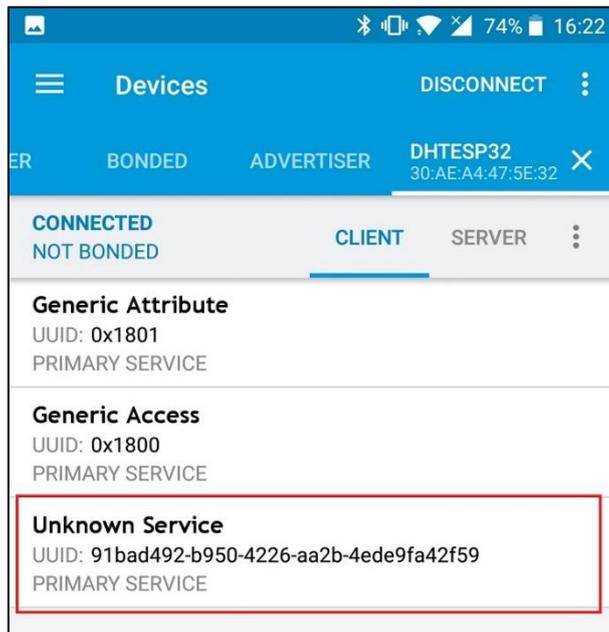
Upload the code to your board. Then, go to your smartphone and open the nRF connect app from Nordic.



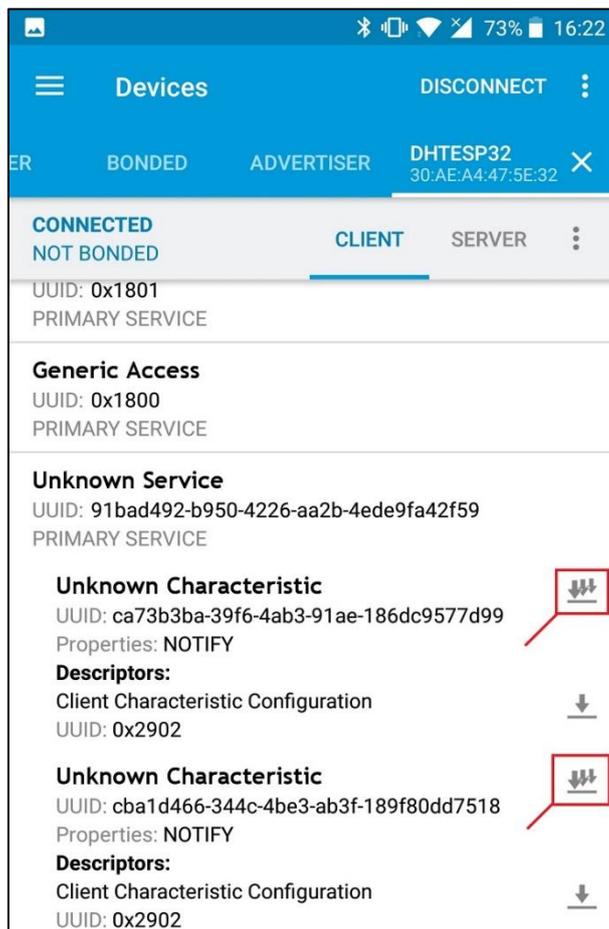
Having the ESP board powered on, turn on the Bluetooth and start scanning. You should find a device called dhtESP32 – this is the BLE server name you've defined earlier.



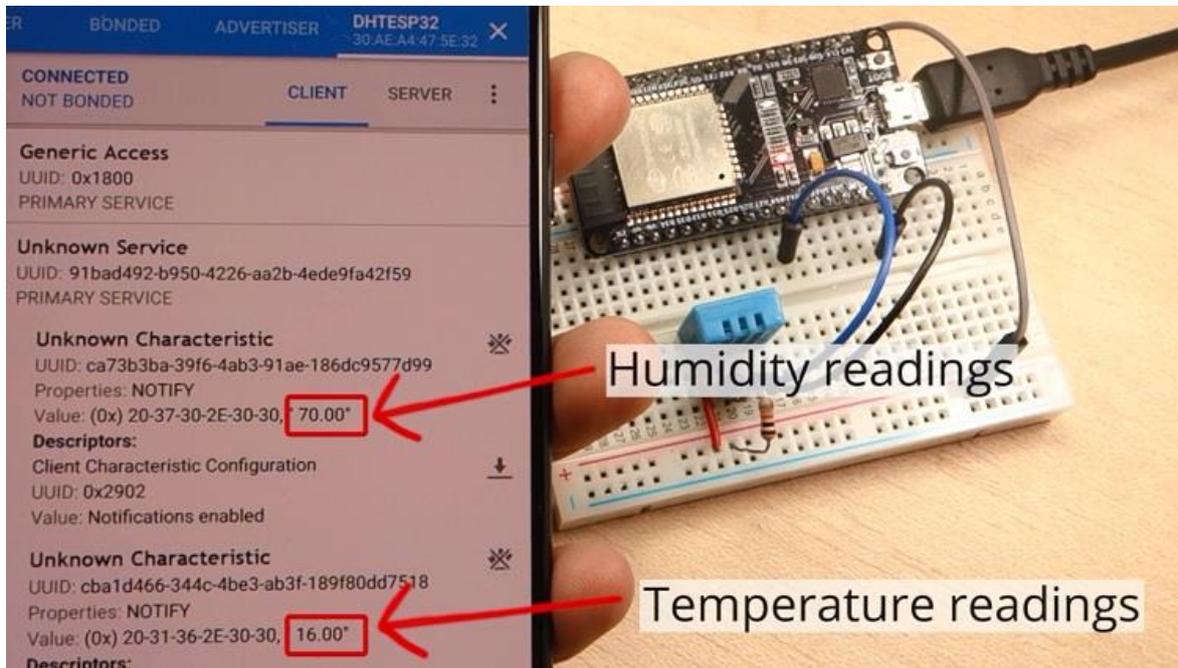
Connect to it, and open the custom service.



Activate the notify properties for the temperature and humidity.



You should start receiving new readings every 10 seconds.



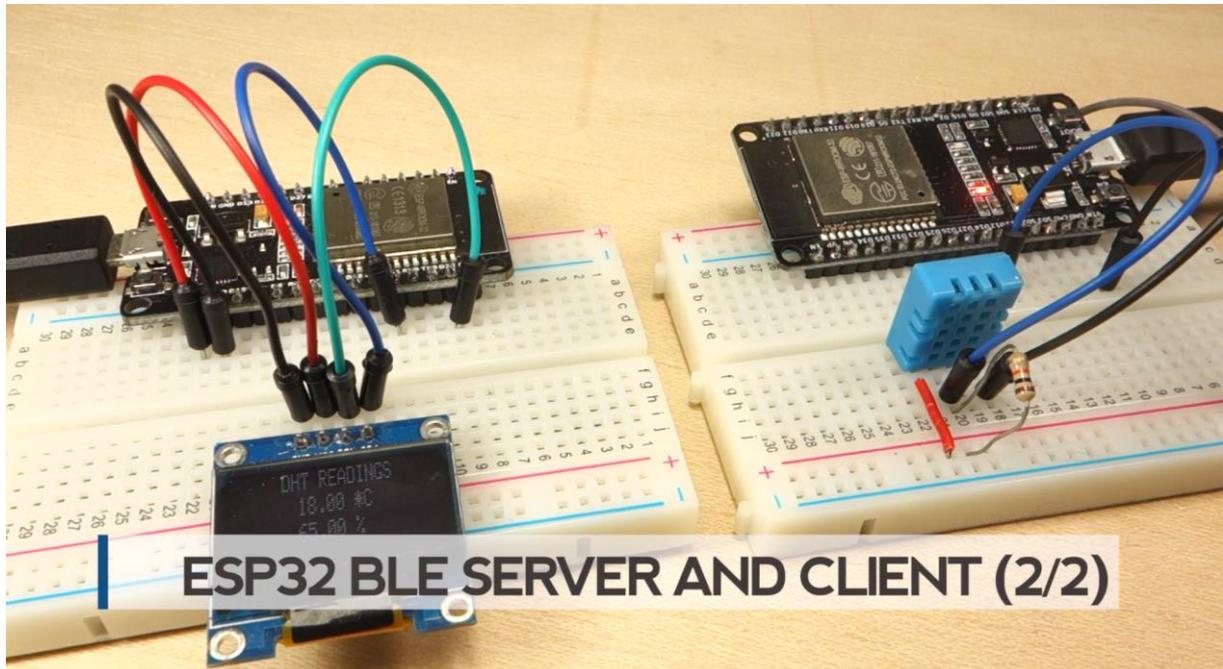
Your ESP32 BLE Server is ready!

Wrapping Up

That's it for now. Go to the next Unit to complete this project. In the next Unit, you're going to create an ESP32 client to receive the readings and display them on an OLED display.

Unit 4 - ESP32 BLE Server and Client

(Part 2/2)



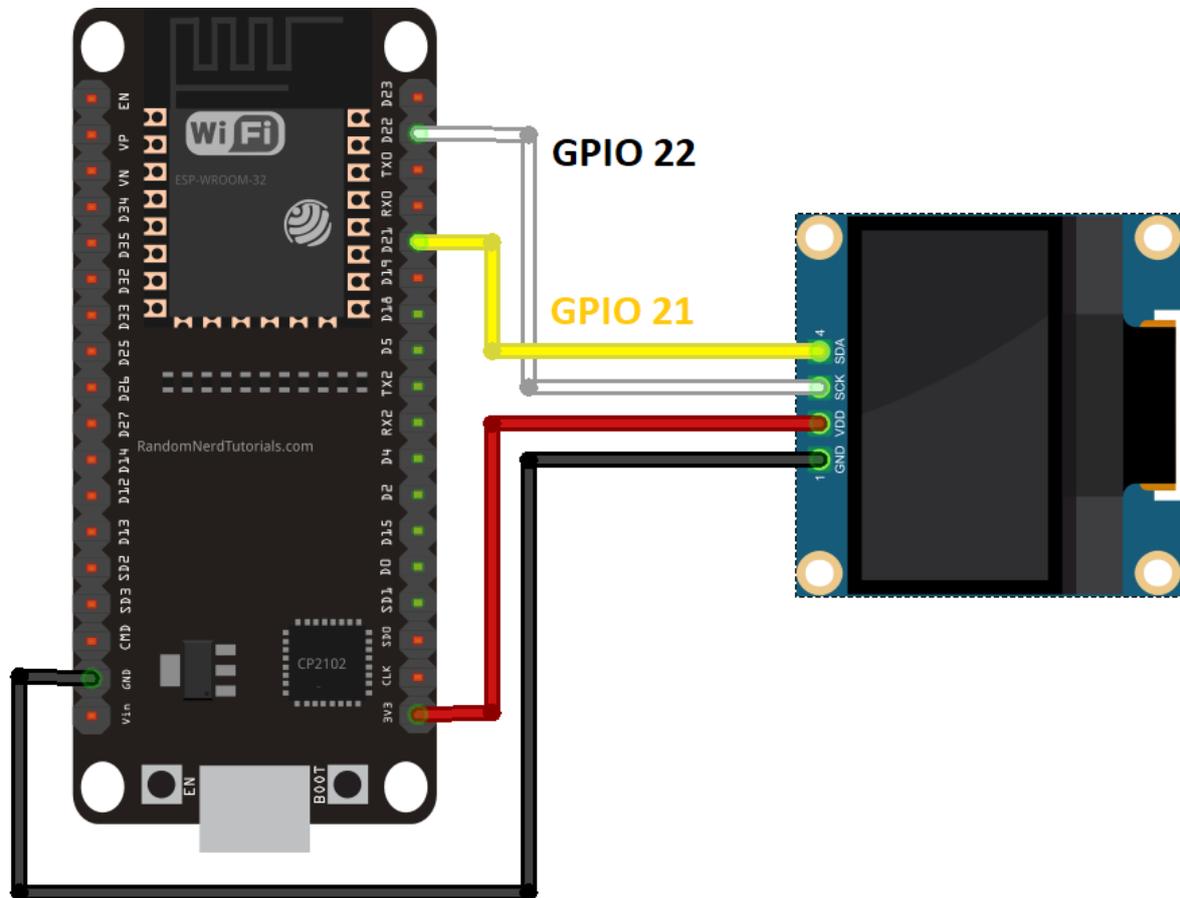
Having the ESP32 BLE server ready from the previous Unit, let's create the ESP32 BLE client that will establish a connection and display the readings on an OLED display. If you don't have the ESP32 BLE server ready yet, go back to the previous Unit.

Schematic

The ESP32 BLE client will be attached to an OLED display, so that we can see the readings sent 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](#)



(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.)

Installing the Adafruit_SSD1306 Library

The [Adafruit_SSD1306 library](#) provides an easy way to write text on the OLED display using the Arduino IDE. Follow the next steps to install the library in your Arduino IDE:

- 1) Click here to download the [Adafruit_SSD1306 library](#). You should have a .zip folder in your Downloads folder
- 2) Unzip the .zip folder and you should get Adafruit_SSD1306-master folder
- 3) Rename your folder from ~~Adafruit_SSD1306-master~~ to Adafruit_SSD1306
- 4) Move the Adafruit_SSD1306 folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Installing the Adafruit GFX library

You also need to install the [Adafruit GFX library](#). Follow the next steps to install that library:

- 1) Click here to download the [Adafruit GFX library](#). You should have a .zip folder in your Downloads folder
- 2) Unzip the .zip folder and you should get Adafruit-GFX-library-master folder
- 3) Rename the folder from Adafruit-GFX-library-master to Adafruit_GFX_library

- 4) Move the Adafruit_GFX_library folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Client Sketch

Copy the BLE client Sketch to your Arduino IDE.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/BLE_Client_OLED/BLE_Client_OLED.ino

```
#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 "dhtESP32"

//UUID's of the service, characteristic that we want to read and
characteristic that we want to write.
static BLEUUID dhtServiceUUID("91bad492-b950-4226-aa2b-4ede9fa42f59");

#ifdef 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

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... Hopefully.
static BLEAddress *pServerAddress;

//Characteristic that we want to read and characteristic that we want
to write.
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)
#define OLED_RESET      4 // Reset pin # (or -1 if sharing Arduino
reset pin)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire);

//Variables to store temperature and humidity
char* temperatureR;
char* humidityR;

//Flags to check whether new temperature and humidity readings are
available
boolean newTemperatureR = false;
boolean newHumidityR = false;

//Connect to the BLE Server that has the name, Service, and
Characteristics
bool connectToServer(BLEAddress pAddress) {
    BLEClient* pClient = BLEDevice::createClient();

    // Connect to the remove BLE Server.
    pClient->connect(pAddress);
    Serial.println(" - Connected to server");

    // Obtain a reference to the service we are after in the remote BLE
server.
    BLERemoteService* pRemoteService = pClient-
>getService(dhtServiceUUID);
    if (pRemoteService == nullptr) {
        Serial.print("Failed to find our service UUID: ");
        Serial.println(dhtServiceUUID.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) {
        if (advertisedDevice.getName() == bleServerName) { //Check if the
name of the advertiser matches
            advertisedDevice.getScan()->stop(); //Scan can be stopped, we
found what we are looking for
            pServerAddress = new BLEAddress(advertisedDevice.getAddress());
//Address of advertiser is the one we need
            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) {
    //store temperature value
    temperatureR = (char*)pData;
    newTemperatureR = 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) {
    //store humidity value
    humidityR = (char*)pData;
    newHumidityR = true;
}

//function that prints the latest sensor readings in the OLED display
void printDHTReadings(){
    display.clearDisplay();
    display.setTextSize(2);
    display.setTextColor(WHITE);
    display.setCursor(0,0);
    display.print("DHT SENSOR");

    //display temperature
    display.setCursor(0,20);
    display.print(temperatureR);
    Serial.print("Temperature:");
    Serial.print(temperatureR);
    #ifdef temperatureCelsius
        //Temperature Celsius
        display.print("*C");
        Serial.print("*C");
    #else
        //Temperature Fahrenheit

```

```

        display.print("*F");
        Serial.print("*F");
    #endif

    //display humidity
    display.setCursor(0,40);
    display.print(humidityR);
    display.print("%");
    display.display();
    Serial.print(" Humidity:");
    Serial.print(humidityR);
    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(0,0);
    display.print("DHT SENSOR");
    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 in the OLED
if (newTemperatureR && newHumidityR) {
    newTemperatureR = false;
    newHumidityR = false;
    printDHTReadings();
}
delay(1000); // Delay a second between loops.
}

```

Importing libraries

You start by importing the required libraries:

```

#include "BLEDevice.h"
#include <Wire.h>
#include <Adafruit_SSD1306.h>
#include <Adafruit_GFX.h>

```

Choosing temperature unit

By default the client will receive the temperature in Celsius degrees, if you comment the following line or delete it, it will start receiving the temperature in Fahrenheit degrees.

```

//Default Temperature is in Celsius
//Comment the next line for Temperature in Fahrenheit
#define temperatureCelsius

```

Important: Note that both the ESP32 server and client should use the same temperature unit, otherwise it will not work.

BLE Server Name and UUIDs

As said in the previous Unit, leave the default BLE server name and UUIDs to match the ones defined in the server sketch.

So the BLE server name should be as follows:

```

#define bleServerName "dhtESP32"

```

And the UUIDs:

```

static BLEUUID dhtServiceUUID("91bad492-b950-4226-aa2b-4ede9fa42f59");

#ifdef temperatureCelsius
    //Temperature Celsius Characteristic
    static BLEUUID temperatureCharacteristicUUID("cba1d466-344c-4be3-ab3f-189f80dd7518");

```

```

#else
    //Temperature Fahrenheit Characteristic
    static BLEUUID temperatureCharacteristicUUID("f78ebbf-fc8b7-4107-93de-889a6a06d408");
#endif

static BLEUUID humidityCharacteristicUUID("ca73b3ba-39f6-4ab3-91ae-186dc9577d99");

```

Declaring variables

Then, you need to declare some variables that will be used later with Bluetooth.

```

//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... Hopefully.
static BLEAddress *pServerAddress;

//Characteristic that we want to read and characteristic that we want to write.
static BLERemoteCharacteristic* temperatureCharacteristic;
static BLERemoteCharacteristic* humidityCharacteristic;

//Activate notify
const uint8_t notificationOn[] = {0x1, 0x0};
const uint8_t notificationOff[] = {0x0, 0x0};

```

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

```

Define the OLED reset pin (even if your display doesn't have one, this is necessary so that the library works properly).

```

#define OLED_RESET      4

```

Instantiate the OLED display with the width and height defined earlier.

```

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire);

```

Define some char variables to hold the temperature and humidity values received by the server.

```

char* temperatureR;
char* humidityR;

```

The following variables are used to check whether new temperature and humidity readings are available.

```
boolean newTemperatureR = false;
boolean newHumidityR = false;
```

printDHTReadings()

Create a function called `printDHTReadings()` that displays the temperature and humidity readings on the OLED display.

```
void printDHTReadings() {
  display.clearDisplay();
  display.setTextSize(2);
  display.setTextColor(WHITE);
  display.setCursor(0,0);
  display.print("DHT SENSOR");

  //display temperature
  display.setCursor(0,20);
  display.print(temperatureR);
  Serial.print("Temperature:");
  Serial.print(temperatureR);
  #ifdef temperatureCelsius
    //Temperature Celsius
    display.print("*C");
    Serial.print("*C");
  #else
    //Temperature Fahrenheit
    display.print("*F");
    Serial.print("*F");
  #endif

  //display humidity
  display.setCursor(0,40);
  display.print(humidityR);
  display.print("%");
  display.display();
  Serial.print(" Humidity:");
  Serial.print(humidityR);
  Serial.println("%");
}
```

setup()

In the `setup()`, start the OLED display with the right settings.

```
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for
128x32
  Serial.println(F("SSD1306 allocation failed"));
  for(;;); // Don't proceed, loop forever
}
```

Then, print a message in the first line saying "DHT SENSOR".

```
display.clearDisplay();
display.setTextSize(2);
display.setTextColor(WHITE,0);
display.setCursor(0,0);
display.print("DHT SENSOR");
```

```
display.display();
```

Start the serial communication at a baud rate of 115200.

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

And initialize the BLE device.

```
BLEDevice::init("");
```

Scan nearby devices

The following methods scan for nearby devices.

```
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, it 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.

```
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {  
    void onResult(BLEAdvertisedDevice advertisedDevice) {  
        if (advertisedDevice.getName() == bleServerName) { //Check if the name  
of the advertiser matches  
            advertisedDevice.getScan()->stop(); //Scan can be stopped, we found  
what we are looking for  
            pServerAddress = new BLEAddress(advertisedDevice.getAddress());  
//Address of advertiser is the one we need  
            doConnect = true; //Set indicator, stating that we are ready to  
connect  
            Serial.println("Device found. Connecting!");  
        }  
    }  
};
```

Connect to the server

If the `doConnect` variable is true, it tries to connect to the BLE server. The `connectToServer()` function handles all the connection between the client and the server.

```
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(dhtServiceUUID);
if (pRemoteService == nullptr) {
    Serial.print("Failed to find our service UUID: ");
    Serial.println(dhtServiceUUID.toString().c_str());
    return (false);
}

// Obtain a reference to characteristics in service of 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);
}

```

It also assigns a callback function responsible to handle 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 active the notify property for each characteristic. For that, use the `writeValue()` method.

```

temperatureCharacteristic->getDescriptor(BLEUUID((uint16_t)0x2902))-
>writeValue((uint8_t*)notificationOn, 2, true);
humidityCharacteristic->getDescriptor(BLEUUID((uint16_t)0x2902))-
>writeValue((uint8_t*)notificationOn, 2, true);

```

Notify new values

When the client receives a new notify value, it will call these two functions: `temperatureNotifyCallback()` and `humidityNotifyCallback()` that are responsible for retrieving the new value, update the OLED with the new readings and print them on the Serial Monitor.

```

static void temperatureNotifyCallback(BLERemoteCharacteristic*
pBLERemoteCharacteristic, uint8_t* pData, size_t length, bool
isNotify) {
    //store temperature value

```

```

    temperatureR = (char*)pData;
    newTemperatureR = true;
}

static void humidityNotifyCallback(BLERemoteCharacteristic*
pBLERemoteCharacteristic, uint8_t* pData, size_t length, bool
isNotify) {
    //store humidity value
    humidityR = (char*)pData;
    newHumidityR = true;
}

```

These two previous functions are executed every time the BLE server notifies the client with a new value, which happens every 10 seconds. These functions save the values received on the `temperatureR` and `humidityR` variables. These also change the `newTemperatureR` and `newHumidityR` 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 there new readings are available. If there are new readings, we set the `newTemperatureR` and `newHumidityR` variables to `false`, so that we are able to receive new readings later on. Then, we call the `printDHTReadings()` function to display the readings on the OLED.

```

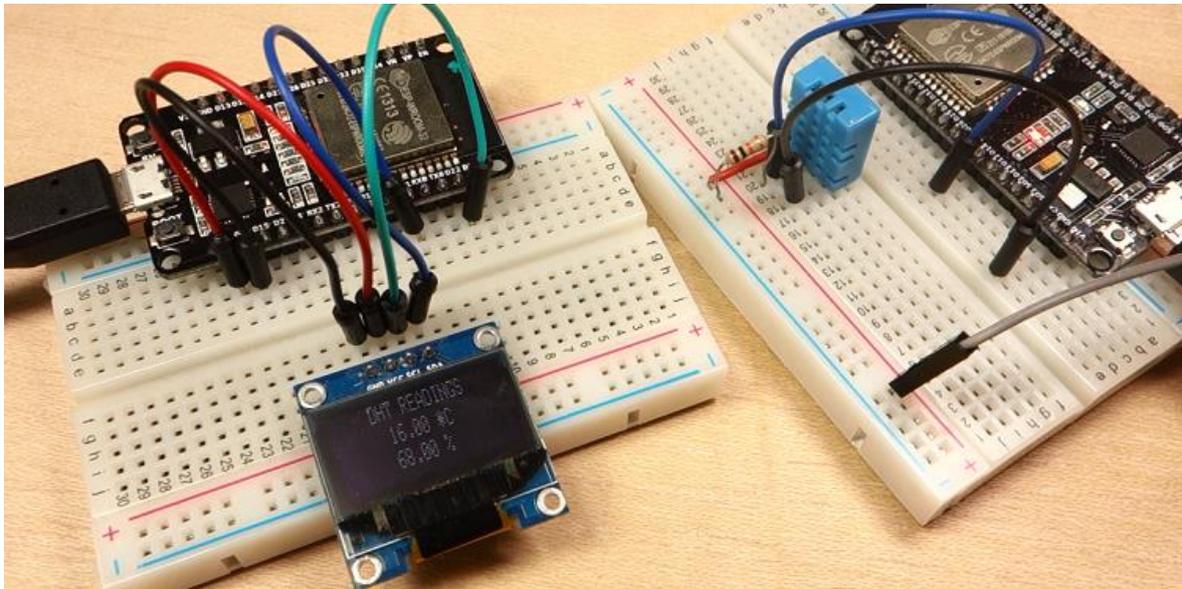
if (newTemperatureR && newHumidityR) {
    newTemperatureR = false;
    newHumidityR = false;
    printDHTReadings();
}

```

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 prepared in the previous video with the server sketch, 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 10 seconds it should display the latest readings.



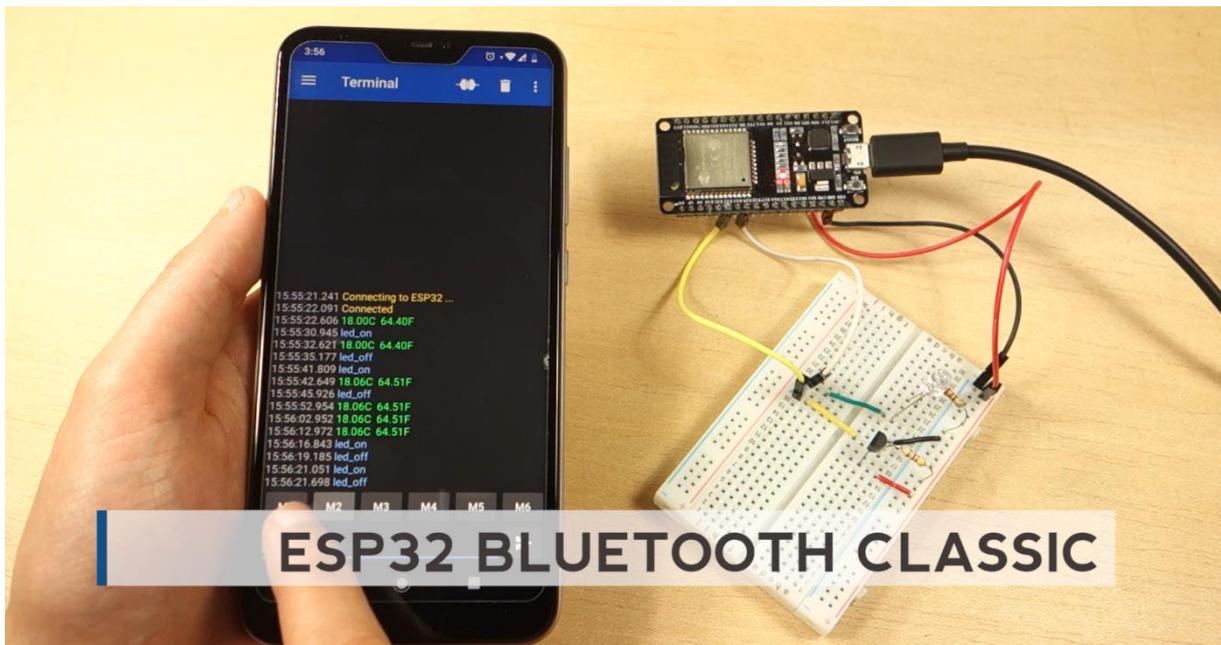
Troubleshooting tip: if you're not receiving any data on the OLED display there are two common problems:

- The DHT is failing to read the temperature and humidity (open the serial monitor in the BLE server to see if it's actually printing any results)
- Your ESP32 might be failing to establish a connection with the server. Reboot the ESP32 running BLE client sketch.

Wrapping Up

The project is completed! Now you now how to create two BLE devices that can exchange useful data. Feel free to modify the examples to use other sensors.

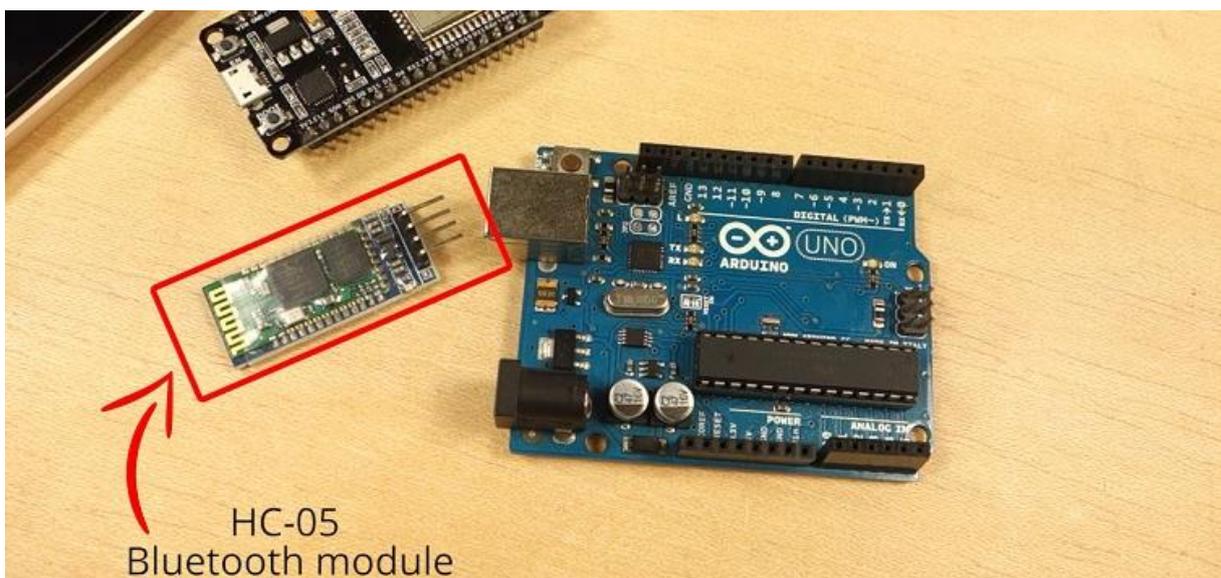
Unit 5 - Bluetooth Classic



The ESP32 comes with Wi-Fi, Bluetooth Low Energy and Bluetooth Classic. In this Unit, we'll 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 send sensor readings with an Android smartphone using Bluetooth Classic.

Note: this project is only compatible with Android smartphones.

At the moment, using Bluetooth Classic is much more simpler than Bluetooth Low Energy. If you've already programmed an Arduino with a Bluetooth module like the HC-05, this is very similar. It uses the standard serial protocol and functions.

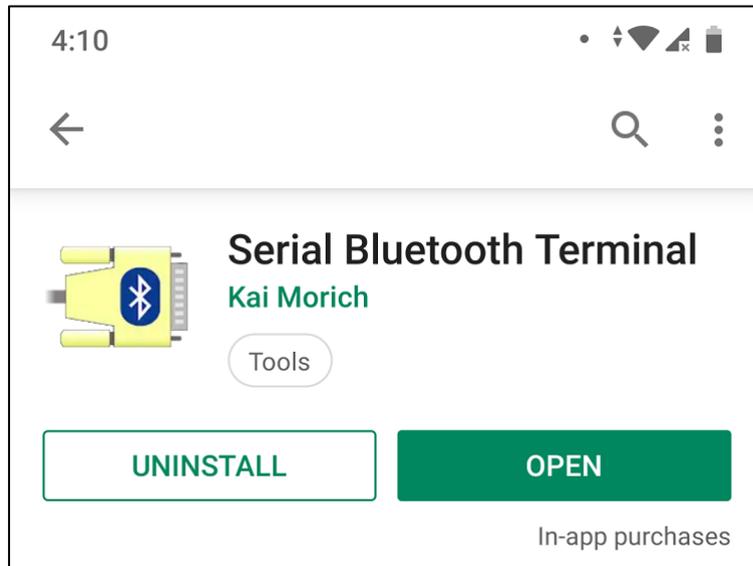


We'll start by using an example that comes with the Arduino IDE. Then, we'll build a simple project to exchange data between the ESP32 and your Android smartphone.

Bluetooth Terminal Application

To proceed with this tutorial, you need a Bluetooth Terminal application installed in your smartphone.

We recommend using the Android app "[Serial Bluetooth Terminal](#)" available in the Play Store.



Serial to Serial Bluetooth

After installing the application, open your Arduino IDE, and go to **File** ▶ **Examples** ▶ **BluetoothSerial** ▶ **SerialtoSerialBT**.

```
//By Evandro Copercini - 2018
//This example creates a bridge between Serial and Classical Bluetooth (SPP)
//and also demonstrate that SerialBT have the same functionalities of a normal
Serial

#include "BluetoothSerial.h"

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

BluetoothSerial SerialBT;

void setup() {
  Serial.begin(115200);
  SerialBT.begin("ESP32test"); //Bluetooth device name
  Serial.println("The device started, now you can pair it with bluetooth!");
}
void loop() {
  if (Serial.available()) {
    SerialBT.write(Serial.read());
  }
  if (SerialBT.available()) {
    Serial.write(SerialBT.read());
  }
  delay(20);
}
```

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/raw/master/code/Bluetooth_Classic_Example/Bluetooth_Classic_Example.ino

How the Code Works

This code establishes a two way serial Bluetooth communication between two devices.

The code starts by including the `BluetoothSerial` library.

```
#include "BluetoothSerial.h"
```

The next three lines check if Bluetooth is properly enabled.

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

Then, create an instance of `BluetoothSerial` called `SerialBT`:

```
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. By default it's called **ESP32test** but you can rename it and give a unique name.

```
SerialBT.begin("ESP32test"); //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. `Serial.read()` returns the data received in the serial port.

The next if statement, checks if there are bytes available to read in the Bluetooth Serial port. If there are, we'll write those bytes in the Serial Monitor.

```
if (SerialBT.available()) {
    Serial.write(SerialBT.read());
}
```

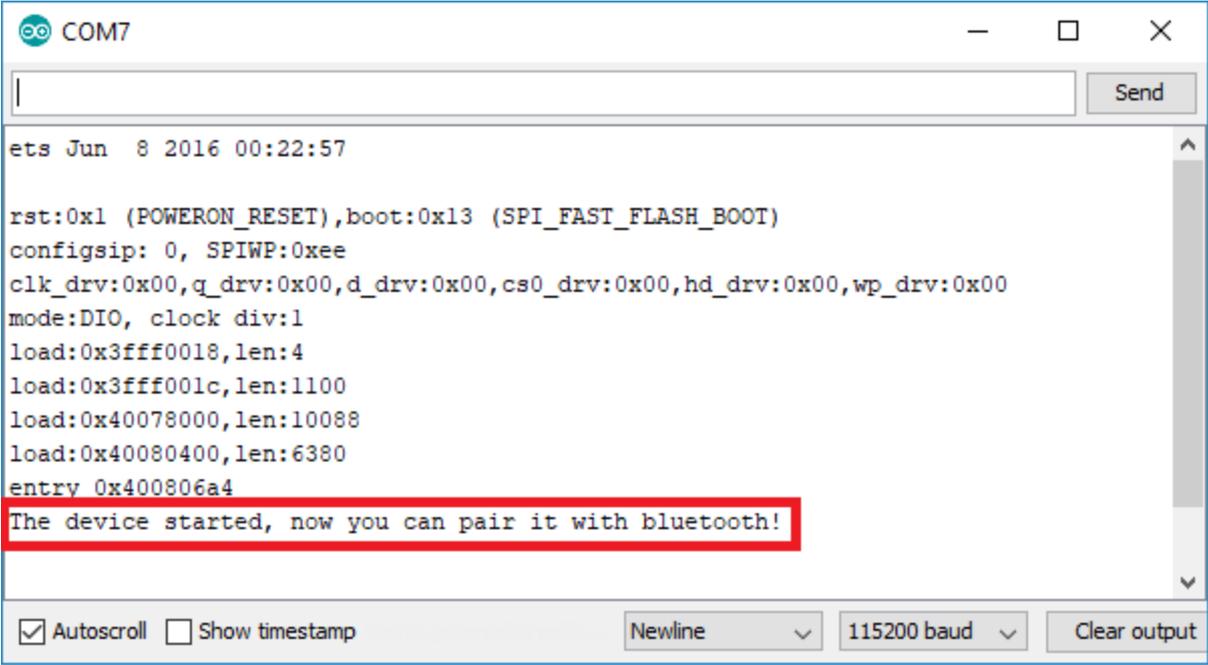
It will be easier to understand exactly how this sketch works during the demonstration.

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.

After a few seconds, you should get a message saying: *"The device started, now you can pair it with bluetooth!"*.

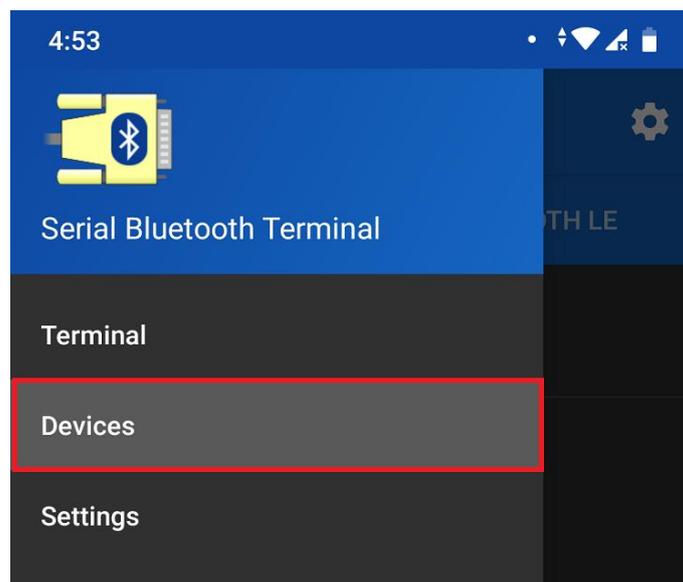


```
ets Jun 8 2016 00:22:57

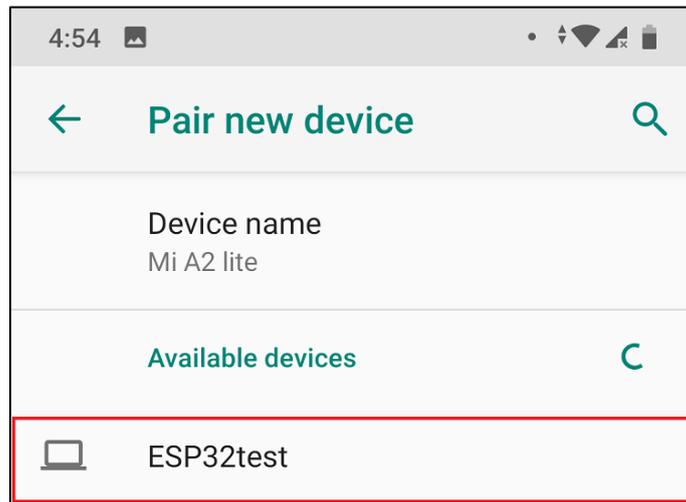
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
config: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1100
load:0x40078000,len:10088
load:0x40080400,len:6380
entry 0x400806a4
The device started, now you can pair it with bluetooth!
```

Go to your smartphone and open the Serial **Bluetooth Terminal** app. Make sure you've enable your smartphone's Bluetooth.

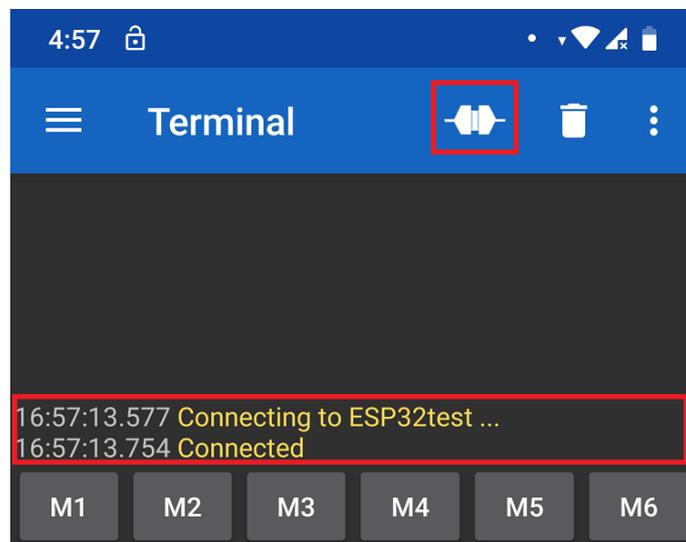
To connect to the ESP32 for the first time, you need to pair a new device. Go to **Devices**.



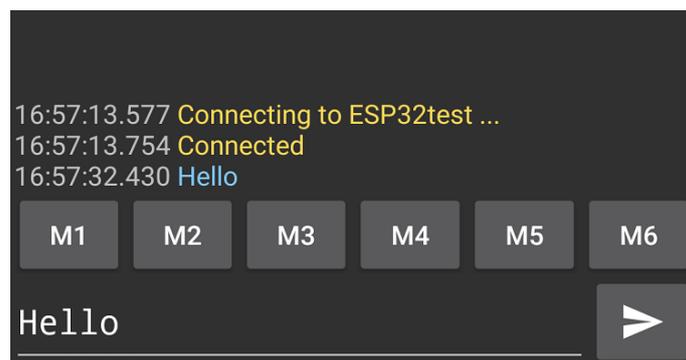
Click the **settings** icon, and select **Pair new device**. You should get a list with the available bluetooth devices, including the **ESP32test**. Pair with the **ESP32test**.



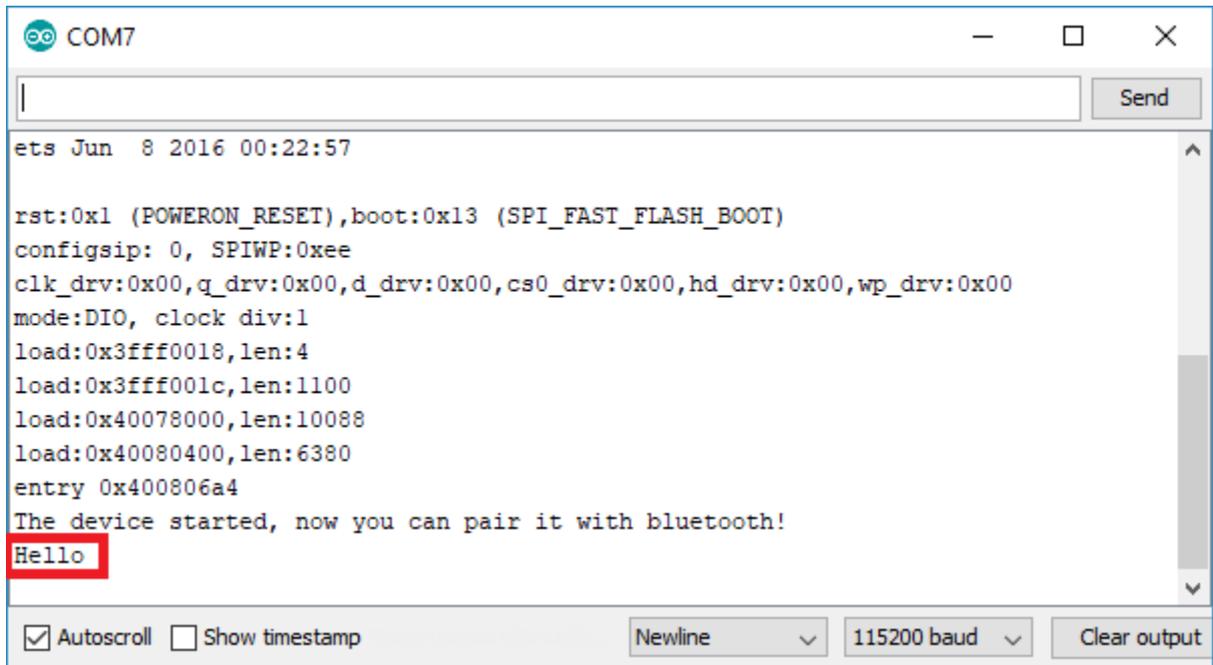
Then, go back to the Serial Bluetooth Terminal. Click the icon at the top to connect to the ESP32. You should get a **“Connected”** message.



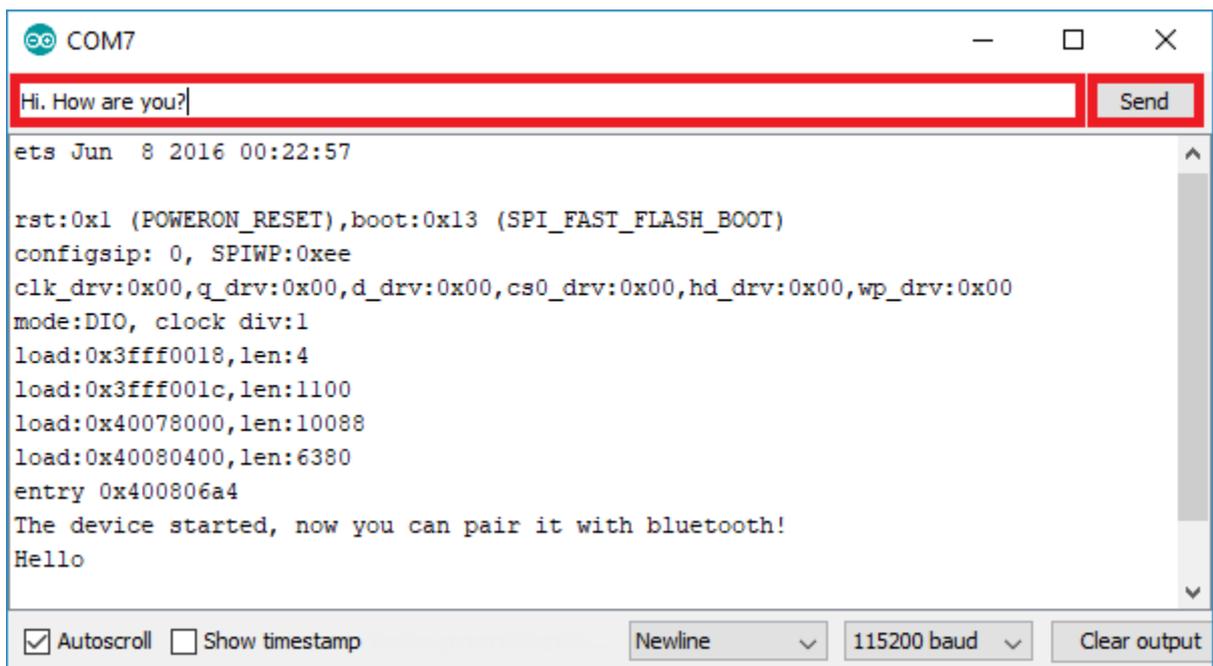
After that, type something in the Serial Bluetooth Terminal app. For example, **“Hello”**.



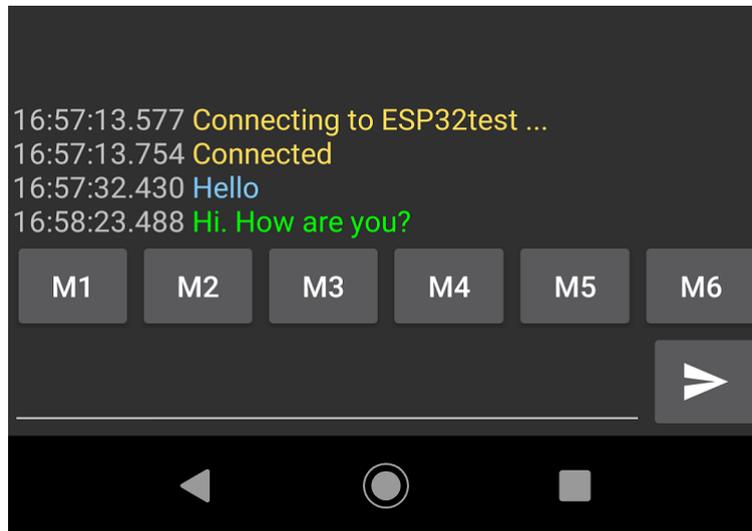
You should instantly receive that message in the Arduino IDE Serial Monitor.



You can also exchange data between your Serial Monitor and your smartphone. Type something in the Serial Monitor top bar and press the “Send” button.



You should instantly receive that message in the Serial Bluetooth Terminal App.

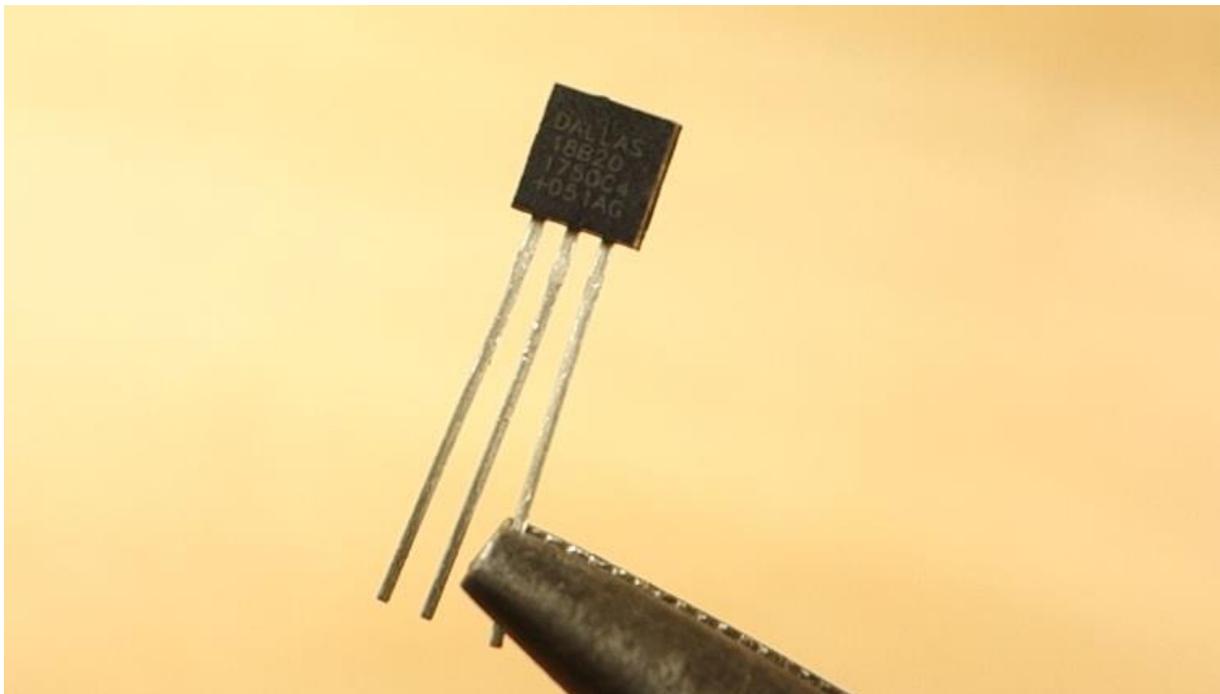


Exchange Data using Bluetooth Serial

Now that you know how to exchange data using Bluetooth Serial, you can modify the previous sketch to make something useful. For example, control the ESP32 outputs when you receive a certain message, or send data to your smartphone like sensor readings.

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. When the ESP32 receives the **led_on** message, we'll turn the GPIO on, when it receives the **led_off** message, we'll turn the GPIO off.

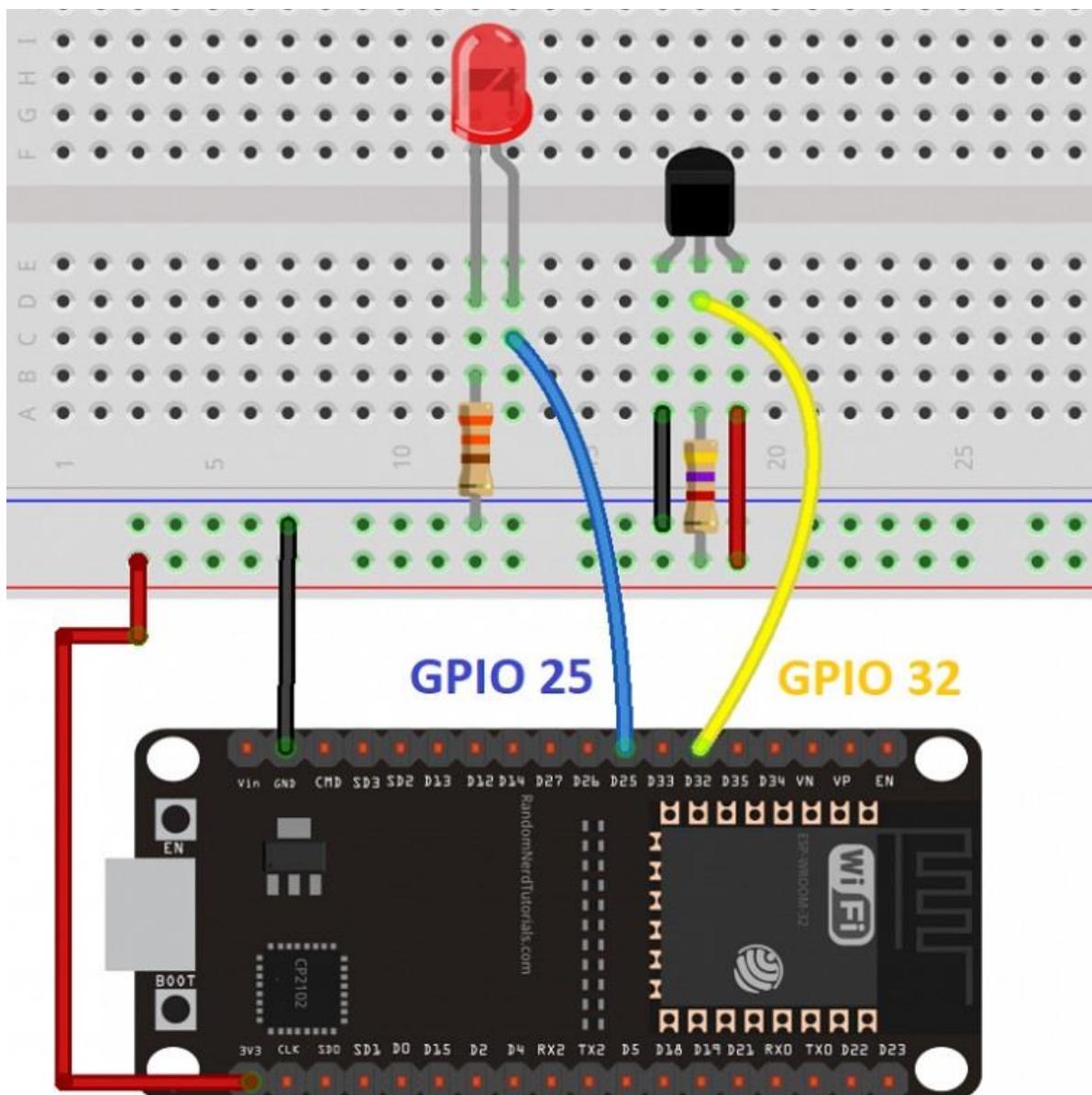
Schematic

Before proceeding with this project, assemble the circuit by following the next schematic diagram.

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](#)

Connect an LED to GPIO 25, and connect the DS18B20 data pin to GPIO 32.



(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.)

Code

To work with the DS18B20 temperature sensor, you need to install the [One Wire library by Paul Stoffregen](#) and the [Dallas Temperature library](#). Follow the next instructions to install these libraries, if you haven't already.

One Wire library

1. [Click here to download the One Wire library](#). You should have a .zip folder in your **Downloads**
2. Unzip the .zip folder and you should get **OneWire-master** folder
3. Rename your folder from **OneWire-master** to **OneWire**
4. Move the **OneWire** folder to your Arduino IDE installation **libraries** folder

Dallas Temperature library

1. [Click here to download the Dallas Temperature library](#). You should have a .zip folder in your **Downloads**
2. Unzip the .zip folder and you should get **Arduino-Temperature-Control-Library-master** folder
3. Rename your folder from **Arduino-Temperature-Control-Library-master** to **DallasTemperature**
4. Move the **DallasTemperature** folder to your Arduino IDE installation **libraries** folder
5. Finally, re-open your Arduino IDE

After assembling the circuit and installing the necessary libraries, copy the following sketch to your Arduino IDE.

SOURCE CODE

[https://github.com/RuiSantosdotme/ESP32-Course/raw/master/code/Bluetooth Classic Project/Bluetooth Classic Project.ino](https://github.com/RuiSantosdotme/ESP32-Course/raw/master/code/Bluetooth%20Classic%20Project/Bluetooth%20Classic%20Project.ino)

```
// 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
// 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 the Code Works

Let's take a quick look at the code and see how it works.

Start by including the necessary libraries. The `BluetoothSerial` library for Bluetooth, and the `OneWire` and `DallasTemperature` for the DS18B20 temperature sensor.

```
#include "BluetoothSerial.h"
#include <OneWire.h>
#include <DallasTemperature.h>
```

Create a `BluetoothSerial` instance called `SerialBT`.

```
BluetoothSerial SerialBT;
```

Create a variable called `ledPin` to hold the GPIO you want to control. In this case, GPIO 25 has an LED connected.

```
const int ledPin = 25;
```

Define the DS18B20 sensor pin and create objects to make it work. The temperature sensor is connected to GPIO 32.

```
// 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: auxiliary 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";
```

Then, to send the `temperatureString` via bluetooth, use `SerialBT.println()`.

```
SerialBT.println(temperatureString);
```

The next if statement reads incoming messages. When you receive messages via serial, you receive a character at a time. You know that the message ended, when you receive `\n`.

So, we check if there's data available in the Bluetooth serial port.

```
if (SerialBT.available()){
    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

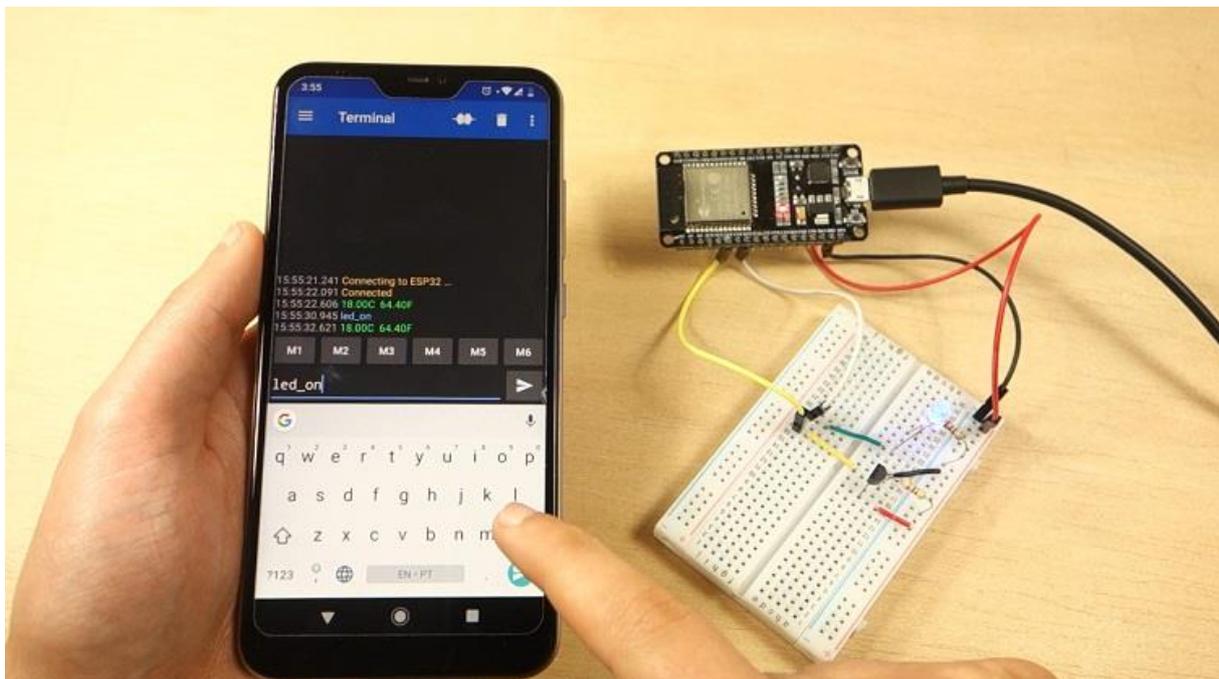
Upload the previous code to your ESP32 board.

Open the Serial Monitor, and press the ESP32 Enable button. When you receive the *"The device started, now you can pair it with bluetooth!"* message, you can go to your smartphone and connect with the ESP32.

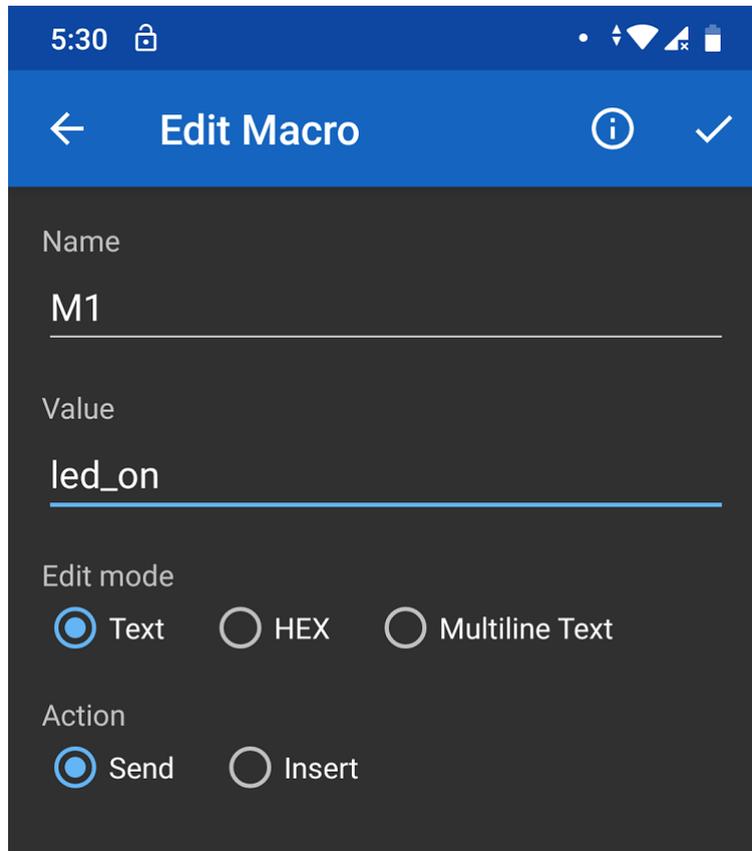
```
mode:PIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1100
load:0x40078000,len:10088
load:0x40080400,len:6380
entry 0x400806a4
The device started, now you can pair it with bluetooth!
```

Autoscroll Show timestamp Newline 115200 baud Clear output

Then, you can write the **"led_on"** and **"led_off"** messages to control the LED.

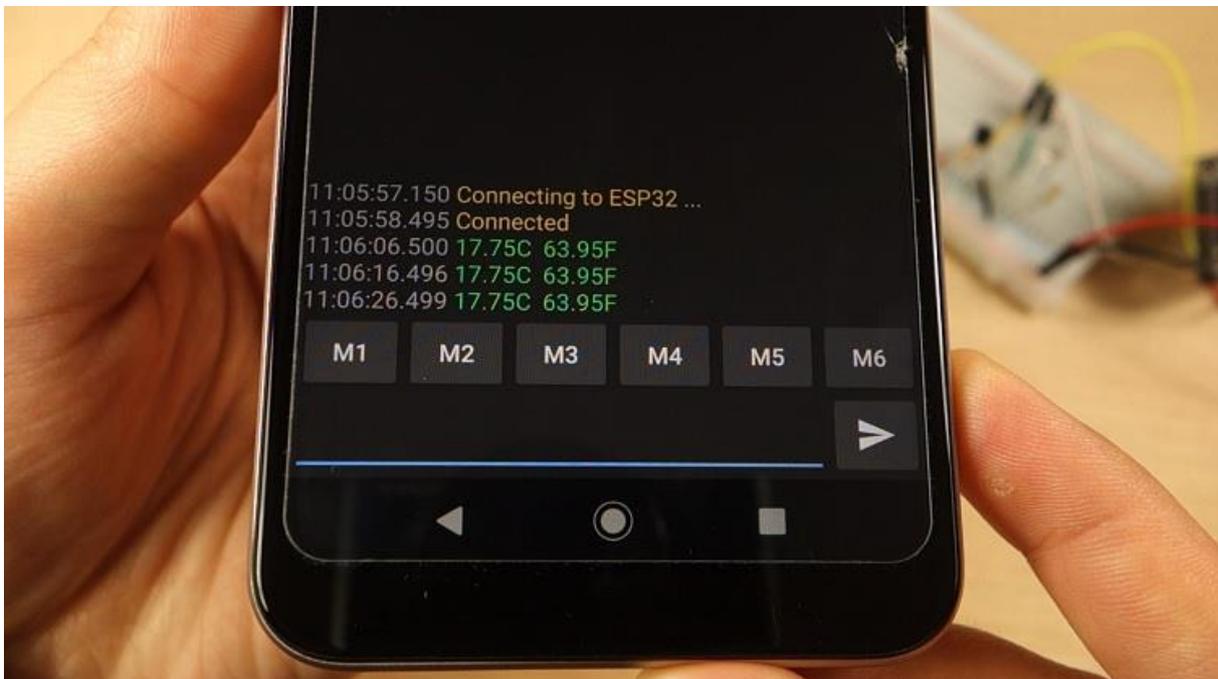


The application has several buttons in which you can save default messages. For example, you can associate **M1** with the **"led_on"** message, and **M2** with the **"led_off"** message.



Now, you are able to control the ESP32 GPIOs.

At the same time, you should be receiving the temperature readings every 10 seconds.



Wrapping Up

In summary, the ESP32 supports BLE and Bluetooth Classic. Using Bluetooth Classic is as simple as using a serial communication and its functions.

MODULE 6

LoRa Technology with ESP32

Unit 1 - ESP32 with LoRa Introduction

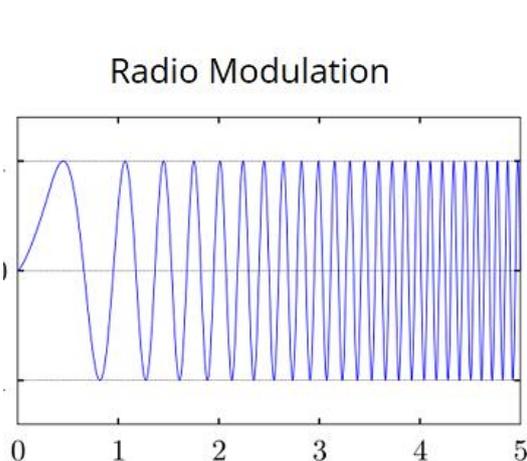


In this Unit we'll explore the basic principles of LoRa, and how it can be used with the ESP32 for IoT projects.

Before getting started, note that LoRa is a quite vast topic that has 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.



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

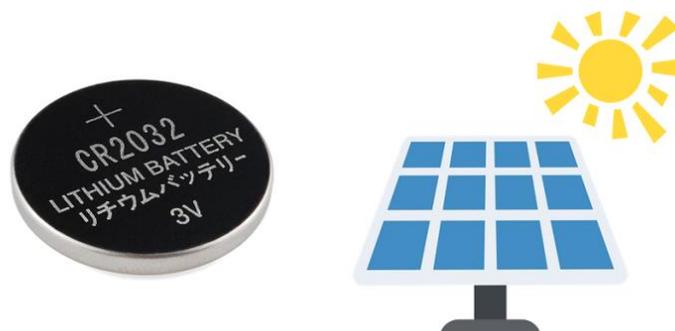
Because these bands are unlicensed, anyone can freely use them without paying or having to get a license. Check the [frequencies used in your country here](#).

LoRa Applications

LoRa long range and low power features, makes it perfect for battery-operated sensors and low-power applications in: Internet of Things (IoT), smart home, machine-to-machine communication, and much more.



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



LoRa is Not a Good Option For...

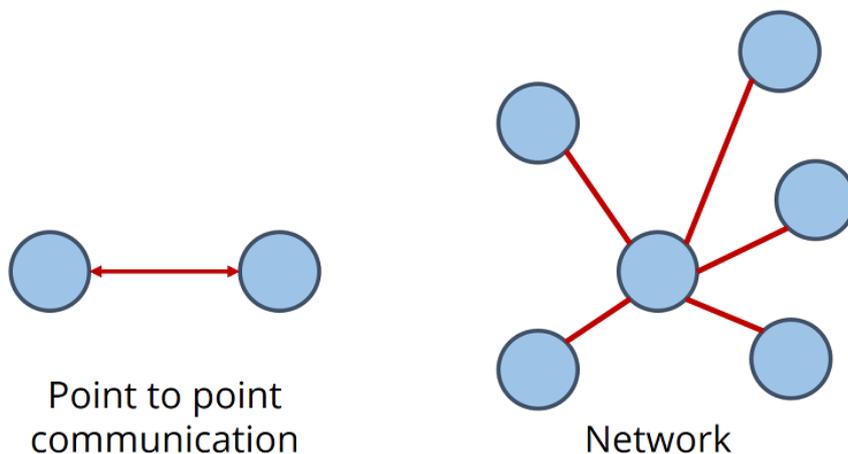


Keep in mind that LoRa is not suitable for projects that:

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

LoRa Topologies

You can use LoRa in:



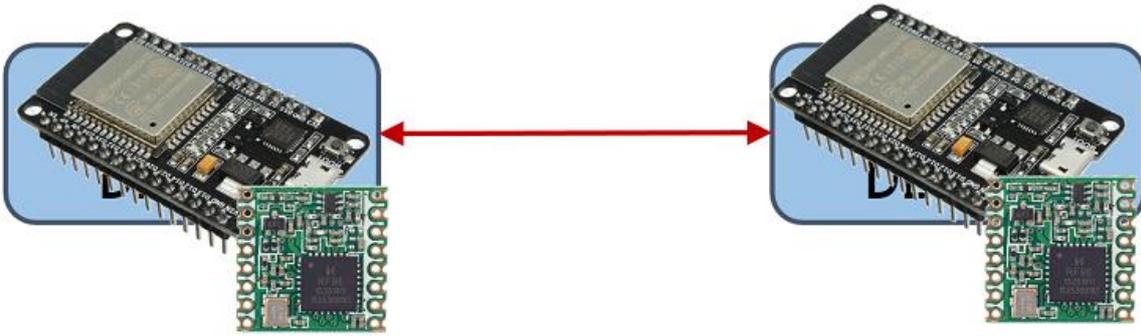
- Point to point communication
- Or build a LoRa network using LoRaWAN

Point to Point Communication

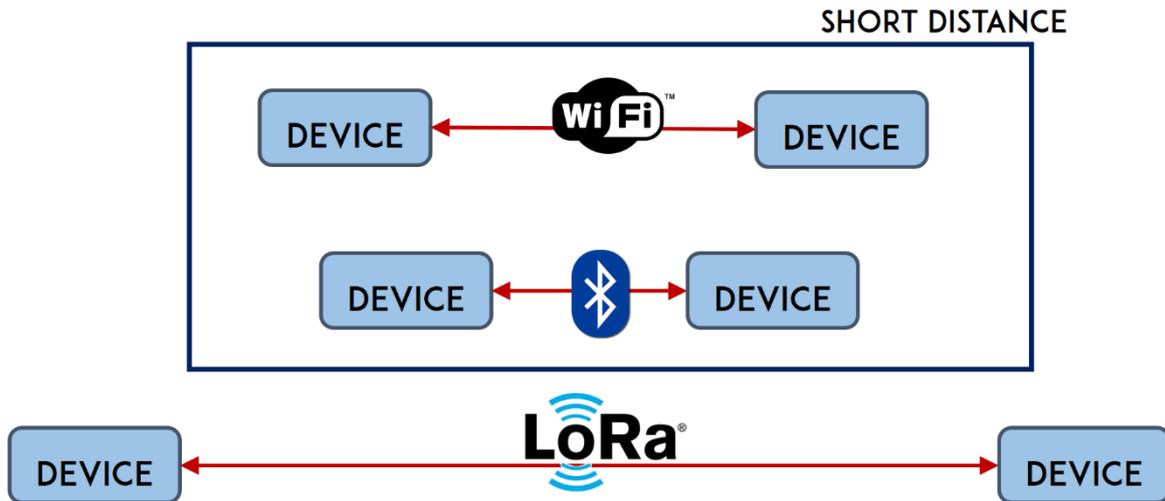
In point to point communication, two LoRa enabled devices talk with each other using RF signals.



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 distance. There are also other LoRa solutions that easily have a range of more than 30Km.

LoRaWAN

You can also build a LoRa network using LoRaWAN.



The LoRaWAN protocol is a Low Power Wide Area Network (LPWAN) specification derived from LoRa technology standardized by the [LoRa Alliance](#).

LoRaWAN Network Architecture

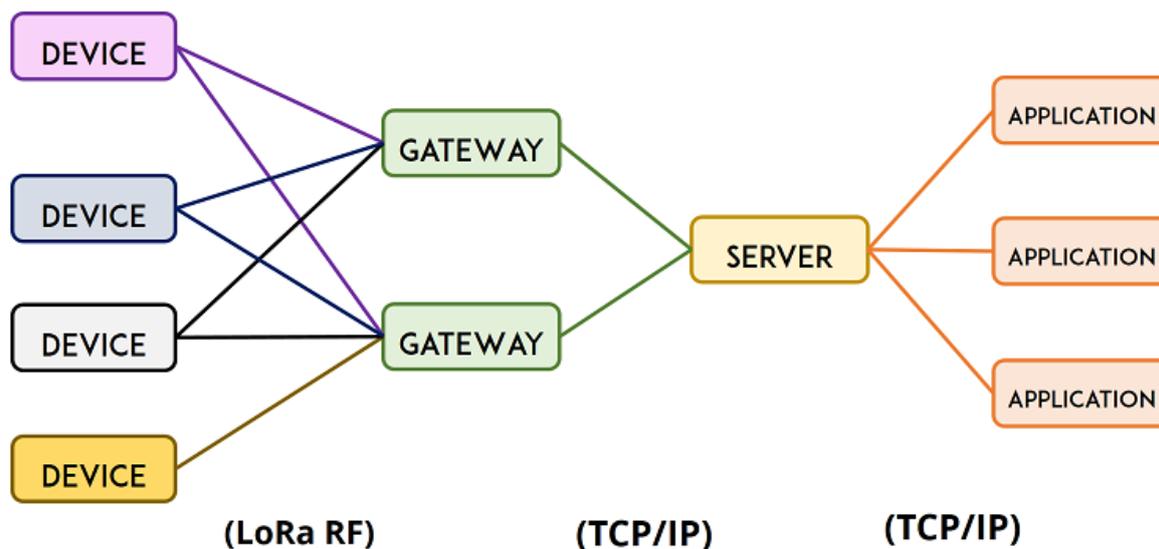
A typical LoRa network consists of four parts:

- 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.



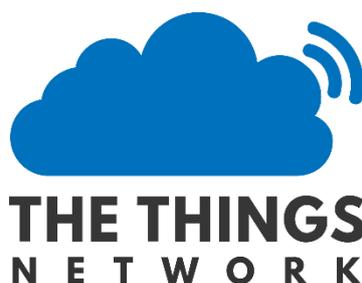
Network Options

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).



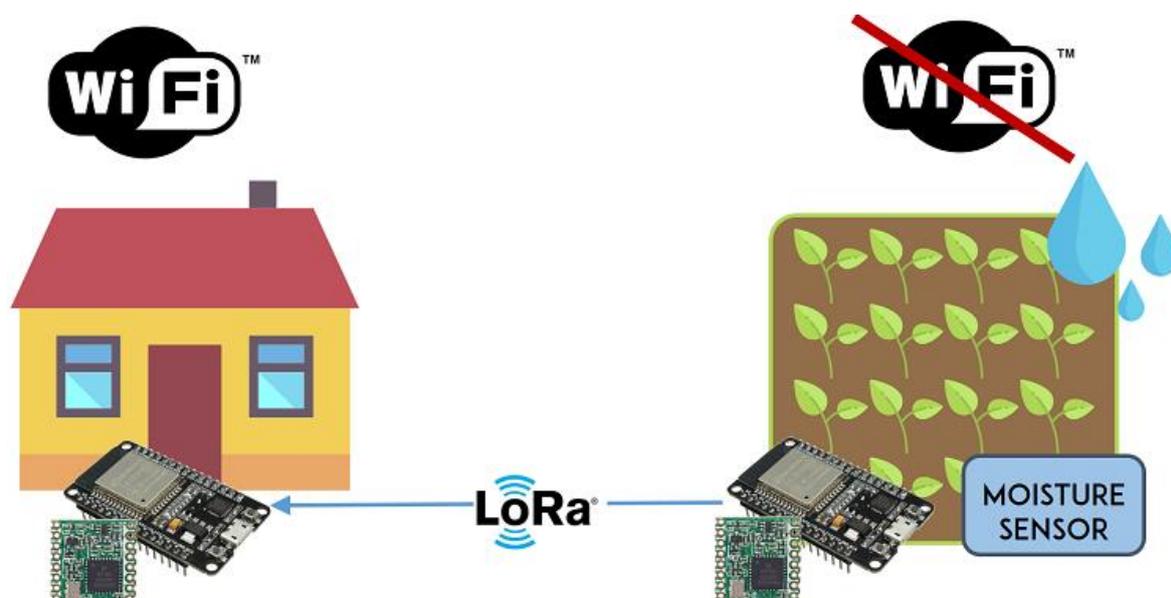
You can allow other people to use your gateway for long range connectivity, or you can use community gateways. If many people set up and share their gateways, we are able to cover wide areas and allow transmission of messages at longer distances using this protocol.



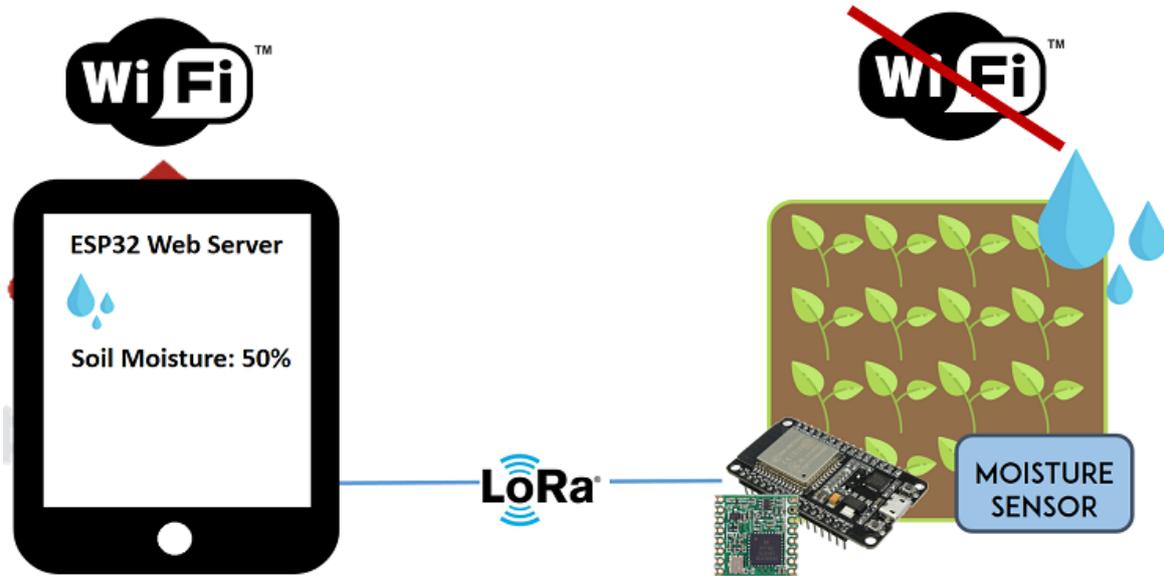
How can LoRa be useful in your home automation projects?

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.



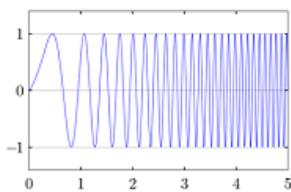
The later ESP32 has access to Wi-Fi, and it can run a web server that displays the moisture readings.



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



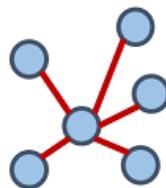
Small amounts of data



Low power consumption



Point to point communication



Network



Useful to monitor sensors without Wi-Fi coverage

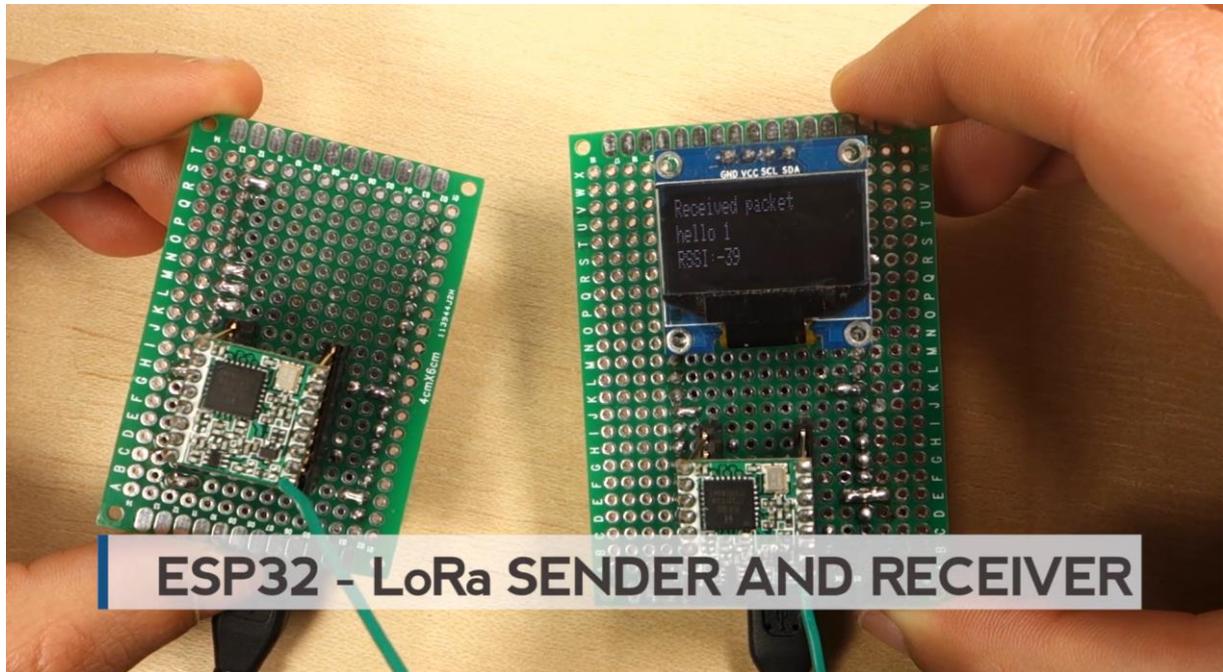
- LoRa is a radio modulation technique;
- LoRa allows long-distance communication of small amounts of data and requires low power;
- You can use LoRa in point to point communication or in a network using LoraWAN;
- LoRa can be especially useful if you want to monitor sensors that are not covered by your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

LoRa Further Reading

As we've mentioned at the beginning of the post, LoRa is a quite complex topic that would give a full course itself. So, here we provide some external further reading information if you want to take this subject further:

- [LoRa for IoT \(epanorama website\)](#)
- [LoRa vs LoRaWAN \(libelium website\)](#)
- [The Things Network Kickstarter page](#)
- [The limits of LoRaWAN \(paper\)](#)

Unit 2 - ESP32 – LoRa Sender and Receiver

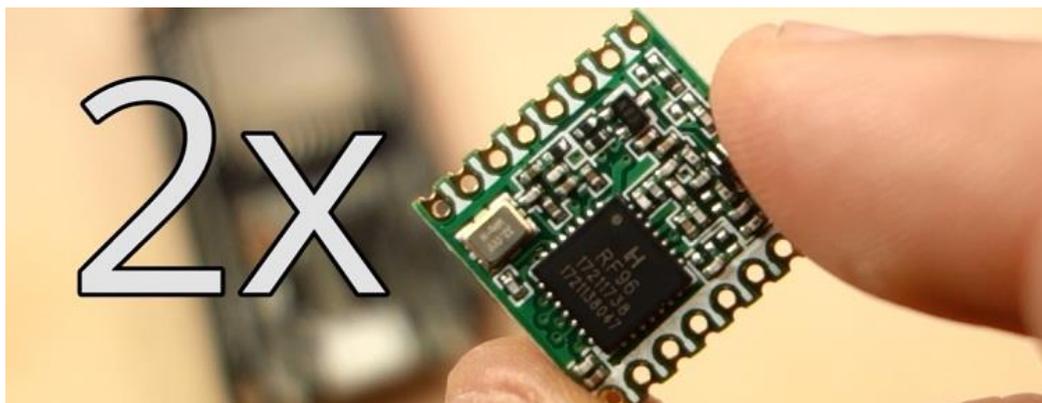


In this Unit you'll program two ESP32 boards to exchange data using LoRa RF signals. You'll also test the modules communication range.

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...

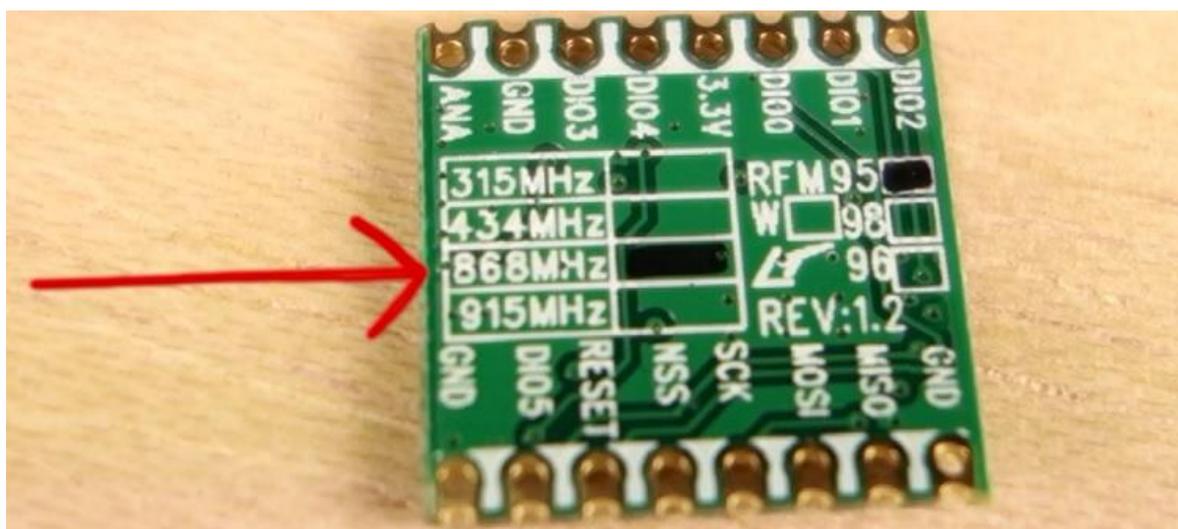
Alternatively, there are ESP32 boards with LoRa and OLED display built-in like the [ESP32 Heltec Wifi Module](#), or the [TTGO board](#).



We have a review and getting started guide for the TTGO LoRA SX1276 with built-in OLED:

- [Review](#)
- [Getting Started Guide](#)

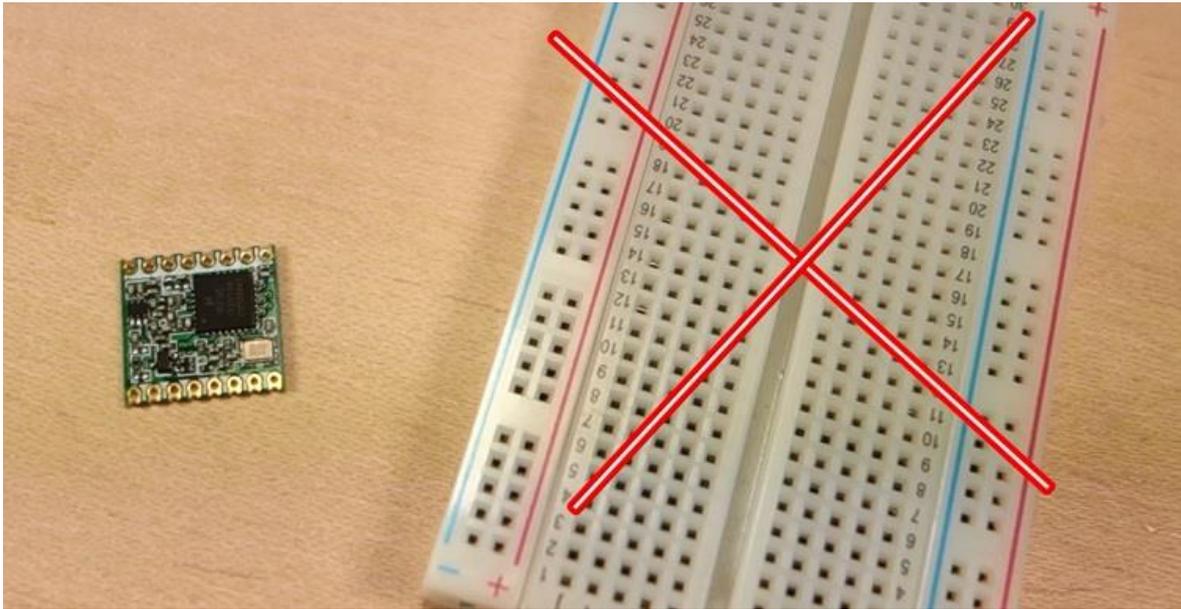
Before getting your LoRa transceiver module, make sure you check the correct frequency for your location. You can visit the following web page to learn more about [RF signals and regulations according to each country](#). For example, in Portugal we can use a frequency between 863 and 870 MHz or we can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



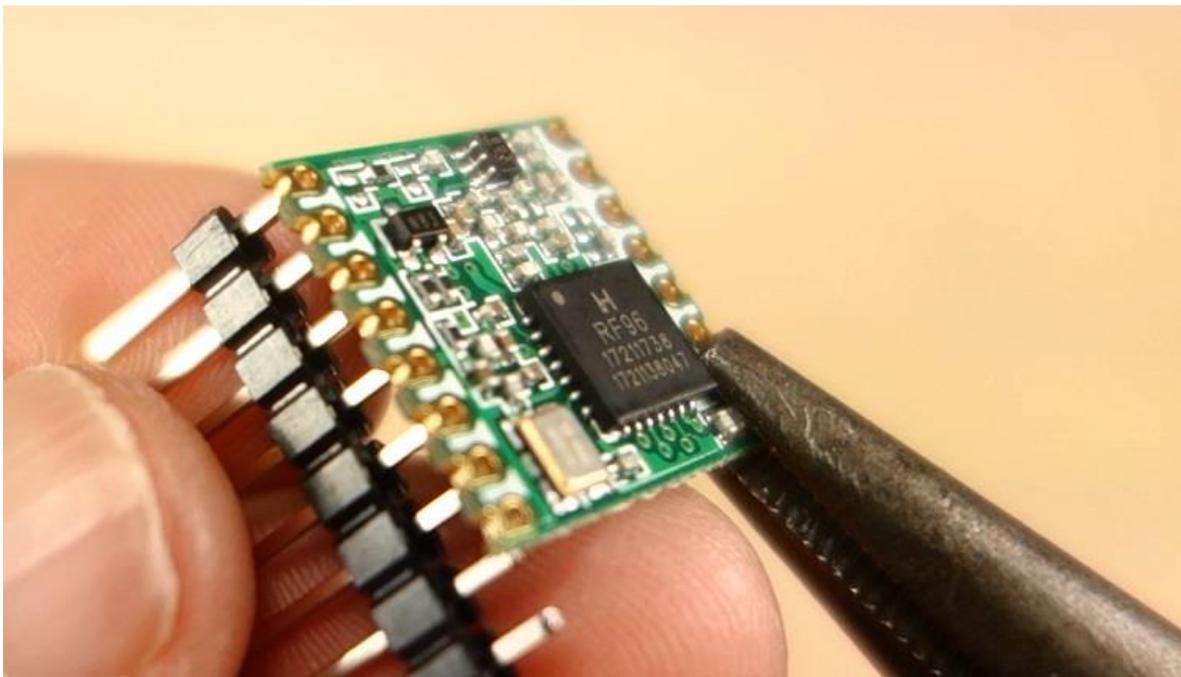
In the RFM95 module, the frequency is marked at the back as you can see in the figure above. Make sure you are compliant with your country's regulation and buy the right board.

Preparing the RFM95 Transceiver Module

The RFM95 transceiver isn't breadboard friendly.



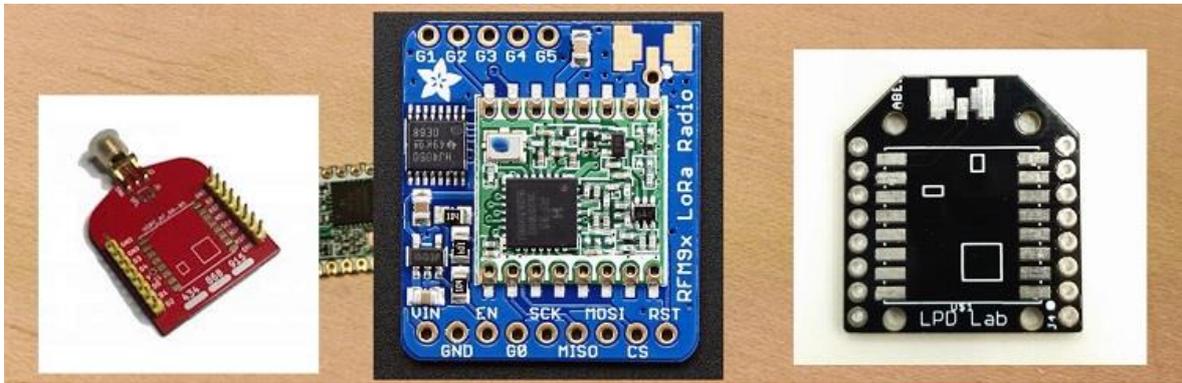
A common row of 2.54mm header pins won't fit on the transceiver pins. The spaces between the connections are smaller than usual.



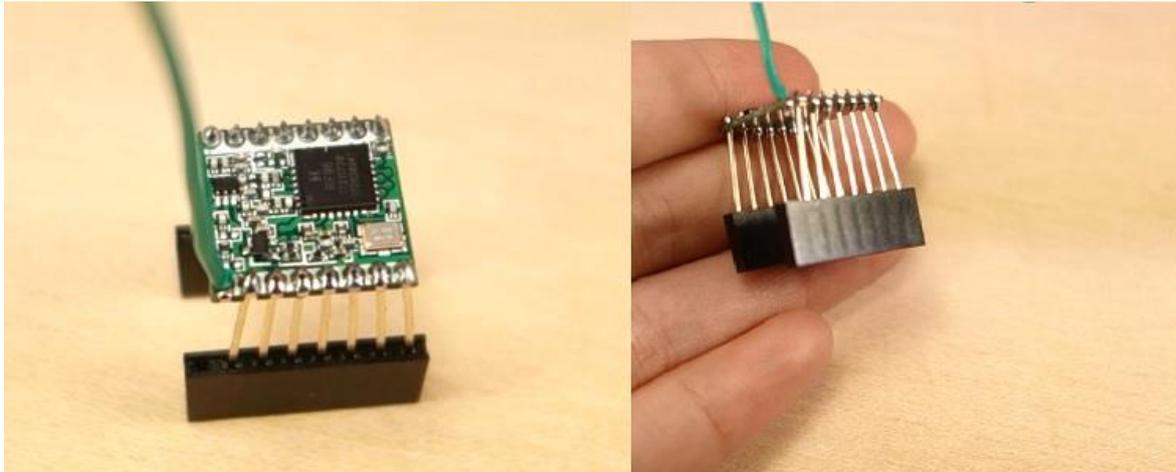
There are a few options that you can use to access the transceiver pins:

- 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.

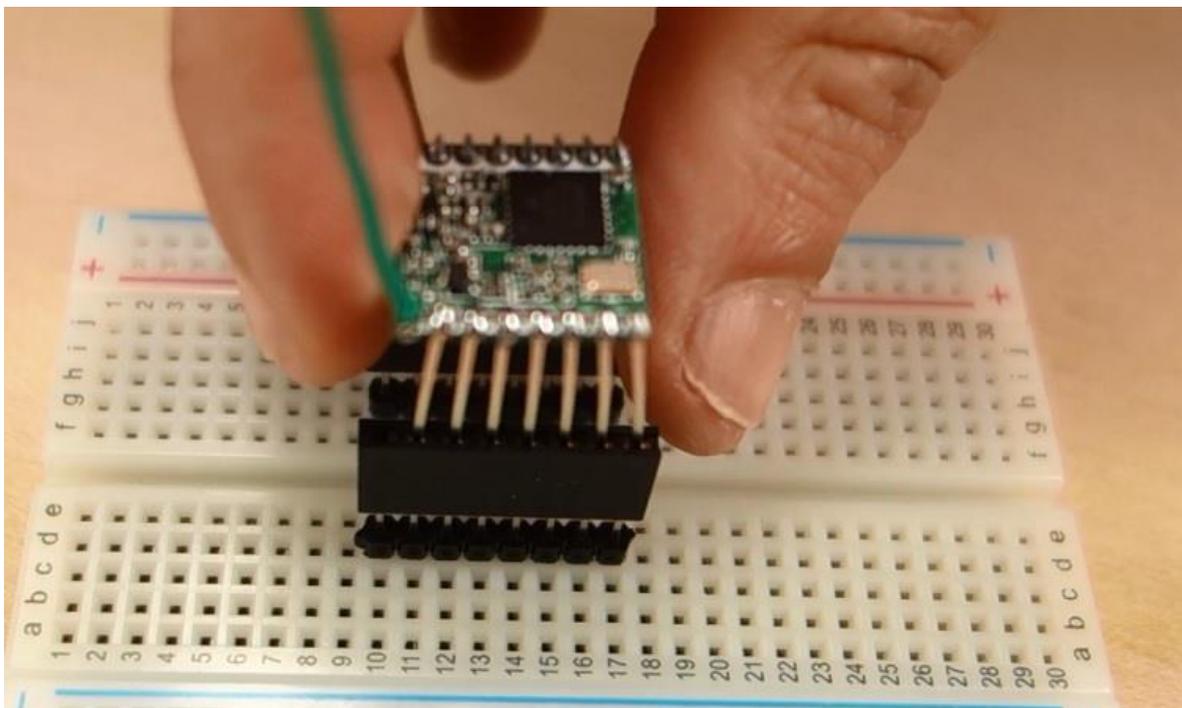
The figure below shows some examples of breakout boards for the RFM95.



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

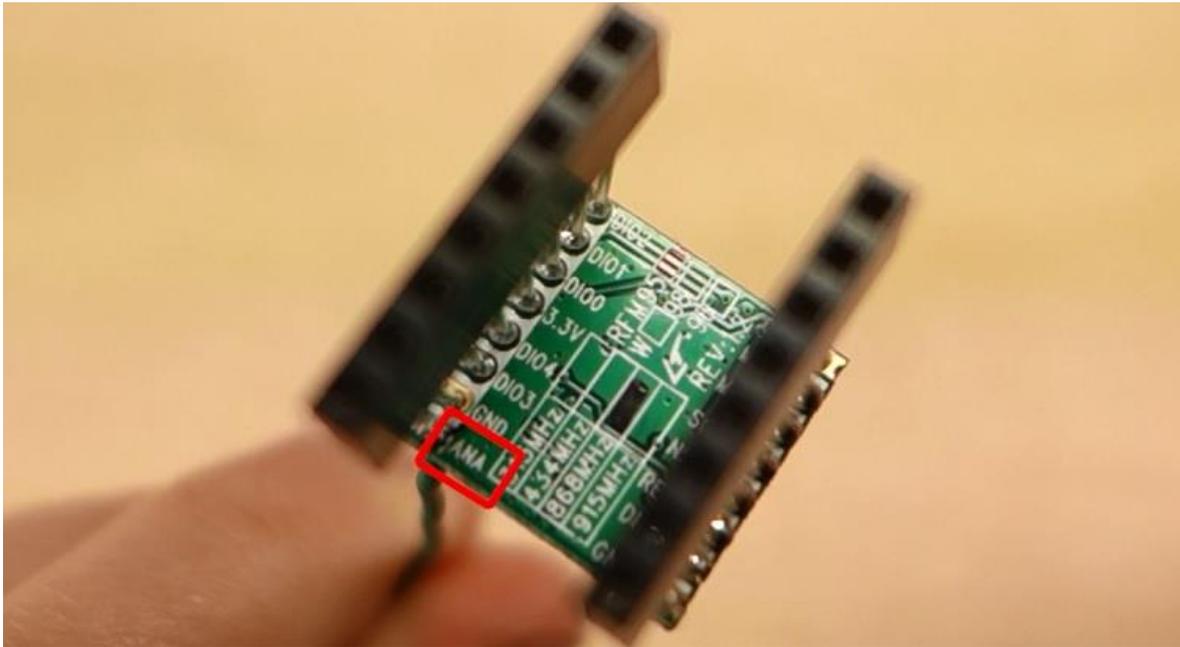


This way you can access the module's pins with regular jumper wires, or even put some header pins to connect them directly to a stripboard or breadboard.

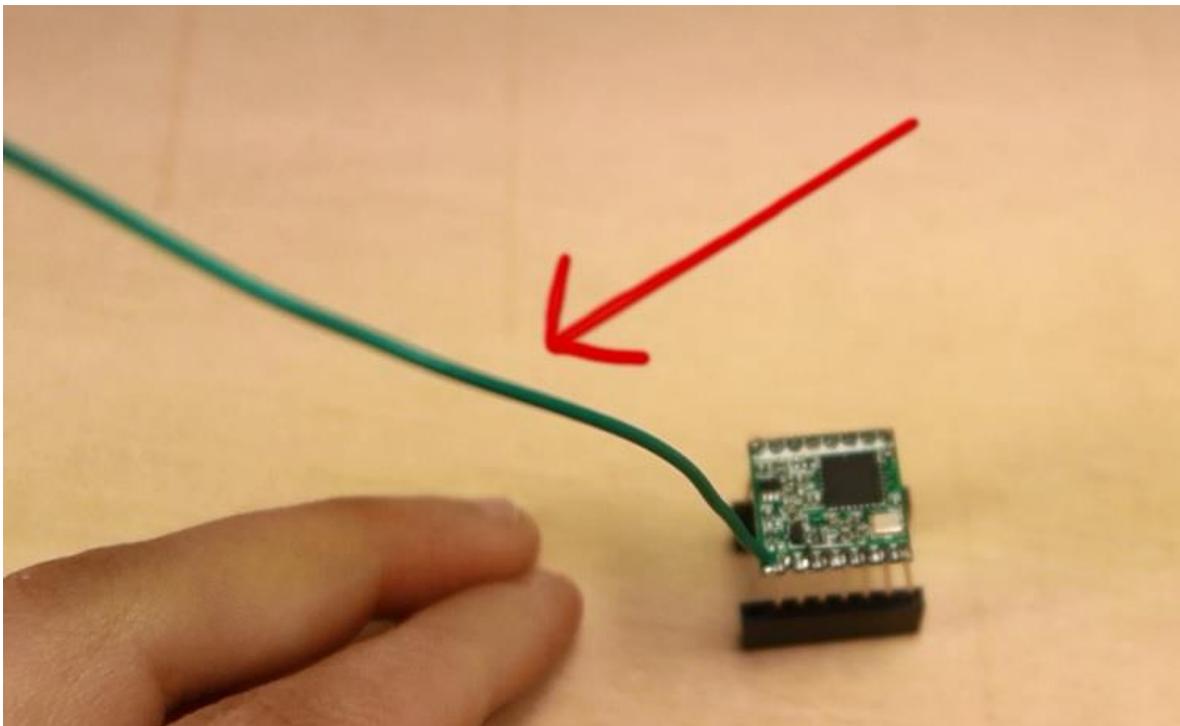


Antenna

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



Important: you MUST attach an antenna to the module.

Preparing the LoRa sender

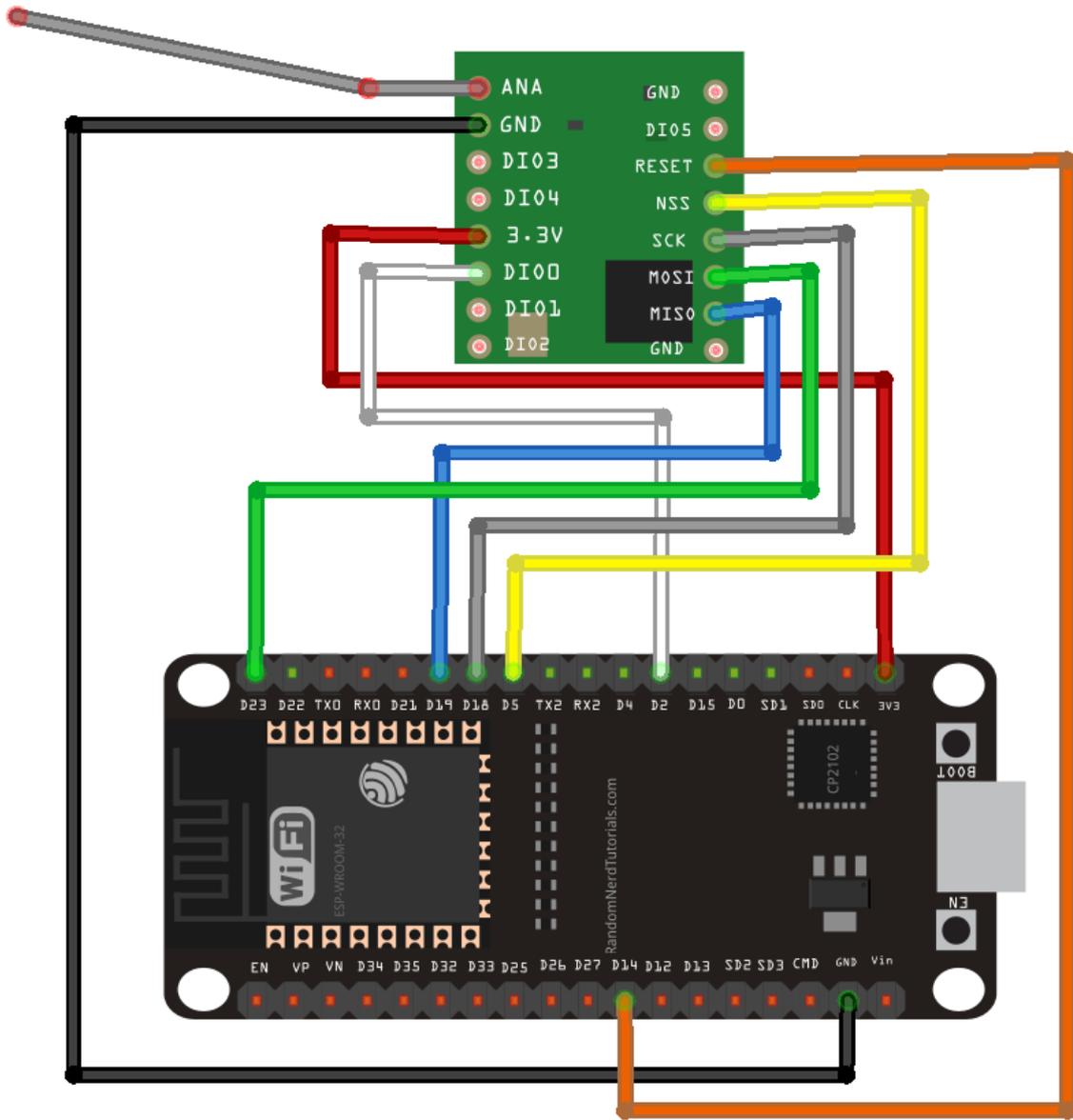
This project is divided into two parts. First you'll setup and program the sender. Then, you'll prepare the receiver.

Schematic

Let's build the sender circuit. Having 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 SPI communication protocol, so we'll be using the ESP's SPI pins. Wire the ESP32 to the transceiver module as shown in the following schematic diagram.

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](#)

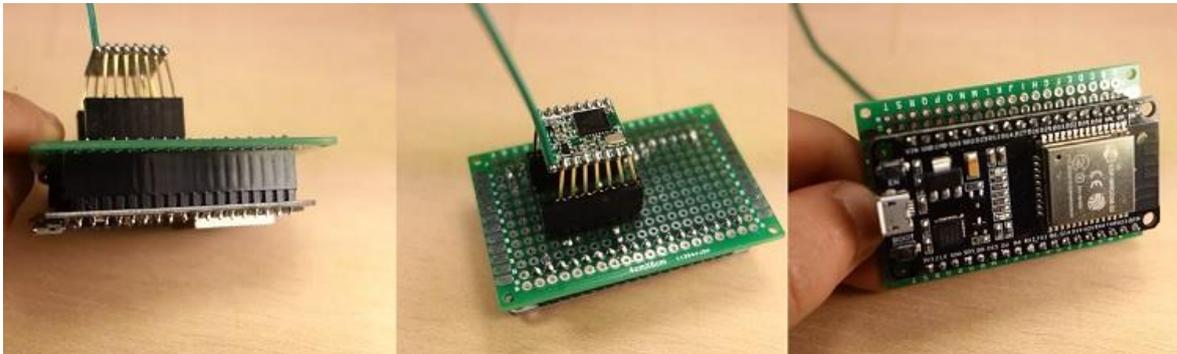


This schematic diagram is a bit confusing, so you can take a look at the following table instead.

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.

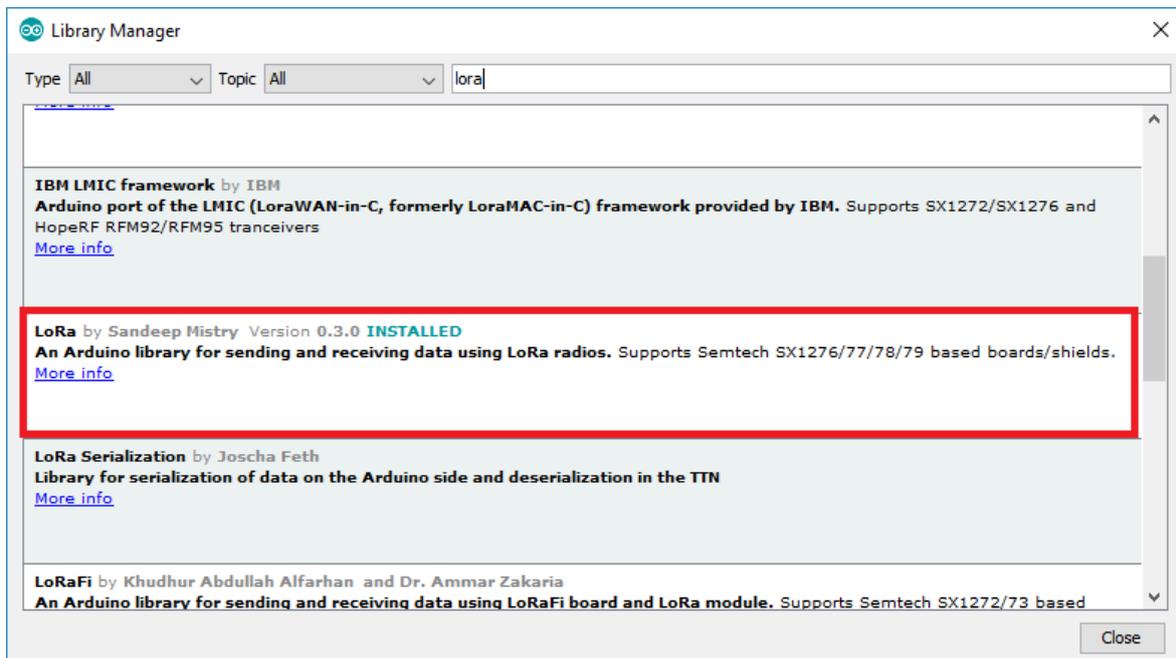
For practical reasons we've made this circuit on a stripboard. It's easier to handle, and the wires don't disconnect. You may use a breadboard if you prefer.



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 be using the [arduino-LoRa library by sandeep mistry](#).

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



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.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/LoRa_RFM95/LoRa_Sender/LoRa_Sender.ino

```
/*  
  Rui Santos  
  Complete project details at http://randomnerdtutorials.com  
*/  
  
#include <SPI.h>  
#include <LoRa.h>  
  
//define the pins used by the transceiver module  
#define ss 5  
#define rst 14  
#define dio0 2  
  
int counter = 0;  
  
void setup() {  
  //initialize Serial Monitor  
  Serial.begin(115200);  
  while (!Serial);  
  Serial.println("LoRa Sender");  
  
  //setup LoRa transceiver module  
  LoRa.setPins(ss, rst, dio0);  
  
  //replace the LoRa.begin(---E-) argument with your location's 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);  
}
```

Let's take a quick look at the code.

It starts by including the needed libraries.

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

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

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

You initialize the counter variable that starts at 0;

```
int counter = 0;
```

setup()

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

```
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();
```

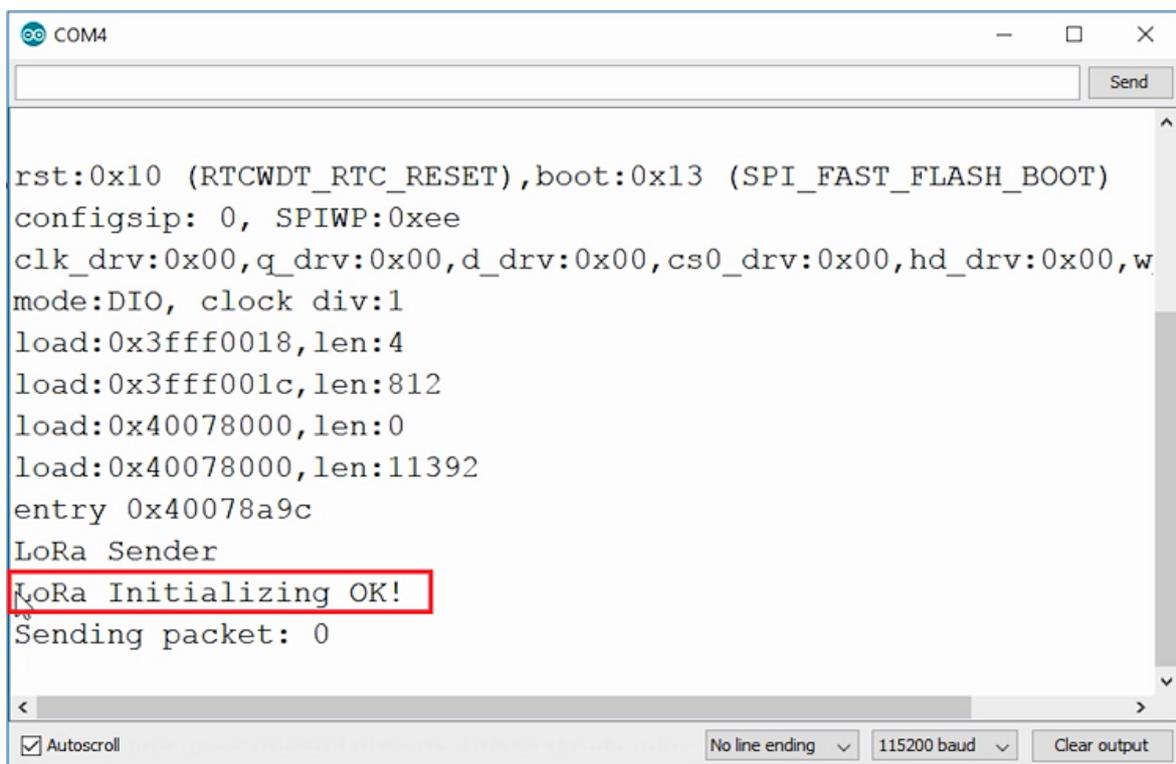
After this, the counter message is incremented by one in every loop which happens every 10 seconds.

```
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.

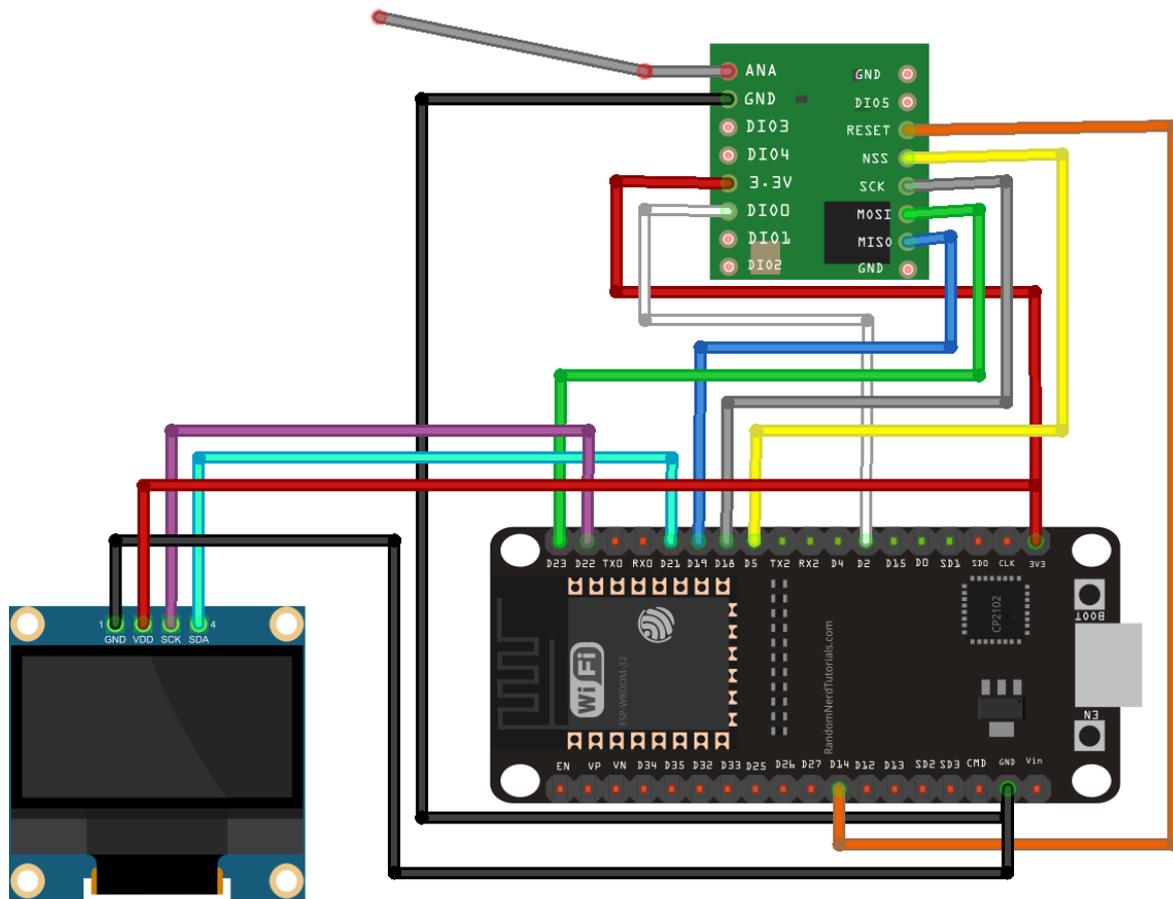


```
COM4  
rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)  
config:0, SPIWP:0xee  
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,w  
mode:DIO, clock div:1  
load:0x3fff0018,len:4  
load:0x3fff001c,len:812  
load:0x40078000,len:0  
load:0x40078000,len:11392  
entry 0x40078a9c  
LoRa Sender  
LoRa Initializing OK!  
Sending packet: 0
```

After making sure the LoRa initializes ok, let's prepare the LoRa receiver.

Preparing 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.



(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.)

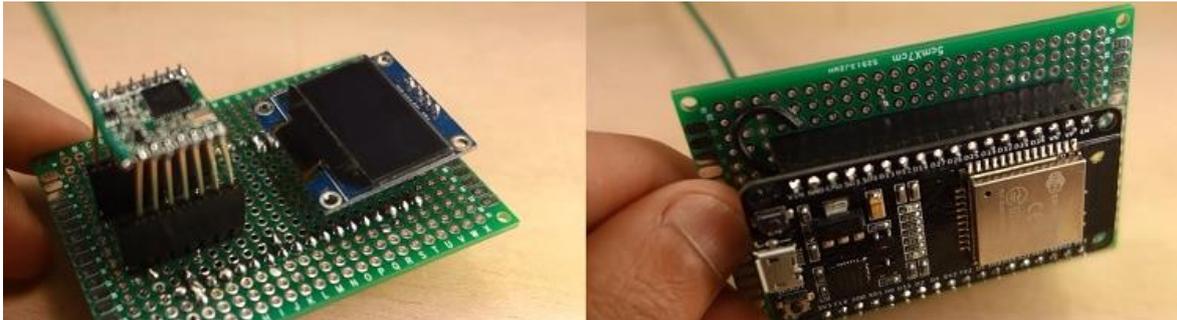
You can also take a look at the next table to check the connections between the ESP32 and the transceiver.

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

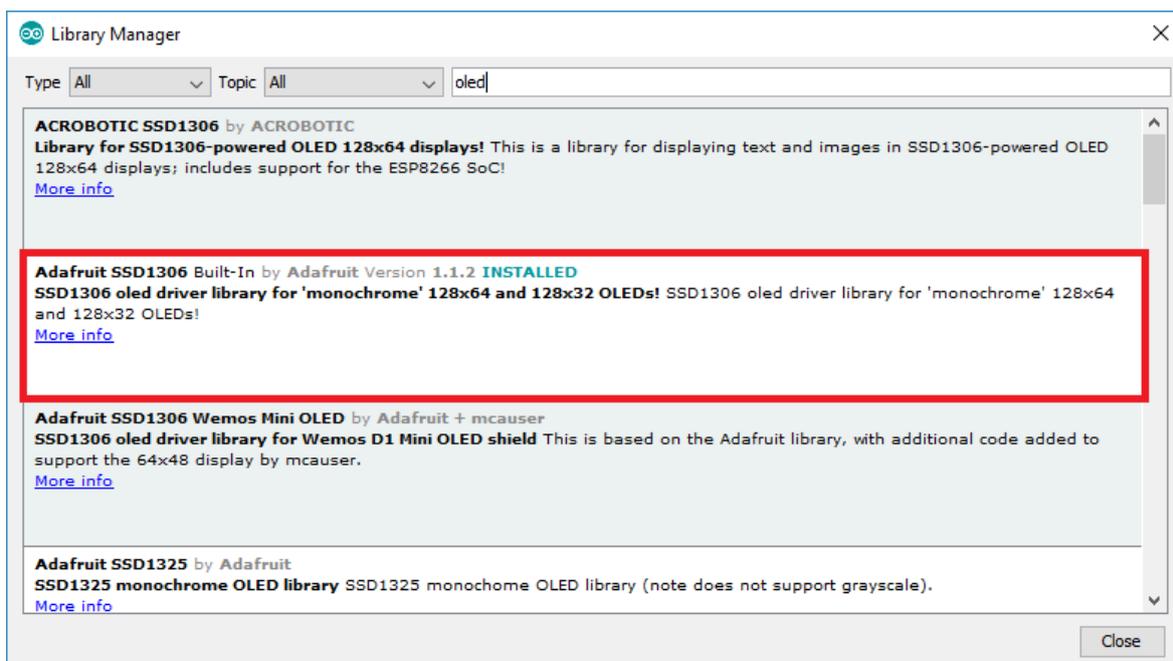
For practical reasons we've made this circuit on a stripboard. The figure below shows how it looks.



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** and search for "OLED". Select the "**Adafruit SSD1306**" library and install it.



The Receiver Sketch

After having the required libraries, copy the receiver sketch to your Arduino IDE. This sketch simply listens for LoRa packets within its range and prints the content of the

packets on the OLED display, as well as the RSSI. The RSSI measures the relative received signal strength.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/LoRa_RFM95/LoRa_Receiver/LoRa_Receiver.ino

```
#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
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 rrsi = LoRa.packetRssi();
Serial.print(" with RSSI ");
Serial.println(rrsi);
display.setCursor(0,20);

display.print("RSSI: ");
display.setCursor(30,20);
display.print(rrsi);
display.display();
}
}

```

Let's take a look at this code.

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 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.

```

#define OLED_RESET 4
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

```

setup()

In the setup() initialize the display

```

if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)){
    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.

```
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 a new packet has been received 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();
```

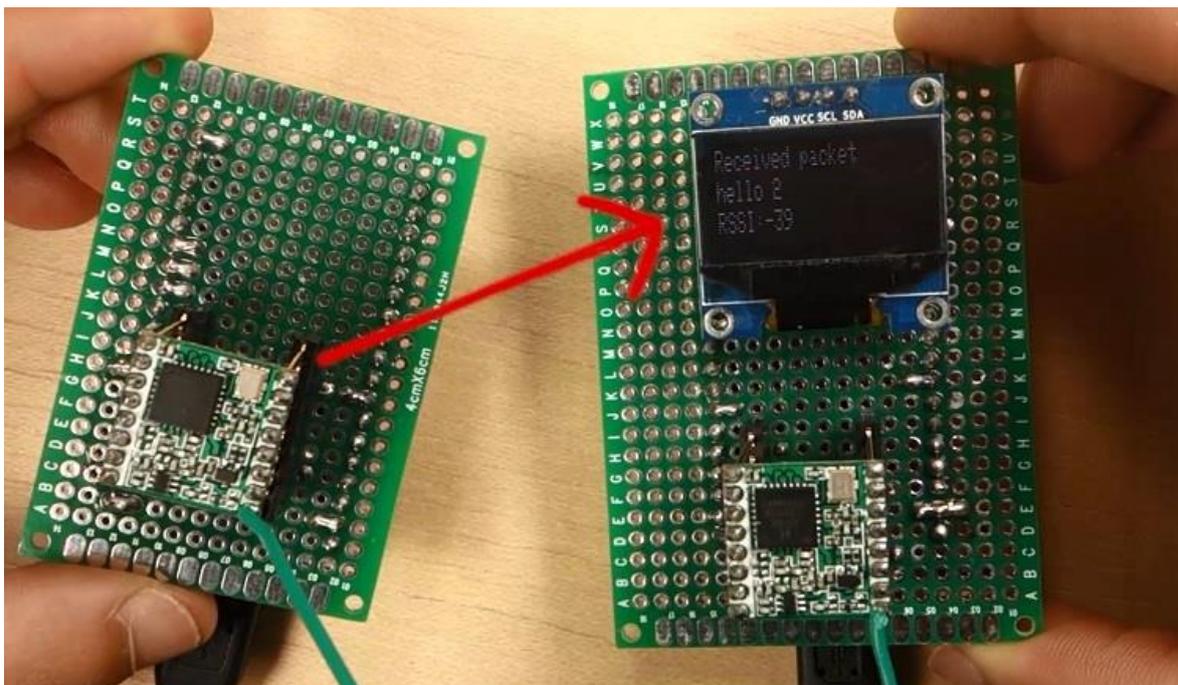
Finally, the next few lines of code print the RSSI of the received packet in dB.

```
int rrsi = LoRa.packetRssi();  
Serial.print(" with RSSI ");  
Serial.println(rrsi);  
display.setCursor(0,20);  
display.print("RSSI: ");  
display.setCursor(30,20);  
display.print(rrsi);  
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 get a stable communication between the two ESPs up to 250 meters. This will greatly vary depending if you are in an urban or rural area, if there are a lot of buildings in between, etc.



Additionally, note that we're using the library's default definitions for LoRa, you may get a wider communication range by changing some settings. We'll not explore this topic in this Unit, but there's a [discussion on github on to to configure the LoRa library for better range](#). Feel free to take a look.

Wrapping Up

In summary:

In this project you've sent a message using LoRa between two ESP32 boards. Keep in mind 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.
- Additionally, 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 you'll be able to get it. So, you need something that identifies your messages.

That's it for now. We hope you're now a bit more familiar with LoRa and can you use it in your ESP32 projects.

Unit 3 - Further Reading about LoRa Gateways

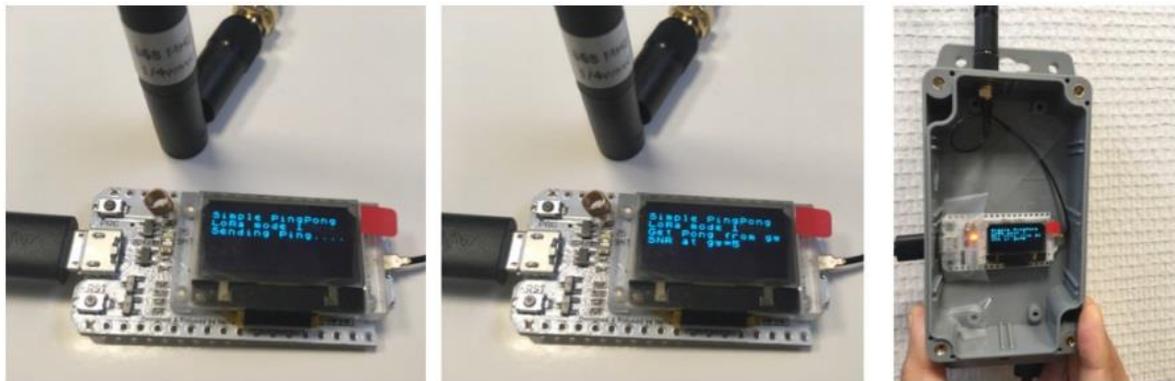
In this Unit will give some extra information about LoRa gateways. We won't explore how to build a LoRa/LoRaWAN gateway in this course, but we'll provide you some information you can explore if you really want to build one.

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 parts.

1. [LoRa Gateway with ESP32](#)



[View source](#)

You can build a LoRa gateway with the ESP32 using the RFM95 transceiver module we've used previously, or an ESP32 with LoRa built-in, and programming 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 how to build 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

Unit 4 - LoRa – Where to Go Next?

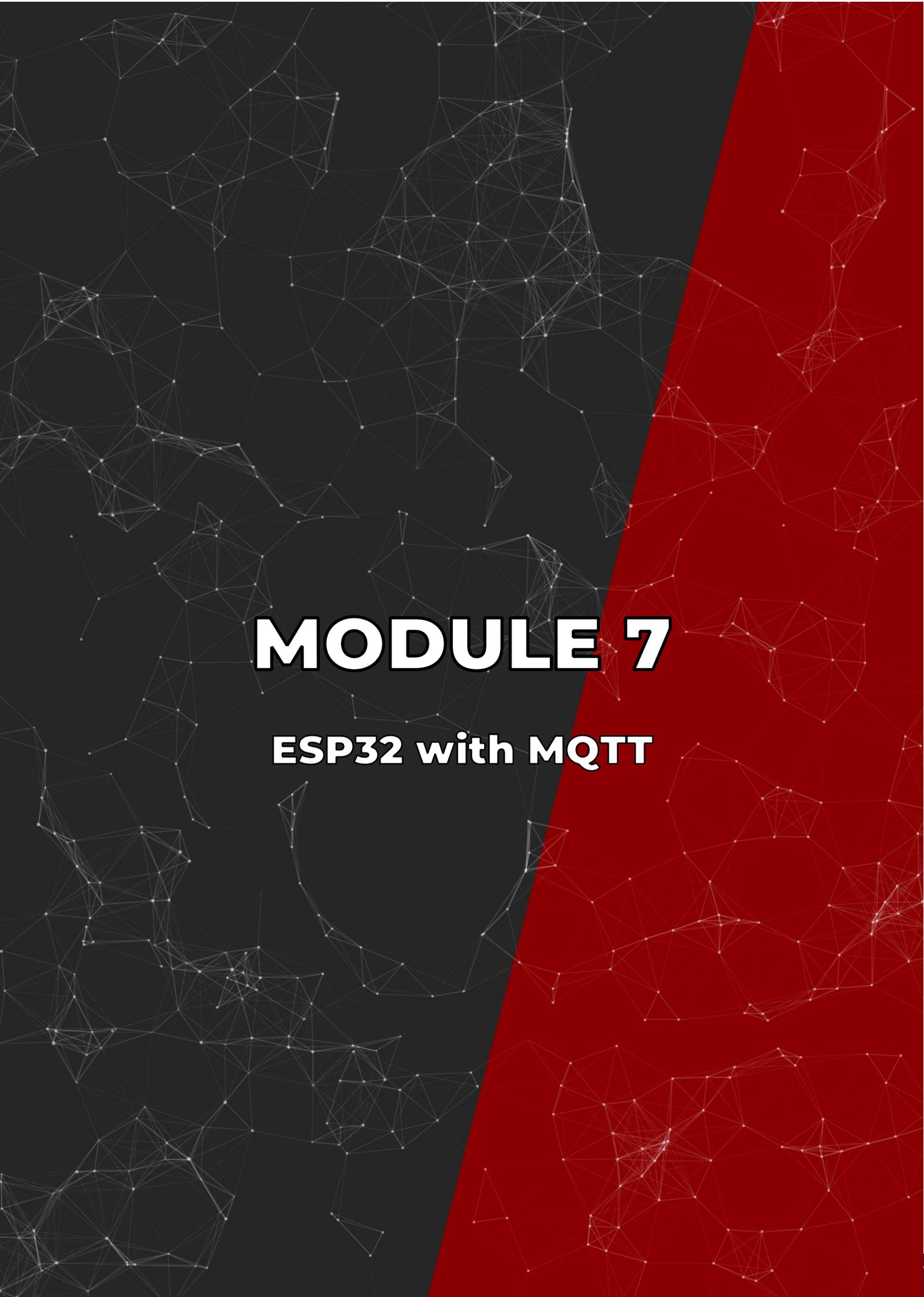
In this Module we've taken a look at the LoRa technology. It is a quite powerful technology for IoT as it allows long range communication with low power consumption.



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 located outside your Wi-Fi network (up to 250m from your house) and publish those readings on a web server.



MODULE 7

ESP32 with MQTT

Unit 1 - ESP32 with MQTT:

Introduction



Introducing MQTT

This Unit is an introduction to MQTT and how it can be used with the ESP32.

MQTT stands for **M**essage **Q**ueuing **T**elemetry **T**ransport. It is a lightweight publish and subscribe system where you can publish and receive messages as a client.



MQTT is a simple messaging protocol, designed for constrained devices with low-bandwidth. So, it's the perfect solution for Internet of Things applications. MQTT allows you to send commands to control outputs, read and publish 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

Publish/Subscribe

The first concept is the publish and 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.



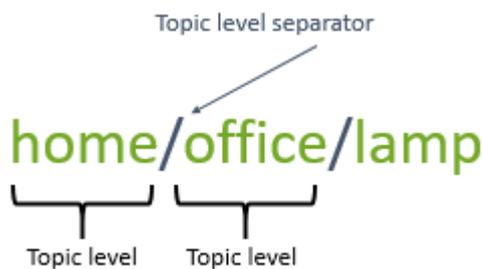
- For example Device 1 publishes on a topic;
- Device 2 is subscribed to the same topic that Device 1 is publishing in;
- So, Device 2 receives the message.

Messages are pieces of information exchanged between your devices: whether it's a command or data.

Topics

Another important concept are the topics. Topics are the way you register interest for incoming messages or how you specify where you want to publish the messages.

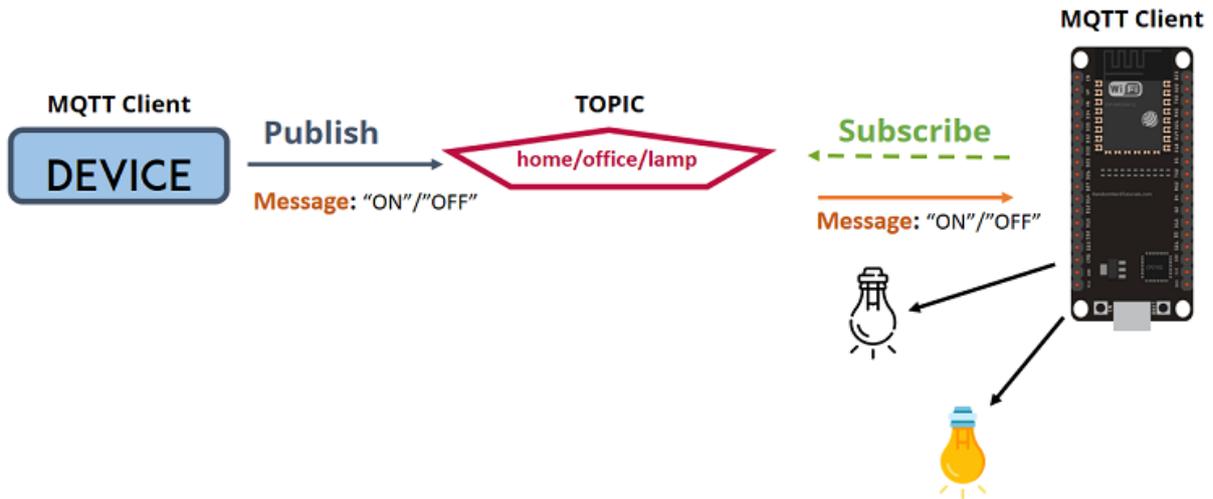
Topics are represented with strings separated by a forward slash. Each forward slash indicates the topic level. Here's an example on how you would create a topic for a lamp in your home office:



Note: topics are case-sensitive, which makes the following topics different.

home/office/lamp
≠
Home/Office/Lamp

If you would like to turn on a lamp in your home office using MQTT and the ESP32 you can imagine the following scenario:



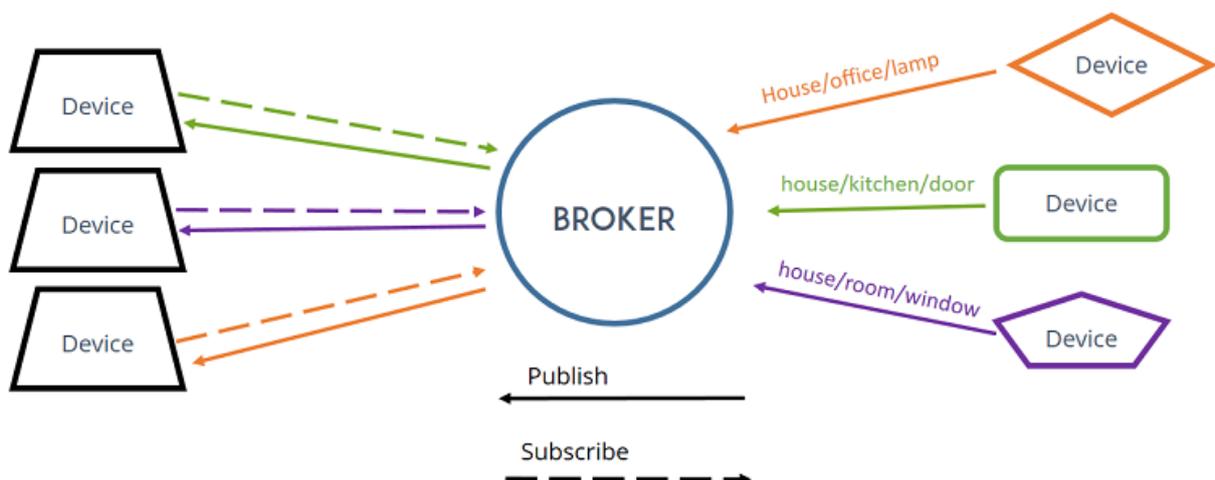
1. You have a device that publishes "on" and "off" messages on the home/office/lamp topic.
2. The ESP32 that controls your lamp, is subscribed to that topic.
3. So, when a new message is published on that topic, the ESP32 receives the "on" or "off" message and turns the lamp on or off.

This first device, can be an ESP32, an ESP8266, or an Home Automation controller platform like Node-RED, Home Assistant, Domoticz, or OpenHAB, for example.



Broker

Finally, you also need to be aware of the term broker. The broker is primarily responsible for receiving all messages, filtering the messages, decide who is interested in them and then send the messages to all subscribed clients.



There are several brokers you can use. For example, you can use the Mosquitto broker hosted on a Raspberry Pi, or you can use a cloud MQTT broker.



We're going to use the Mosquitto broker installed in a Raspberry Pi .

Wrapping Up

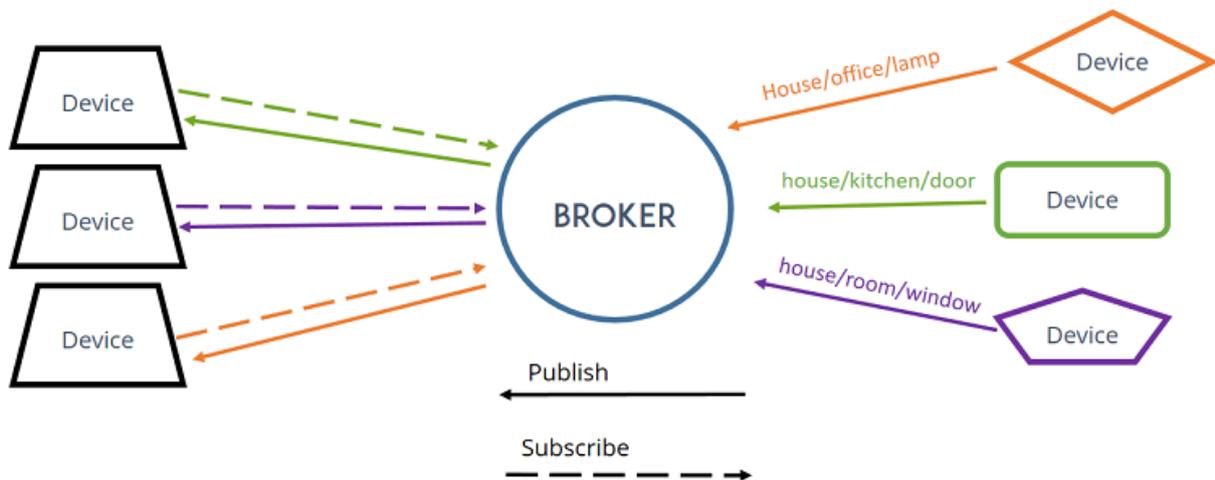
In summary:

- MQTT is a communication protocol very useful in 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.

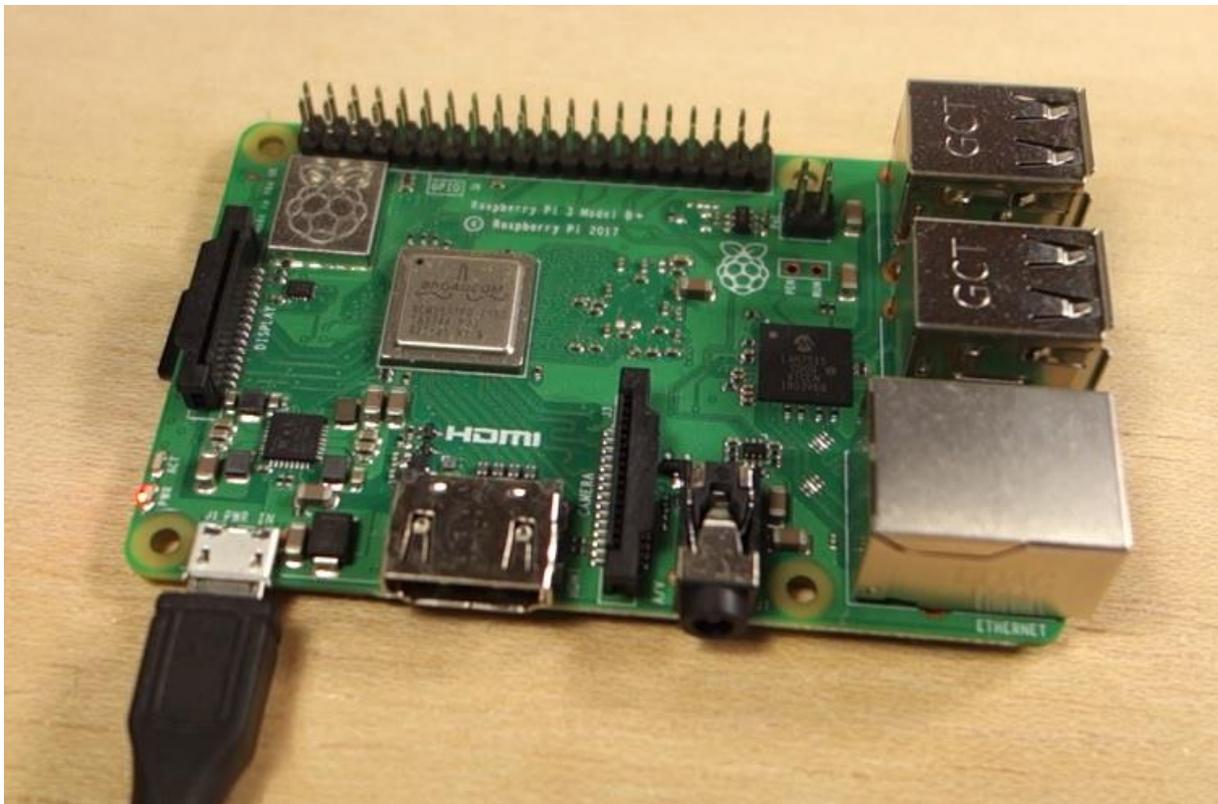
In the next three Units, you'll learn how to install an MQTT broker and exchange data between two ESP32 boards using MQTT protocol.

Unit 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 message 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:

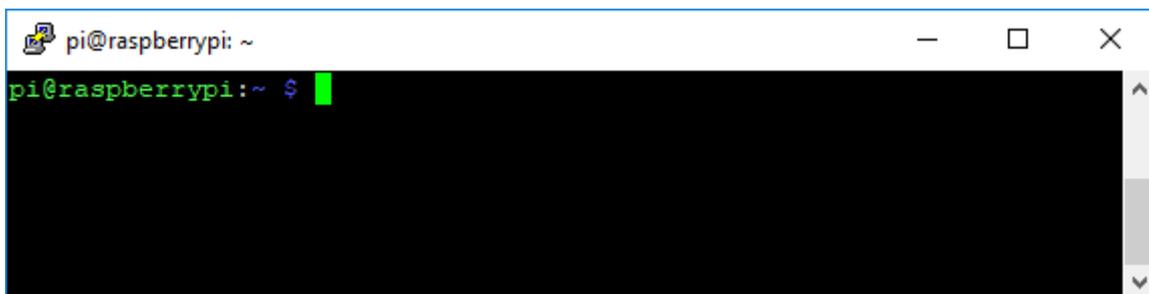
1. You should be familiar with the Raspberry Pi board – [read Getting Started with Raspberry Pi](#);
2. You should have the Raspbian or Raspbian Lite operating system installed in your Raspberry Pi – [read Installing Raspbian Lite, Enabling and Connecting with SSH](#);
3. You also need the following hardware:
 - [Raspberry Pi board](#) – read [Best Raspberry Pi Starter Kits](#)
 - [MicroSD Card – 16GB Class10](#)
 - [Raspberry Pi Power Supply \(5V 2.5A\)](#)

After having your Raspberry Pi board prepared with Raspbian OS, you can continue with this Unit. Let's install the Mosquitto Broker.



Installing Mosquitto Broker on Raspbian OS

Open a new Raspberry Pi terminal window:



To install the Mosquitto Broker enter these next commands:

```
pi@raspberrypi:~ $ sudo apt update  
pi@raspberrypi:~ $ sudo apt install -y mosquitto mosquitto-clients
```

You'll have to type **Y** and press **Enter** to confirm the installation. To make Mosquitto auto start on boot up enter:

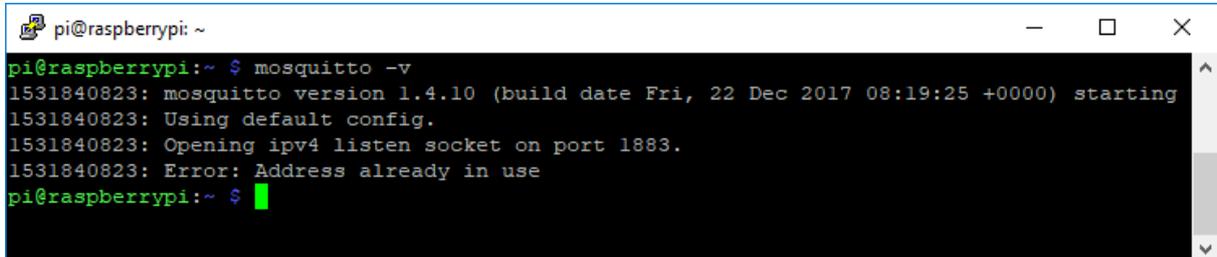
```
pi@raspberrypi:~ $ sudo systemctl enable mosquitto.service
```

Testing Installation

Send the command:

```
pi@raspberrypi:~ $ mosquitto -v
```

This returns the Mosquitto version that is currently running in your Raspberry Pi. It should be 1.4.X or above.



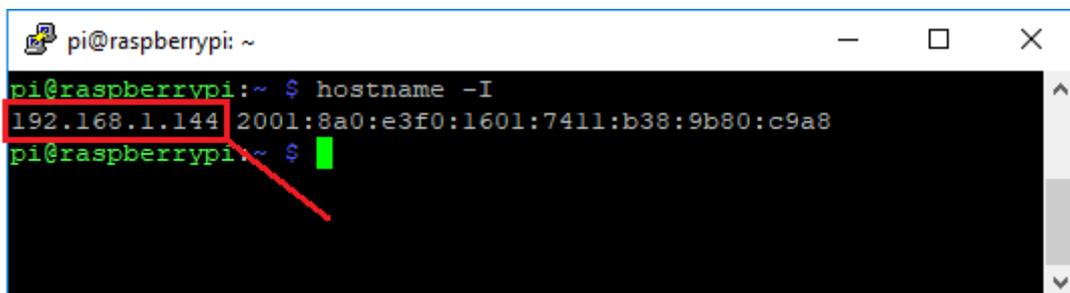
```
pi@raspberrypi:~ $ mosquitto -v
1531840823: mosquitto version 1.4.10 (build date Fri, 22 Dec 2017 08:19:25 +0000) starting
1531840823: Using default config.
1531840823: Opening ipv4 listen socket on port 1883.
1531840823: Error: Address already in use
pi@raspberrypi:~ $
```

Note: sometimes the command `mosquitto -v` prompts a warning message saying *"Error: Address already in use"*. That warning message means that your Mosquitto Broker is already running, so don't worry about that.

Raspberry Pi IP Address

To retrieve your Raspberry Pi IP address, type the next command in your Terminal window:

```
pi@raspberrypi:~ $ hostname -I
```



```
pi@raspberrypi:~ $ hostname -I
192.168.1.144 2001:8a0:e3f0:1601:7411:b38:9b80:c9a8
pi@raspberrypi:~ $
```

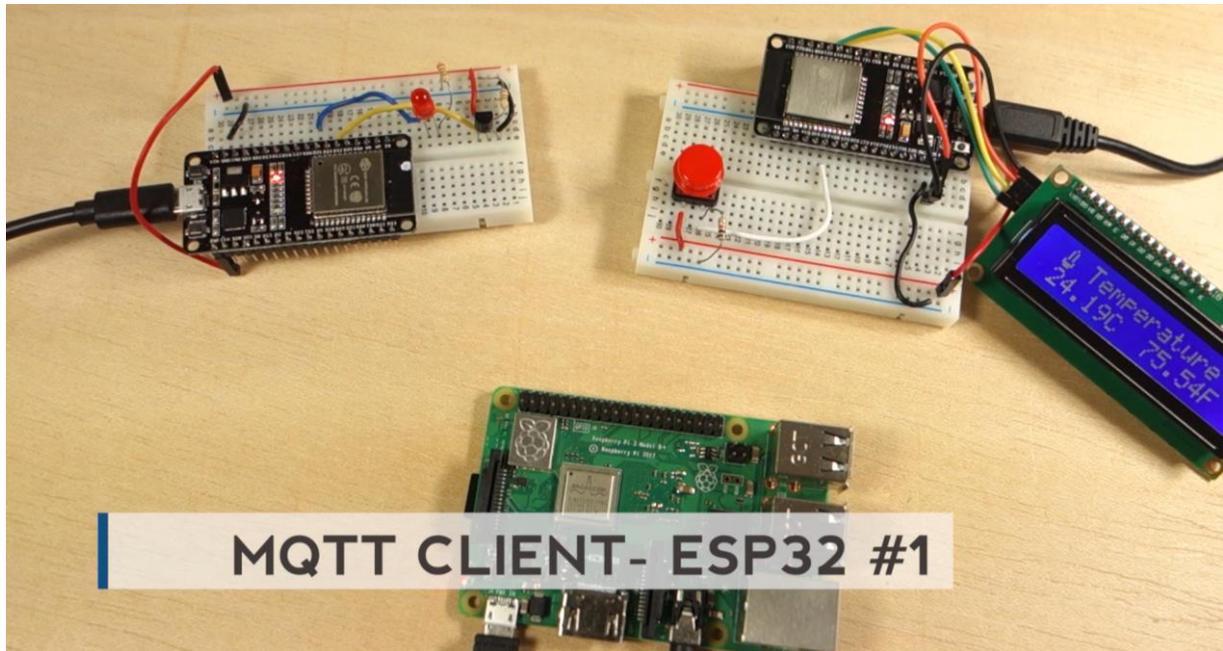
In our case, the Raspberry Pi IP address is **192.168.1.144**. Save your IP address. You'll need it in the next Units, so that the ESP32 is able to connect to the Mosquitto MQTT broker.

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 ties together with practical examples.

Unit 3 - MQTT Project: MQTT Client

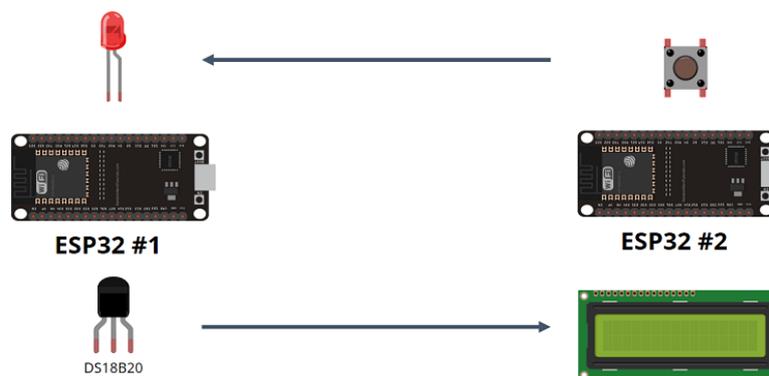
ESP32#1



Project Overview

In this Unit we're going to demonstrate 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.

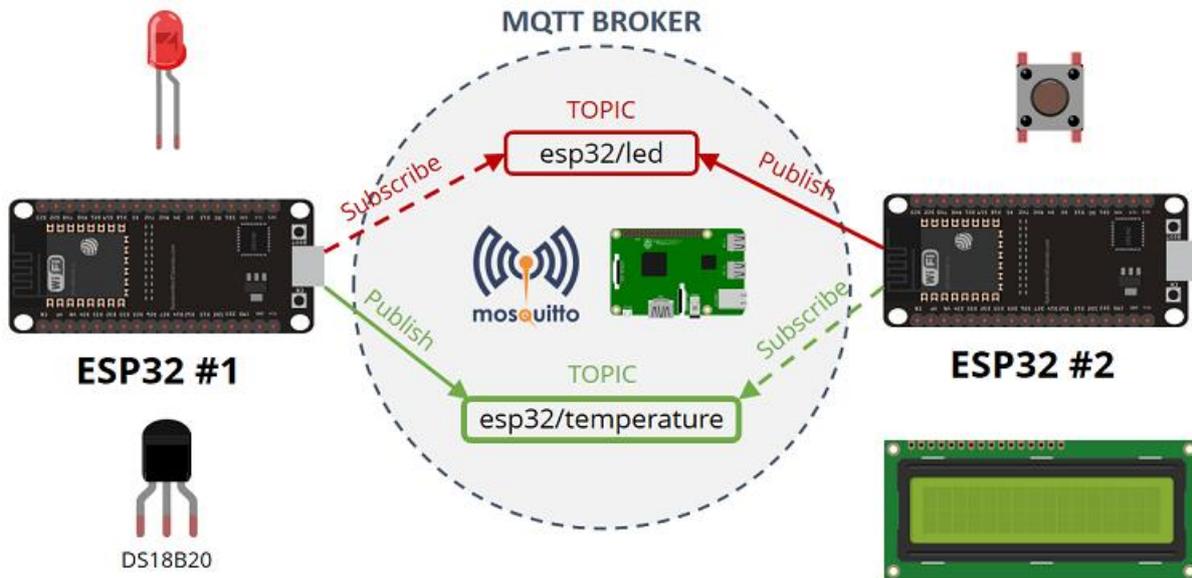
Here's a quick overview of the project. We'll have two ESP32 boards: ESP32 #1 and ESP32 #2:



- 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.

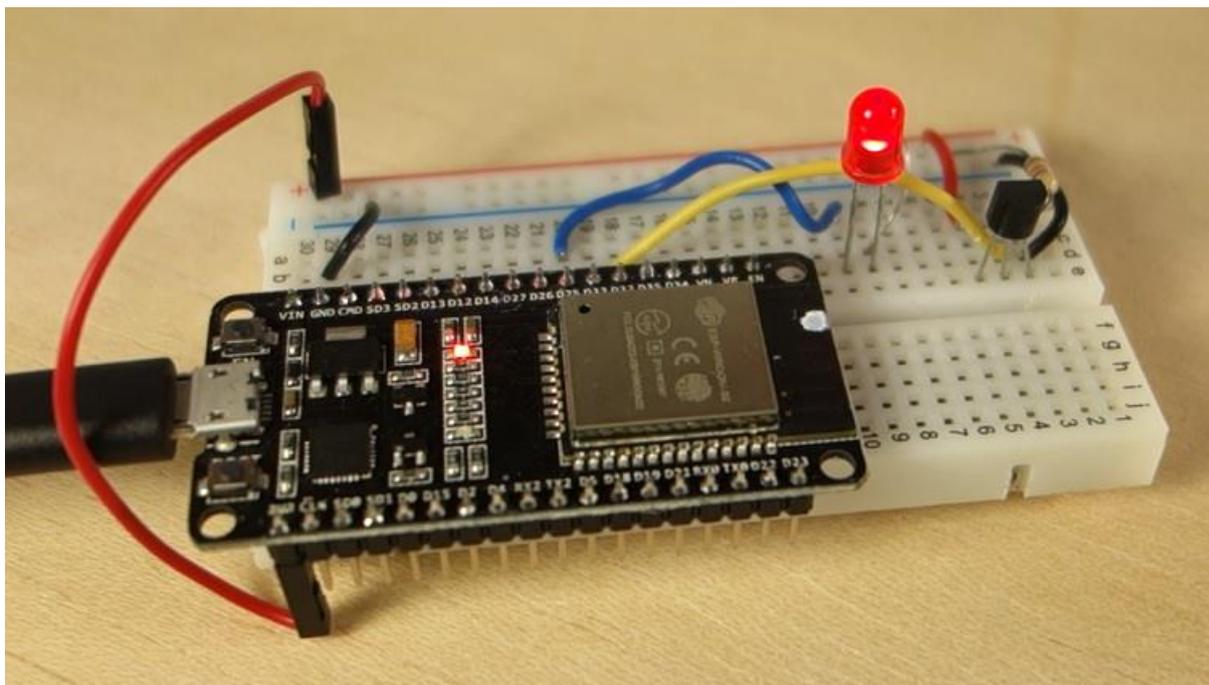
Here's the MQTT diagram of this setup.



- 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 prepare ESP32 #1 (in the next Unit you'll prepare ESP32 #2).



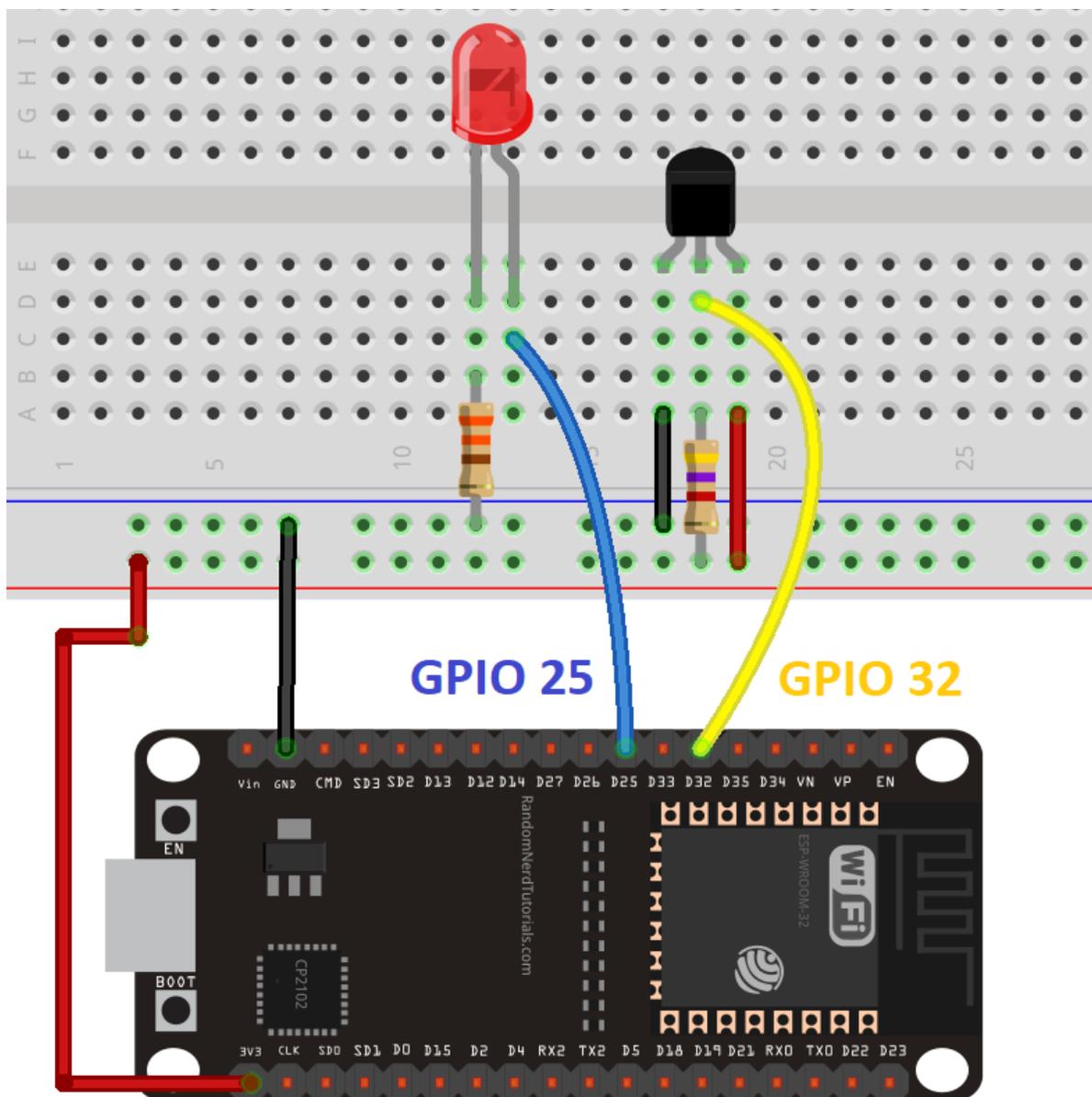
Schematic

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](#)

Start by assembling the circuit by following the next schematic diagram:

- Connect an LED with a 330 ohm resistor to GPIO 25;
- Wire the DS18B20 sensor with a 4.7K pull up resistor to GPIO 32.



(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.)

Installing Libraries

After that, follow the next steps to install the required libraries in your Arduino IDE.

Installing the Async TCP Library

To use MQTT with the ESP, you need the [Async TCP library](#).

1. [Click here to download the Async TCP client library](#). You should have a `.zip` folder in your **Downloads** folder
2. Unzip the `.zip` folder and you should get **AsyncTCP-master** folder
3. Rename your folder from **AsyncTCP-master** to **AsyncTCP**
4. Move the **AsyncTCP** folder to your Arduino IDE installation **libraries** folder
5. Finally, re-open your Arduino IDE

Installing the Async MQTT Client Library

To use MQTT with the ESP, you also need the [Async MQTT client library](#).

1. [Click here to download the Async MQTT client library](#). You should have a `.zip` folder in your **Downloads** folder
2. Unzip the `.zip` folder and you should get **async-mqtt-client-master** folder
3. Rename your folder from **async-mqtt-client-master** to **async_mqtt_client**
4. Move the **async_mqtt_client** folder to your Arduino IDE installation **libraries** folder
5. Finally, re-open your Arduino IDE

Installing the One Wire Library and Dallas Temperature Library

You need to install the [One Wire library by Paul Stoffregen](#) and the [Dallas Temperature library](#), so that you can use the DS18B20 sensor. Follow the next steps to install those libraries.

One Wire library

1. [Click here to download the One Wire library](#). You should have a `.zip` folder in your **Downloads** folder
2. Unzip the `.zip` folder and you should get **OneWire-master** folder
3. Rename your folder from **OneWire-master** to **OneWire**
4. Move the **OneWire** folder to your Arduino IDE installation **libraries** folder
5. Finally, re-open your Arduino IDE

Dallas Temperature library

1. [Click here to download the Dallas Temperature library](#). You should have a `.zip` folder in your **Downloads** folder
2. Unzip the `.zip` folder and you should get **Arduino-Temperature-Control-Library-master** folder

3. Rename your folder from **Arduino-Temperature-Control-Library-master** to **DallasTemperature**
4. Move the **DallasTemperature** folder to your Arduino IDE installation **libraries** folder
5. Finally, re-open your Arduino IDE

Code

With the libraries installed, open your Arduino IDE and copy the code provided.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/MQTT/ESP32_Client_1_LED_DS18B20/ESP32_Client_1_LED_DS18B20.ino

```

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

#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

// 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 SYSTEM_EVENT_STA_GOT_IP:
            Serial.println("WiFi connected");
            Serial.println("IP address: ");
            Serial.println(WiFi.localIP());
            connectToMqtt();
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            Serial.println("WiFi lost connection");
            xTimerStop(mqttReconnectTimer, 0); // ensure we don't reconnect
            to MQTT while reconnecting to Wi-Fi
            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);

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);
    }
}

```

If you want to make the code work straight away, just type your Wi-Fi SSID and password.

```

#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)

```

You can upload the code as it is and it will work. But we recommend continuing reading this Unit to learn how it actually 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
```

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 = "";
unsigned long previousMillis = 0;
const long interval = 5000;
```

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 objects to make it work.

```
// 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 Async Mqtt Client 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 SYSTEM_EVENT_STA_GOT_IP:
            Serial.println("WiFi connected");
            Serial.println("IP address: ");
            Serial.println(WiFi.localIP());
            connectToMqtt();
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            Serial.println("WiFi lost connection");
            xTimerStop(mqttReconnectTimer, 0); // ensure we don't reconnect
            to MQTT while reconnecting to Wi-Fi
            xTimerStart(wifiReconnectTimer, 0);
            break;
    }
}
```

Subscribe to MQTT topic

The *onMqttConnect()* function is responsible to subscribe 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.

```
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 in 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.

```

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 callbacks 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);
```

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 duplicate 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.

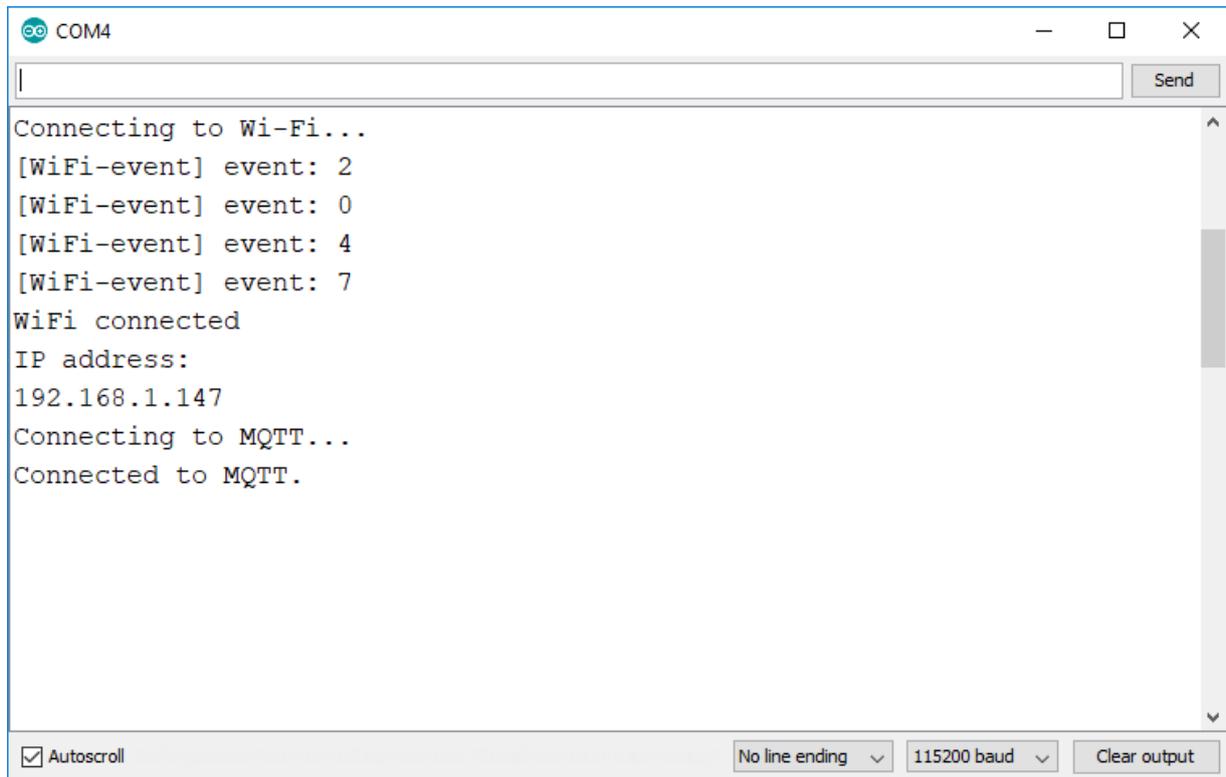
Uploading the code

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 being successfully connected to your router and MQTT broker.



```
COM4
Connecting to Wi-Fi...
[WiFi-event] event: 2
[WiFi-event] event: 0
[WiFi-event] event: 4
[WiFi-event] event: 7
WiFi connected
IP address:
192.168.1.147
Connecting to MQTT...
Connected to MQTT.
```

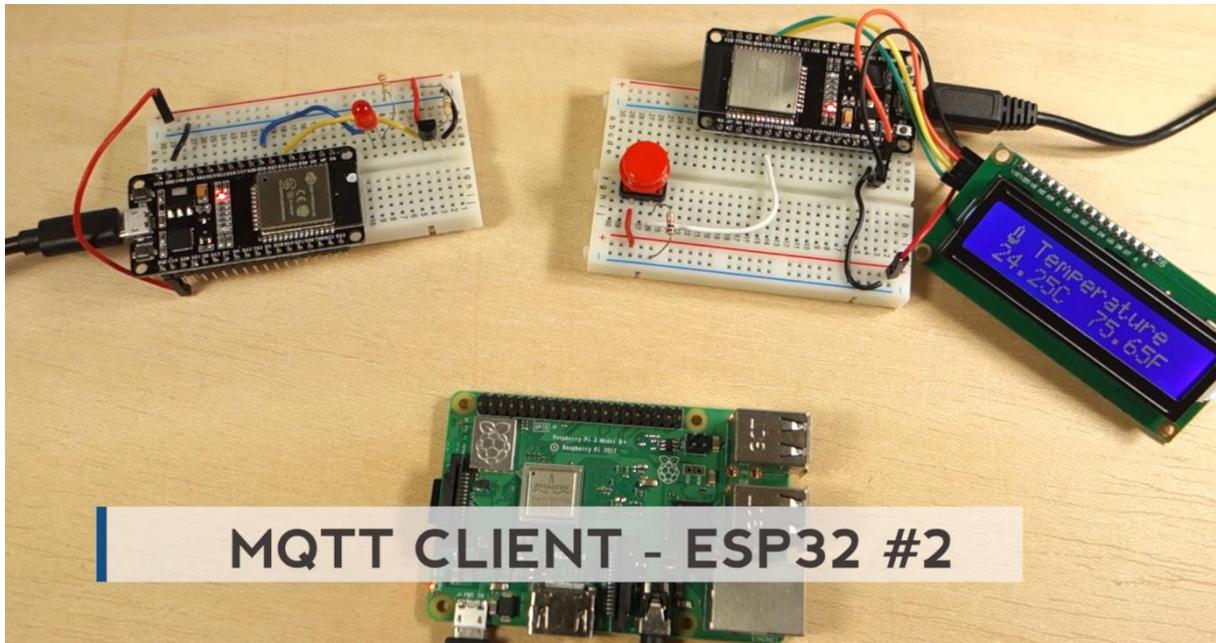
As you can see, it's working as expected.

Continue To The Next Unit...

Go to the next Unit to prepare ESP32 #2 and continue this project.

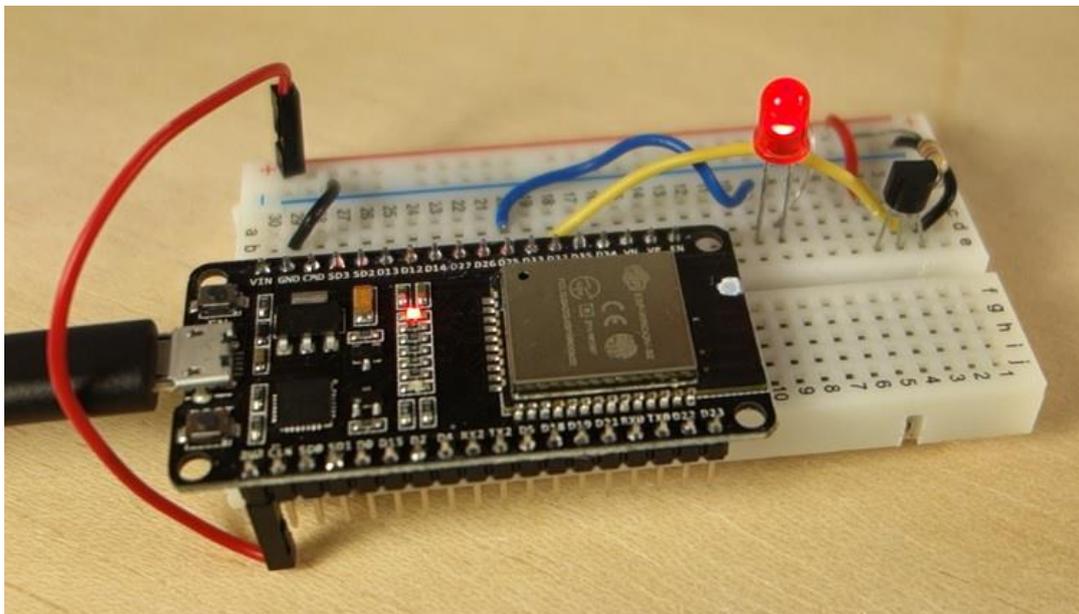
Unit 4 - MQTT Project : MQTT Client

ESP32 #2



Project Overview

This is part 2 of the ESP32 MQTT project example. If you've followed the previous Unit, your ESP32 #1 is ready.



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.

Schematic

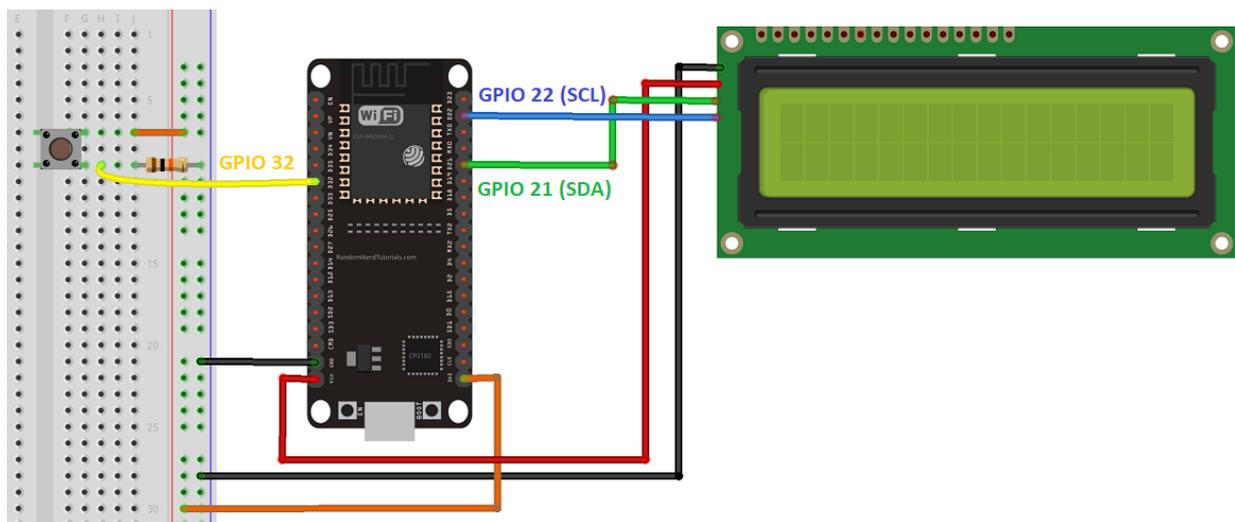
Here's a list of the parts you need to build the circuit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [Pushbutton](#)
- [10k Ohm resistor](#)
- [16×2 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 next schematic diagram as a reference.



(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.)

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. Unzip the .zip folder and you should get **LiquidCrystal_I2C-master** folder
3. Rename your folder from **LiquidCrystal_I2C-master** to **LiquidCrystal_I2C**
4. Move the **LiquidCrystal_I2C** folder to your Arduino IDE installation **libraries** folder
5. Finally, re-open your Arduino IDE

Note: to learn more on how to use the I2C LCD with the ESP32, you can read the following tutorial: [How to Use I2C LCD with ESP32 on Arduino IDE](#).

Code

After having both libraries installed, open your Arduino IDE and copy the code provided.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/MQTT/ESP32_Client_2_LCD_PUSHBUTTON/ESP32_Client_2_LCD_PUSHBUTTON.ino

```
/*  
  Rui Santos  
  Complete project details at https://randomnerdtutorials.com  
*/  
  
#include <WiFi.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(192, 168, 1, XXX)  
#define MQTT_PORT 1883  
  
// 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;
// current reading from the input pin (pushbutton)
int buttonState;
// previous reading from the input pin (pushbutton)
int lastButtonState = LOW;
// the last time the output pin was toggled
unsigned long lastDebounceTime = 0;
// the debounce time; increase if the output flickers
unsigned long debounceDelay = 50;

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 SYSTEM_EVENT_STA_GOT_IP:
      Serial.println("WiFi connected");
      Serial.println("IP address: ");
      Serial.println(WiFi.localIP());
      connectToMqtt();
      break;
    case SYSTEM_EVENT_STA_DISCONNECTED:
      Serial.println("WiFi lost connection");
      xTimerStop(mqttReconnectTimer, 0); // ensure we don't reconnect
to MQTT while reconnecting to Wi-Fi
      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) {
lcd.clear();
lcd.setCursor(1, 0);
lcd.write(0);
lcd.print(" Temperature");
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);

  // 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);

  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)P
    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;
}

```

Similarly to the previous example, if you type your SSID, password and MQTT broker IP address 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(192, 168, 1, XXX)

```

However, we recommend continuing reading this Unit to learn how it works.

How the Code Works

We'll skip most code sections, because they were already explained in the previous Unit.

Add your SSID, password and MQTT broker 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)

```

Then, 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;

```

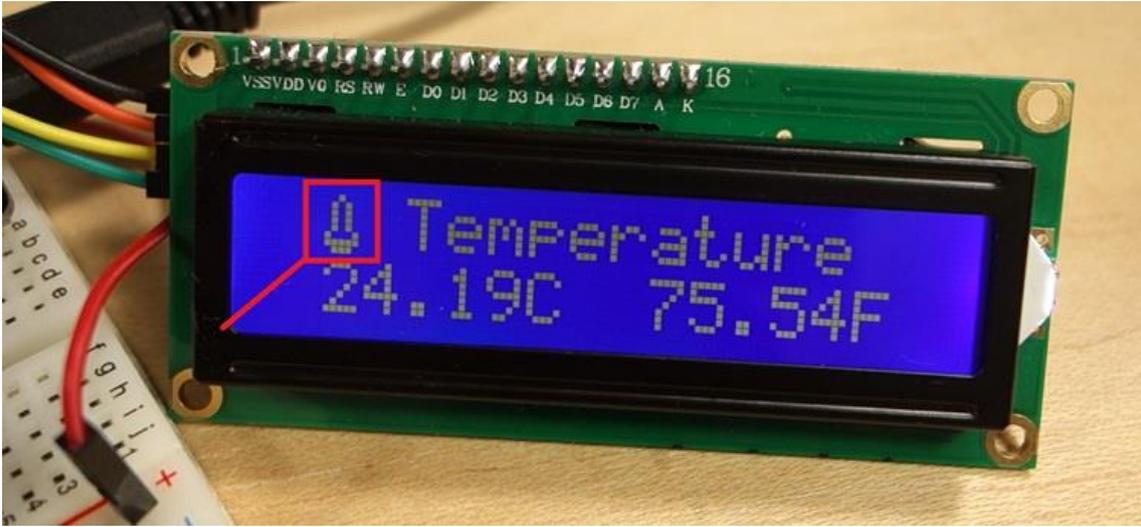
The following byte array is used to display a thermometer icon on the LCD.

```

// 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;  
// current reading from the input pin (pushbutton)  
int buttonState;  
// previous reading from the input pin (pushbutton)  
int lastButtonState = LOW;  
// the last time the output pin was toggled  
unsigned long lastDebounceTime = 0;  
// the debounce time; increase if the output flickers  
unsigned long debounceDelay = 50;
```

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.

```
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.

```

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) {
        lcd.clear();
        lcd.setCursor(1, 0);
        lcd.write(0);
        lcd.print(" Temperature");
        lcd.setCursor(0, 1);
        lcd.print(messageTemp);
    }
}
(...)

```

As explored in the previous Unit, this is where you implement what happens when you receive a message in a subscribed topic. You can create more *if* statements to check other topics, or to check the message content.

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);
```

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. We've used a similar example in a previous Unit (Module 2 - Unit 7).

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

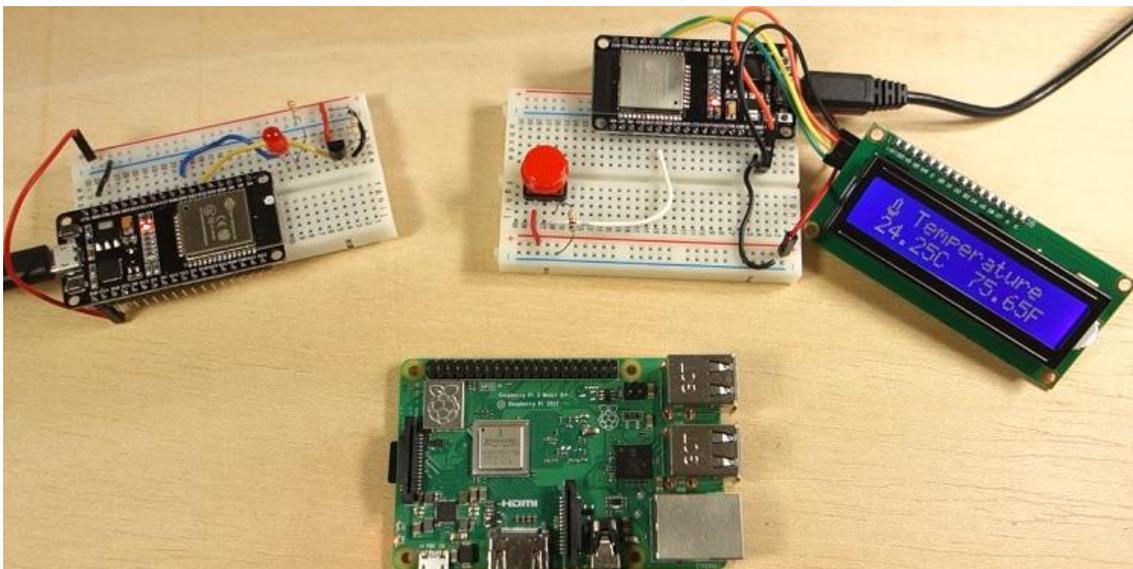
With your Raspberry Pi powered on and running the Mosquitto MQTT broker, open the serial monitor at the 115200 baud rate.



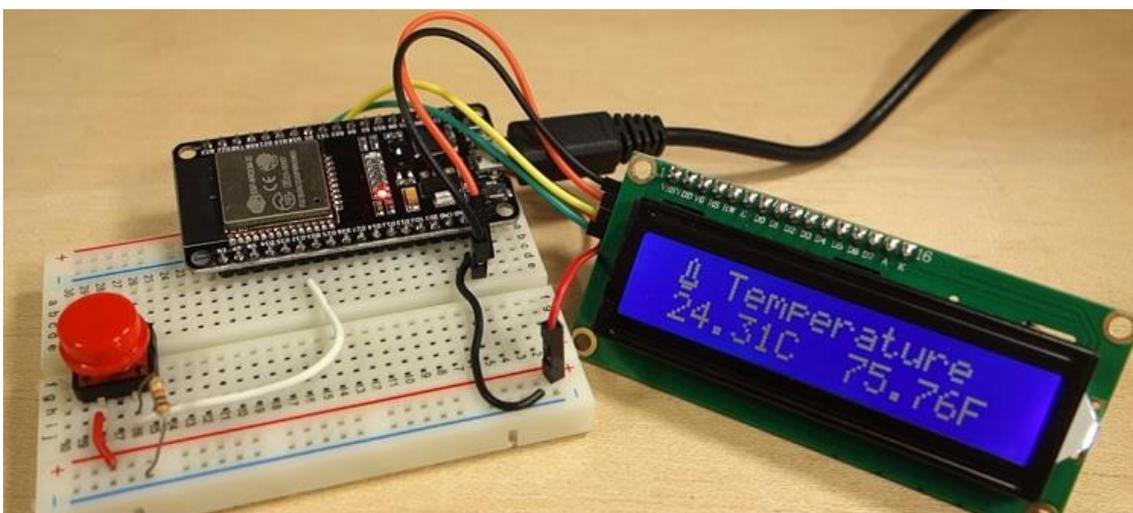
Check if your ESP32 is being successfully connected to your router and MQTT broker. As you can see in the following figure, it's working properly:

```
COM4
Connecting to Wi-Fi...
[WiFi-event] event: 2
[WiFi-event] event: 0
[WiFi-event] event: 4
[WiFi-event] event: 7
WiFi connected
IP address:
192.168.1.148
Connecting to MQTT...
Connected to MQTT.
```

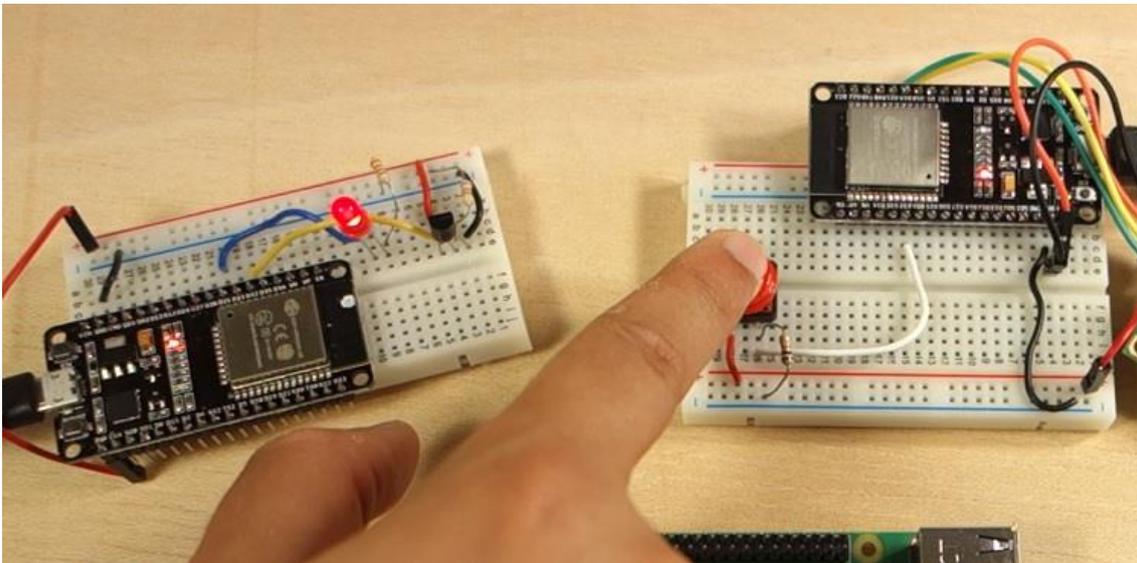
Now, power both ESP32 boards and leave the mosquitto MQTT broker running.



The ESP32 #2 instantly receives the temperature readings from ESP32 #1 in both Celsius and Fahrenheit degrees.



If you press the pushbutton, it instantly toggles the LED attached to the ESP32 #1.



This system can easily be expanded 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.

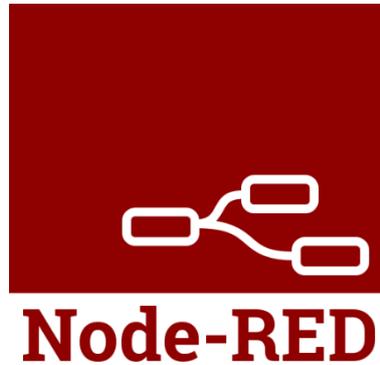


Wrapping Up

In this Unit (and previous one) we've shown you how to exchange data between two ESP32 boards using MQTT. In the next Units, you'll learn how to control the ESP32 through Node-RED and how to send data from the ESP32 to the Node-RED dashboard using MQTT.

Unit 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 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 how to connect the ESP32 to Node-RED using MQTT communication protocol. Many of our readers use Node-RED in their home automation projects and are interested in interfacing the ESP32 with Node-RED. These Units are mainly addressed for them.

Prerequisites

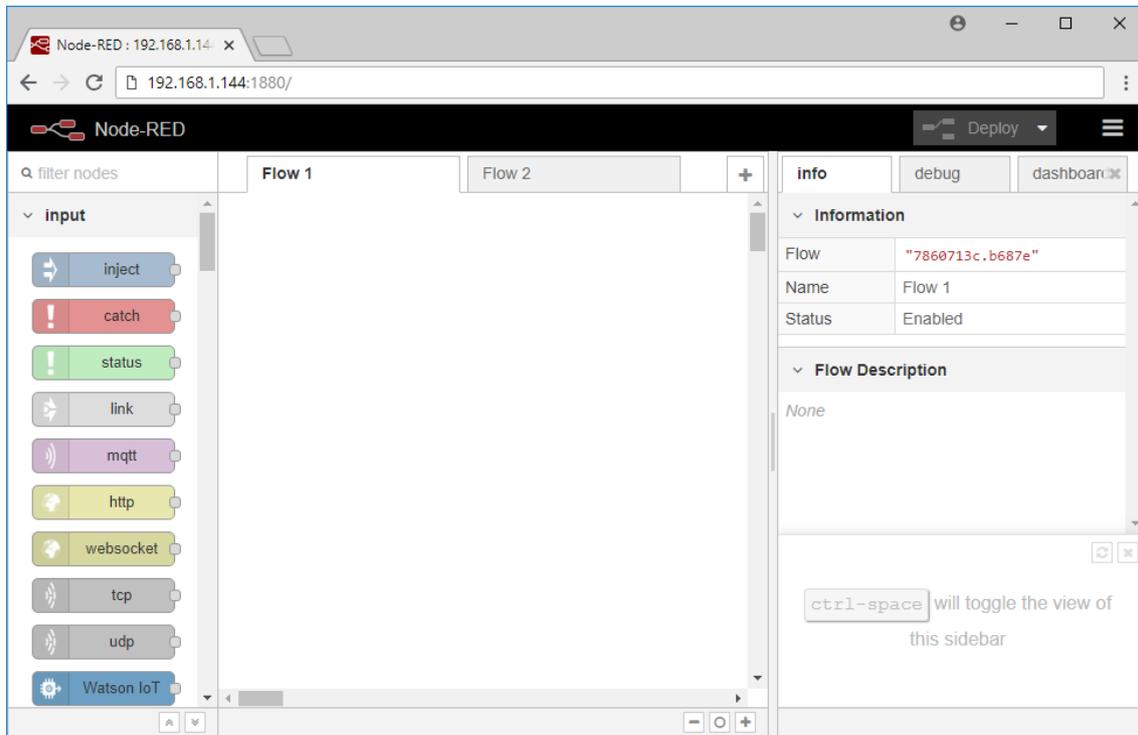
Before continuing:

- You should be familiar with the Raspberry Pi board – [read Getting Started with Raspberry Pi](#);
- You should have the Raspbian or Raspbian Lite operating system 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 \(5V 2.5A\)](#);
- You also need [Mosquitto MQTT Broker installed](#).

What's Node-RED?

[Node-RED](#) is a powerful open source visual wiring tool for building Internet of Things (IoT) applications.

Node-RED uses visual programming that allows you to connect code blocks, known as nodes, together to perform a task, simplifying much of the programming significantly. The nodes when wired together are called flows.



Why Node-RED is a great solution?

Node-RED is open source and developed by IBM and runs perfectly with the Raspberry Pi.

Node-RED allows you to prototype a complex home automation system quickly, giving you more time to spend on designing and making cool stuff.

What can you do with Node-RED?

Node-RED makes it easy to:

- Access your Raspberry Pi GPIOs
- Establish an MQTT connection with other boards (ESP32, ESP8266, Arduino, etc)
- Create a responsive graphical user interface for your projects with Node-RED Dashboard
- Communicate with third-party services (IFTTT.com, Adafruit.io, Thing Speak, etc)
- Retrieve data from the web (weather forecast, stock prices, emails. etc)
- Create time triggered events
- Store and retrieve data from a database

Here's a library with some [examples of flows](#) and nodes for Node-RED.

Installing Node-RED

Getting Node-RED installed in your Raspberry Pi is quick and easy. It just takes a few commands.

Having a Terminal window of your Raspberry Pi open, enter the next command to install Node-RED:

```
pi@raspberrypi:~ $ bash <(curl -sL https://raw.githubusercontent.com/node-red/raspbian-deb-package/master/resources/update-nodejs-and-nodered)
```

The installation should be completed after a couple of minutes.

Autostart Node-RED on boot

To automatically run Node-RED when the Pi boots up, you need to enter the following command:

```
pi@raspberrypi:~ $ sudo systemctl enable nodered.service
```

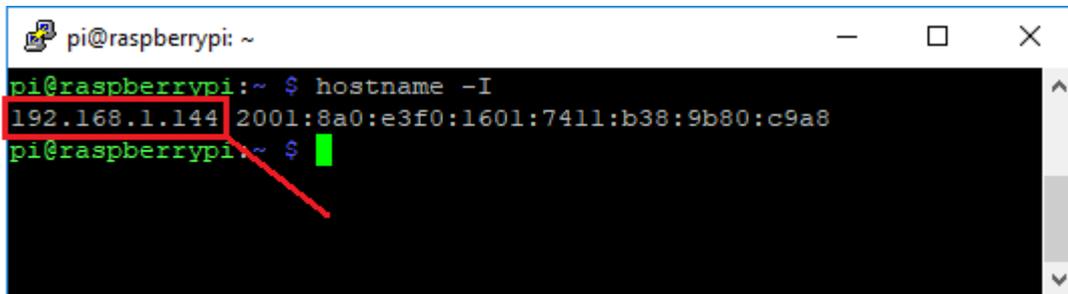
Now, restart your Raspberry Pi, so the auto-start takes effect:

```
pi@raspberrypi:~ $ sudo reboot
```

Raspberry Pi IP Address

To retrieve your Raspberry Pi IP address, type the next command in your Terminal window:

```
pi@raspberrypi:~ $ hostname -I
```



In our case, the Raspberry Pi IP address is **192.168.1.144** (save it, because you'll need it in the next Unit).

Testing the Installation

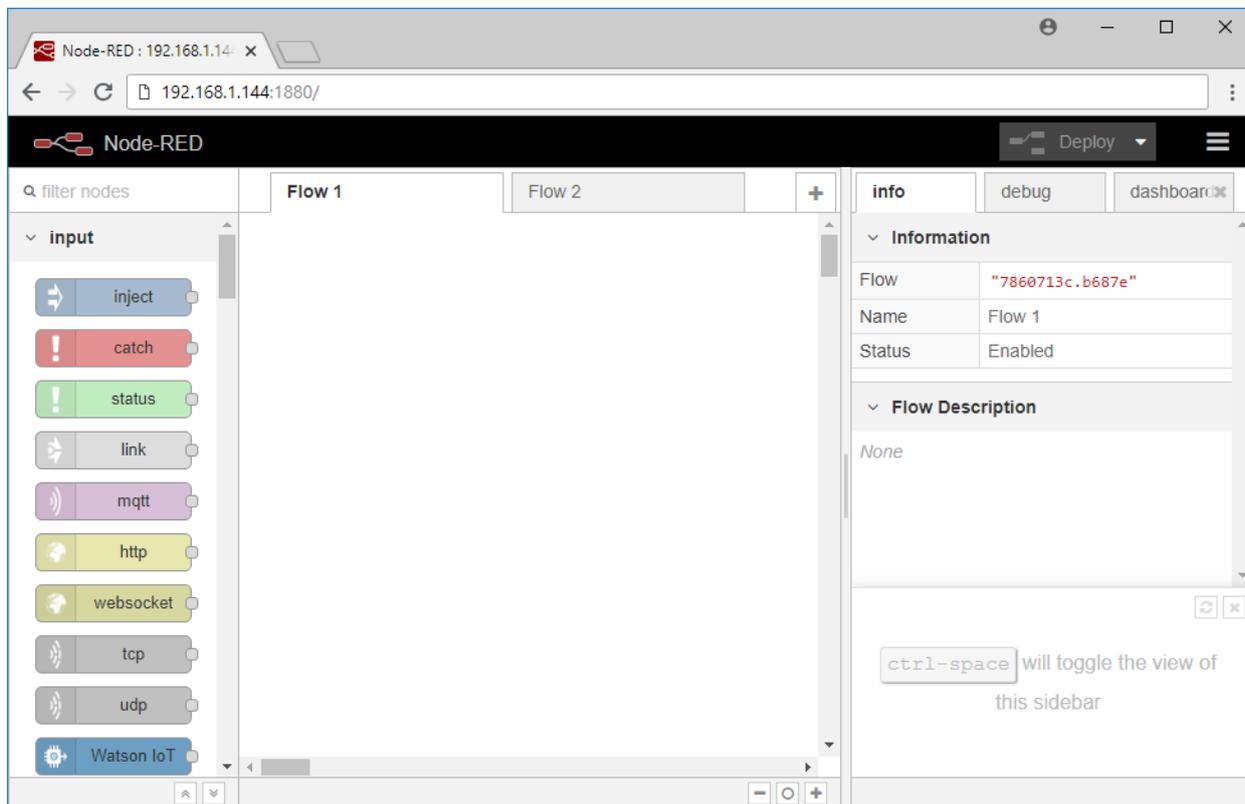
When your Pi is back on, you can test the installation by entering the IP address of your Pi in a web browser followed by the 1880 port number:

```
http://YOUR_RPi_IP_ADDRESS:1880
```

In my case is:

```
http://192.168.1.144:1880
```

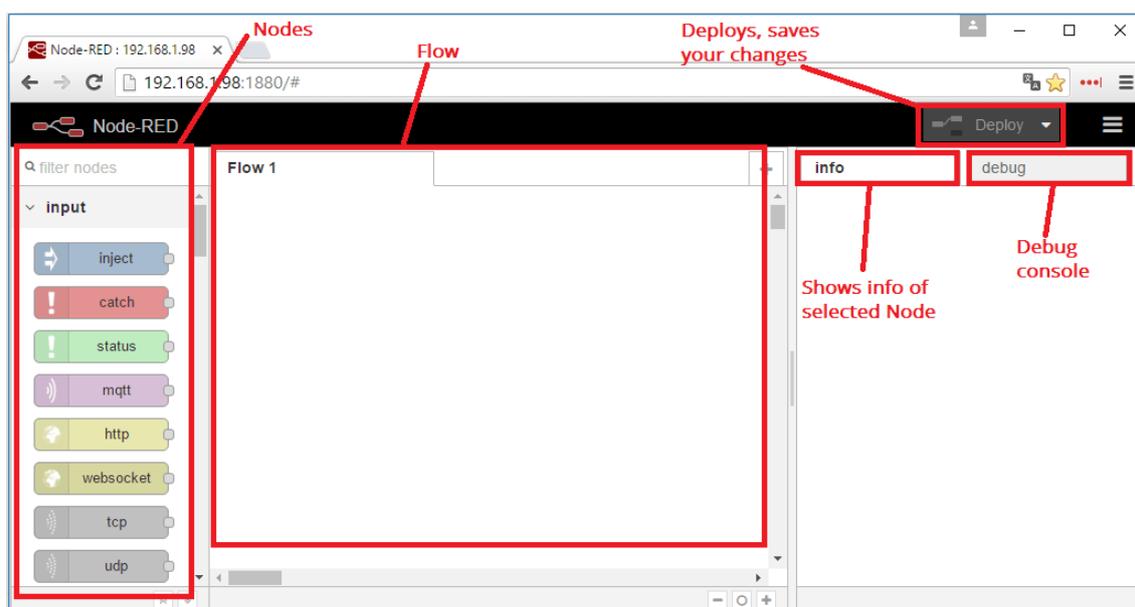
A page as shown in the figure below should load:



Node-RED Overview

On the left-side, 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 info tab.

In the center, you have the Flow and this is where you place the nodes.



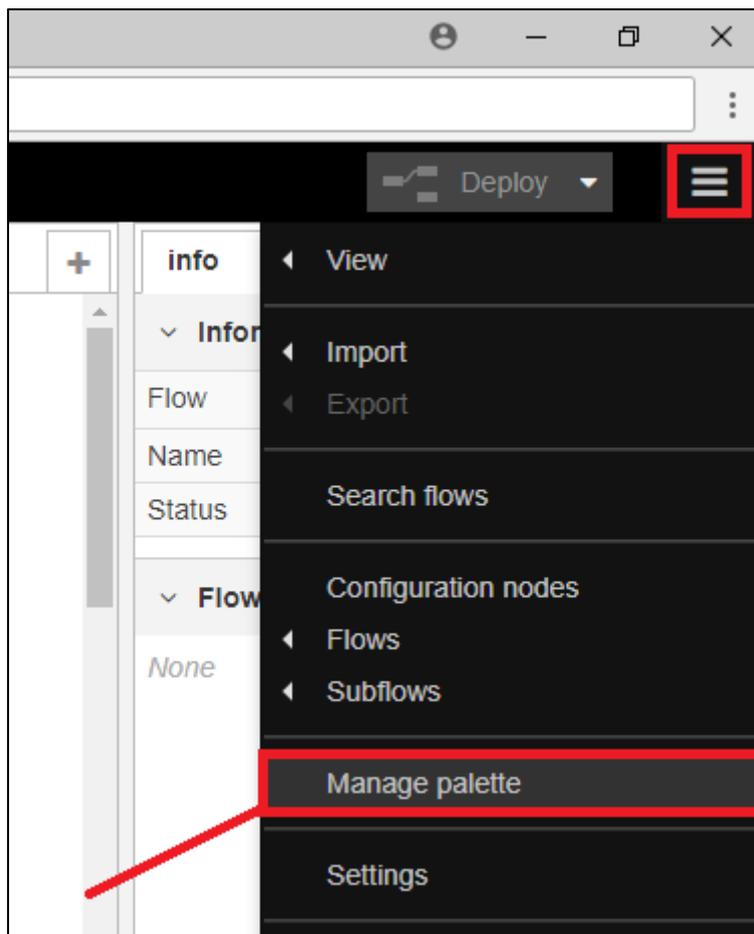
Installing Node-RED Dashboard

Node-RED Dashboard is a set of nodes that provides tools to create a user interface with buttons, charts, text, and much more.

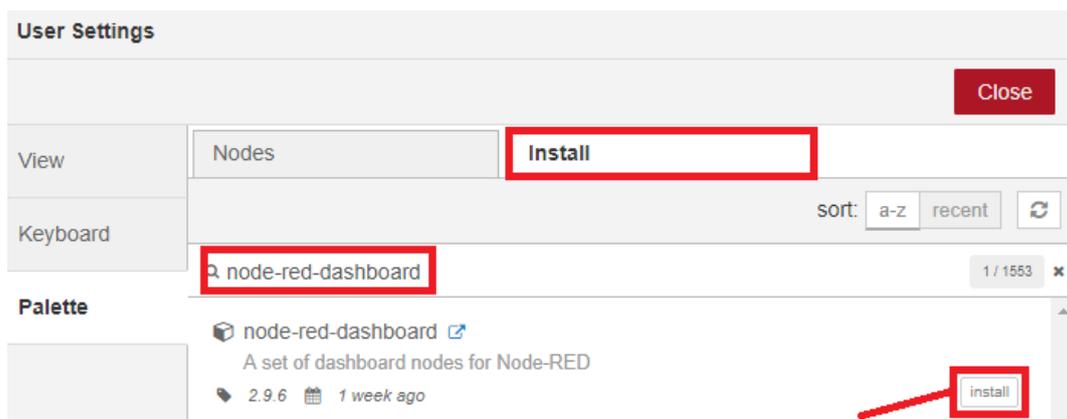
To learn more about Node-RED Dashboard you can check the following links:

- Node-RED site: <http://flows.nodered.org/node/node-red-dashboard>
- GitHub: <https://github.com/node-red/node-red-dashboard>

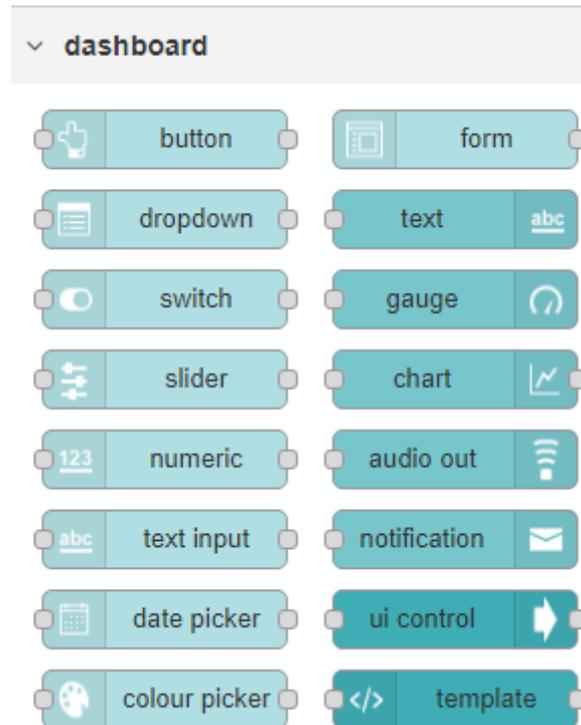
To install the Node-RED Dashboard, go to the “Settings” menu in the top-right corner and open the “Manage palette” menu:



A new window opens with the User Settings. Open the “Install” tab, search for “node-red-dashboard”, and press the “Install” button:



Close the menu and a new set of nodes should appear in your left window with the dashboard nodes:



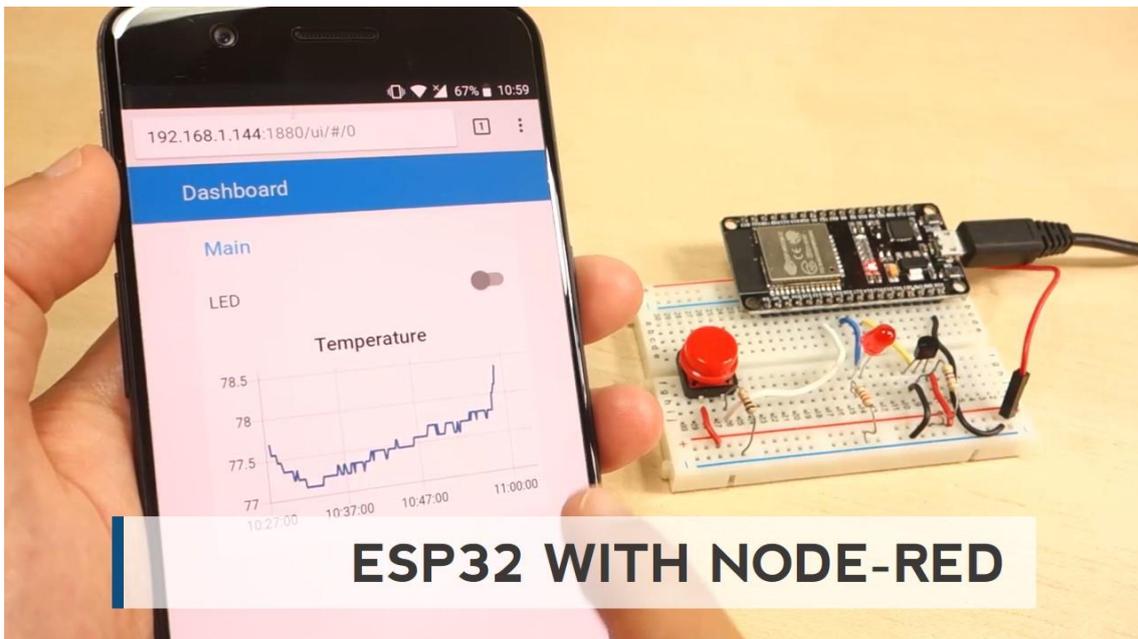
Continue To The Next Unit ...

Now, you have everything ready to integrate Node-RED with your ESP32 and create a user interface. Continue to the next Unit to learn how to create a Node-RED user interface to interact with the ESP32 board.

Unit 6 - Connect ESP32 to Node-RED

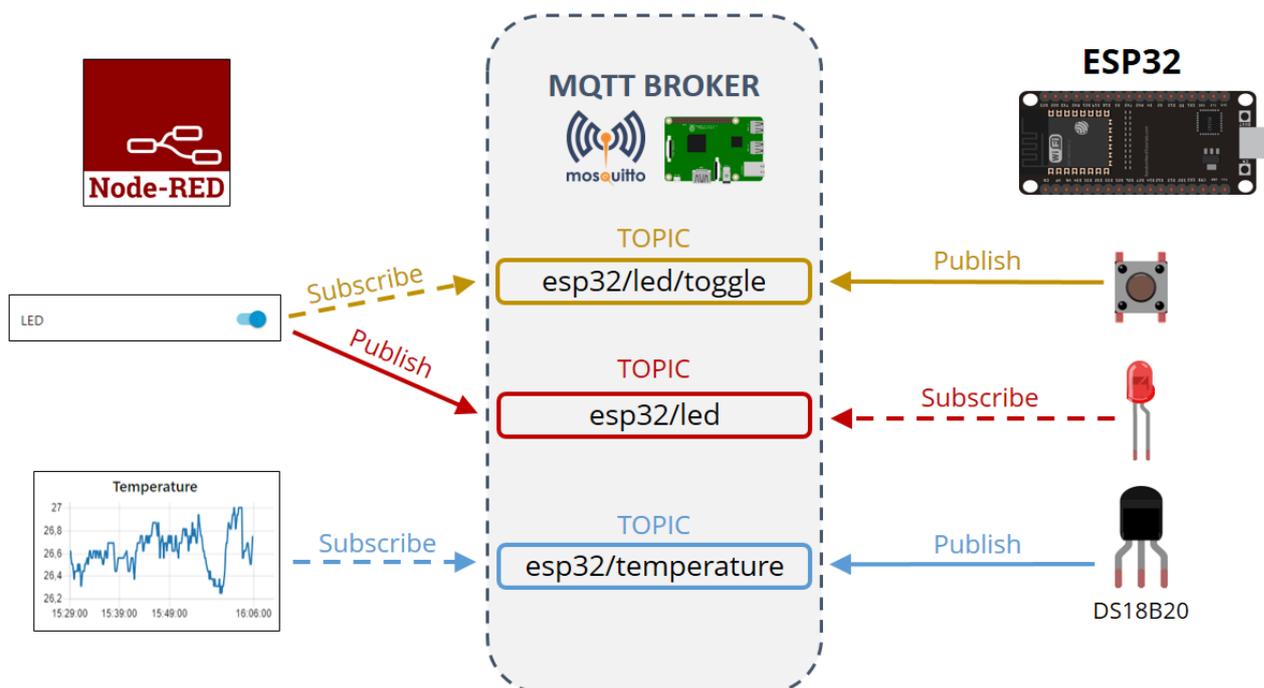
using MQTT

In this Unit we're going to demonstrate how to use Node-RED Dashboard to control the ESP32 GPIOs and display temperature readings in a chart. We're going to 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:

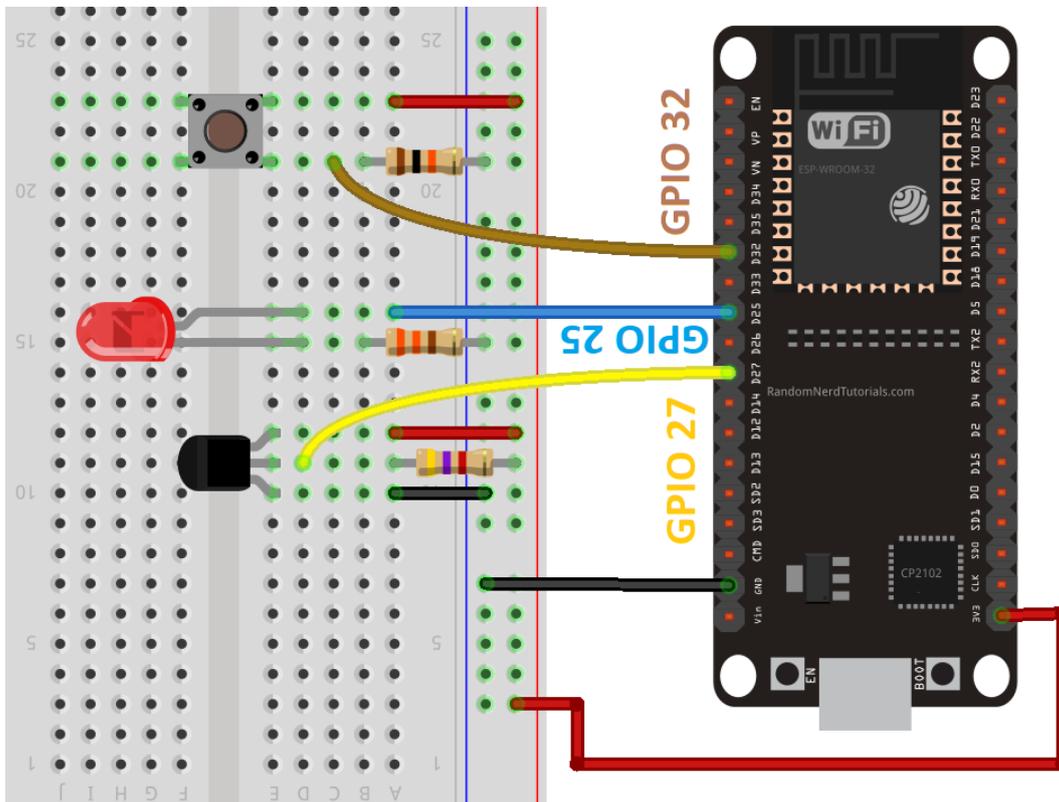
- [Async TCP](#)
- [Async MQTT Client](#)
- [Dallas Temperature](#)
- [One Wire](#)

Schematic

To build the circuit for this project you need the following parts:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [5mm LED](#)
- [330 Ohm resistor](#)
- [DS18B20 temperature sensor](#)
- [4.7k Ohm resistor](#)
- [Pushbutton](#)
- [10k Ohm resistor](#)
- [Jumper wires](#)
- [Breadboard](#)

Wire the circuit 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.)

Uploading the code

After installing the required libraries, copy the following code to your Arduino IDE.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_NodeRED/ESP32_NodeRED.ino

```

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

#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, X)
#define MQTT_PORT 1883

```

```

// 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 SYSTEM_EVENT_STA_GOT_IP:
            Serial.println("WiFi connected");
            Serial.println("IP address: ");
            Serial.println(WiFi.localIP());
            connectToMqtt();
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            Serial.println("WiFi lost connection");
            xTimerStop(mqttReconnectTimer, 0); // ensure we don't reconnect to MQTT
while reconnecting to Wi-Fi
            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);

    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 and MQTT broker IP address 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(192, 168, 1, X)

```

We won't explore how this code works. The parts related with 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 Celsius degrees
// uint16_t packetIdPub2 = mqttClient.publish("esp32/temperature", 2, true,
// String(sensors.getTempCByIndex(0)).c_str());

```

Then, uncomment the next 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());

```

Creating the Node-RED flow

Before creating the flow, you need to have installed in your Raspberry Pi:

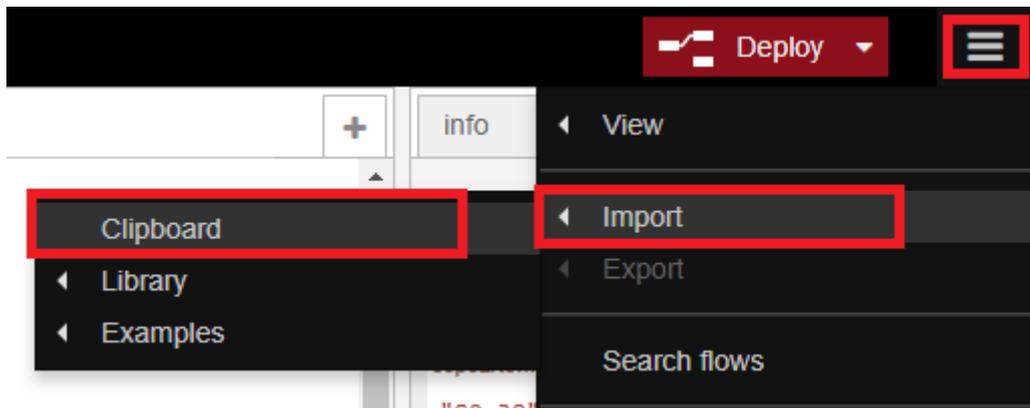
- Node-RED
- Node-RED Dashboard
- Mosquitto MQTT Broker

Importing the flow

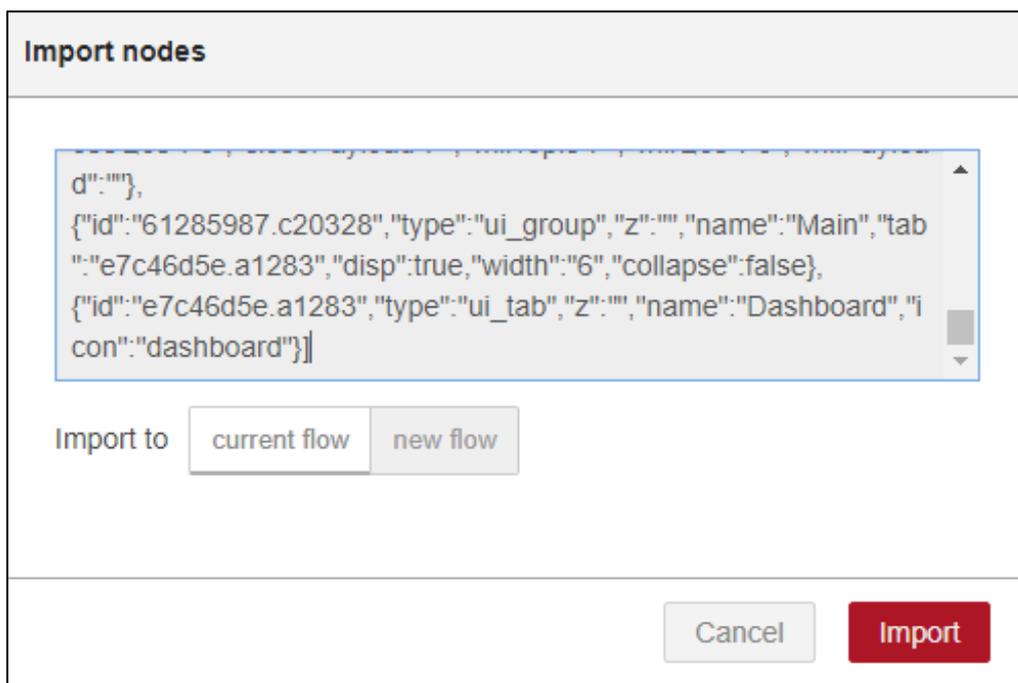
After that, import the Node-RED flow provided. Go to the [GitHub repository to see the raw file](#), and copy the flow provided.

```
2 lines (1 sloc) | 2.32 KB
1 [{"id":"9e58624.7faaba","type":"mqtt out","z":"c02b79b2.501998","name":"","topic":"esp32/led","qos":"","retain":"","broker":"10e78a89.5b4fd"}]
```

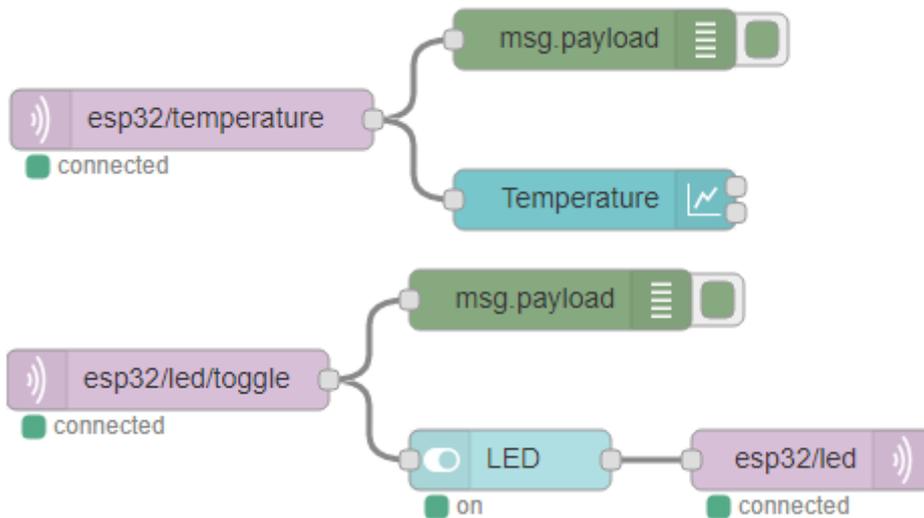
Next, in the Node-RED window, at the top right corner, select the menu, and go to **Import** ▶ **Clipboard**.



Then, paste the flow provided and click the **Import** button.



The following nodes should load in your flow:



After making any changes to your flow, click the Deploy button to save all the changes.



Understanding the Flow

Let's take a look at each node to better 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.

Edit mqtt in node

Delete Cancel Done

node properties

Server localhost:1883

Topic esp32/temperature

QoS 2

Name Name

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.

Edit chart node

Delete Cancel Done

▼ node properties

Group Main [Dashboard]

Size auto

Label Temperature

Type Line chart enlarge points

X-axis last 1 hours OR 1000 points

X-axis Label ▼ HH:mm:ss

Y-axis min max

Legend None Interpolate linear

Series Colours

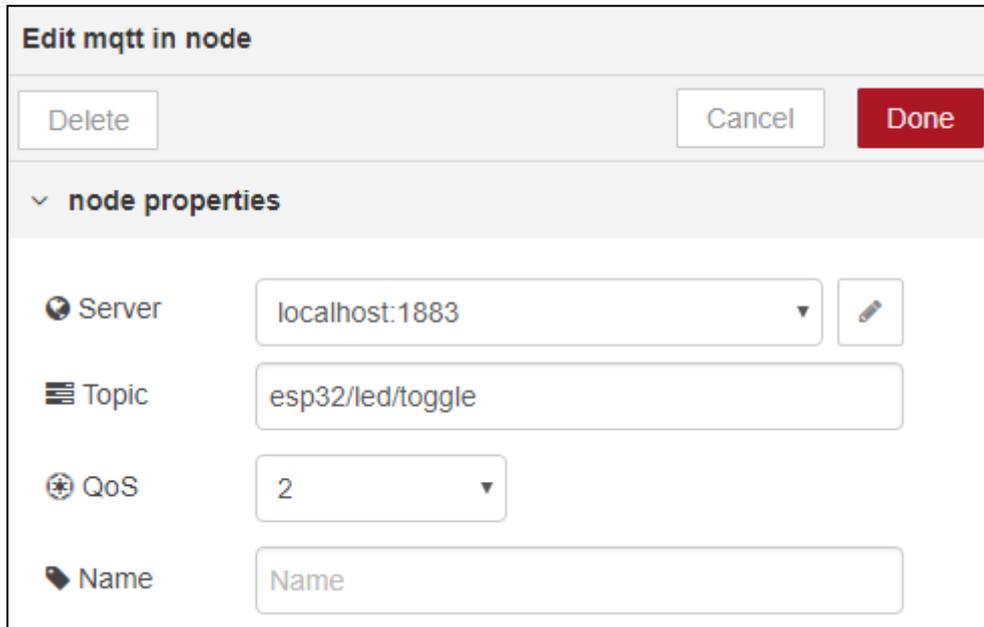
Blank label display this text before valid data arrives

Use deprecated (pre 2.5.0) data format.

Name

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.



The screenshot shows a dialog box titled "Edit mqtt in node". At the top, there are three buttons: "Delete", "Cancel", and "Done". Below the buttons is a section titled "node properties" with a dropdown arrow. Underneath, there are four rows of configuration options:

- Server:** A dropdown menu showing "localhost:1883" and a pencil icon for editing.
- Topic:** A text input field containing "esp32/led/toggle".
- QoS:** A dropdown menu showing "2".
- Name:** A text input field containing "Name".

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

▼ **node properties**

Group Main [Dashboard]

Size auto

Label LED

Icon Default

→ If msg arrives on input, pass through to output:

When clicked, send:

On Payload on

Off Payload off

Topic

Name

MQTT out node

After this, we have an MQTT publish Node that will publish the message from the switch on the **esp32/led** topic.

Edit mqtt out node

Delete
Cancel
Done

▼ **node properties**

Server localhost:1883

Topic esp32/led

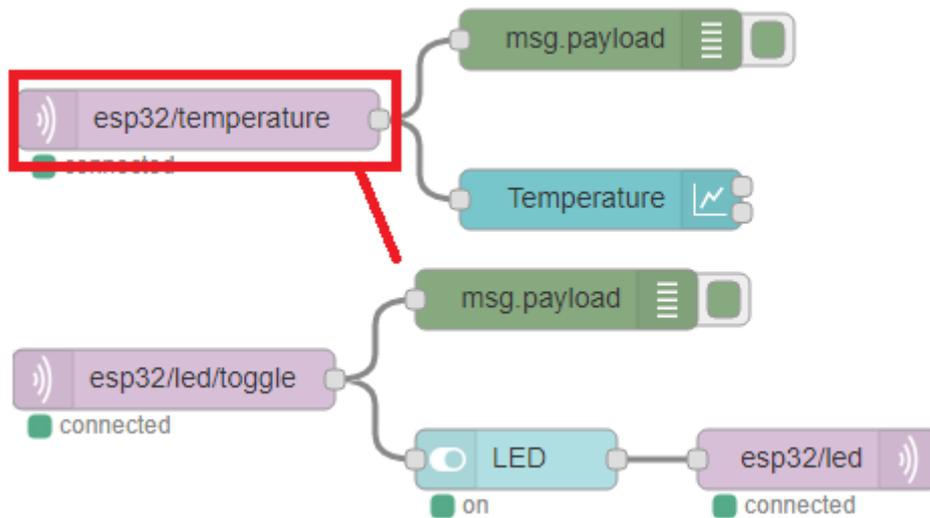
QoS Retain

Name Name

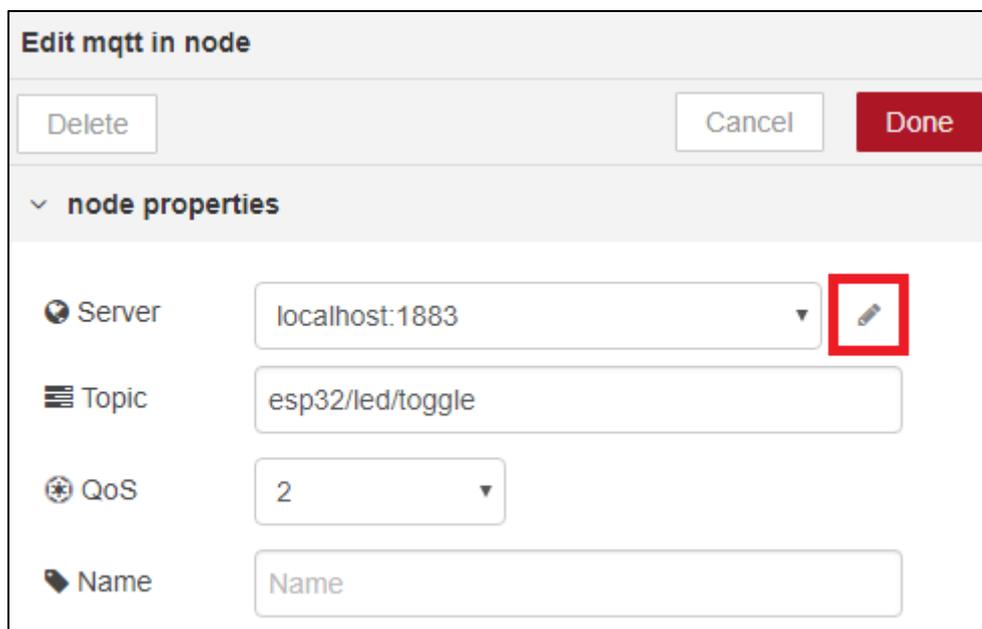
The sequence and logic of these nodes is easier to understand if you also take a look at the MQTT diagram at the beginning of this Unit.

MQTT broker settings

We're assuming that Node-RED and MQTT broker are installed in 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:



The image shows the 'Edit mqtt in node' dialog box. It has a 'Delete' button on the left, and 'Cancel' and 'Done' buttons on the right. Under the 'node properties' section, there are four fields: 'Server' with a dropdown menu showing 'localhost:1883' and a red box around the edit icon; 'Topic' with a text input field containing 'esp32/led/toggle'; 'QoS' with a dropdown menu showing '2'; and 'Name' with a text input field containing 'Name'.

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 Port

Enable secure (SSL/TLS) connection

Client ID

Keep alive time (s) Use clean session

Use legacy MQTT 3.1 support

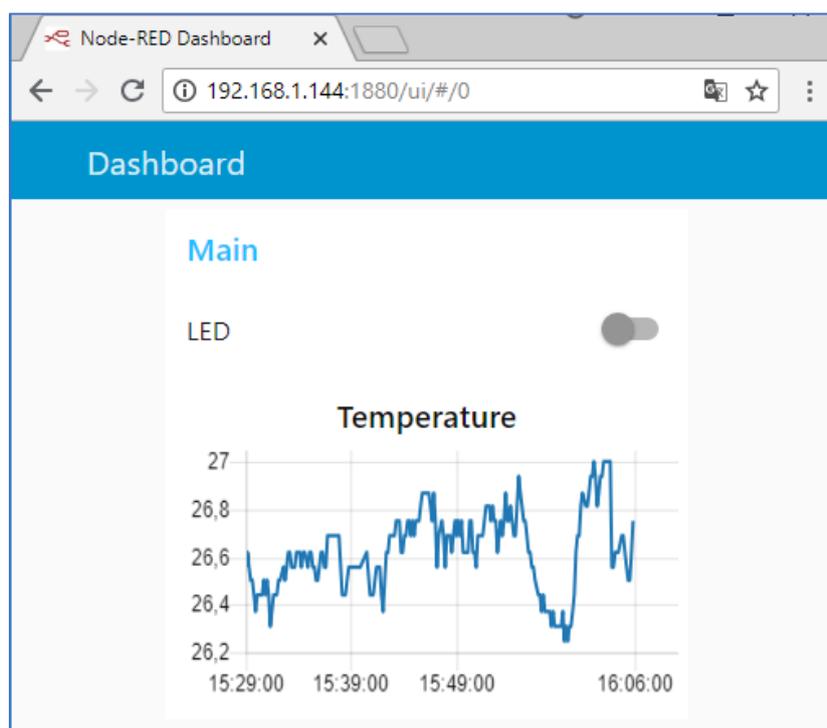
Note: since the Raspberry Pi is running the Mosquitto broker locally, we can leave the Server “localhost” and Port “1883”. You might need to change those settings and update them in your specific use case, so Node-RED can establish an MQTT connection with your broker.

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:

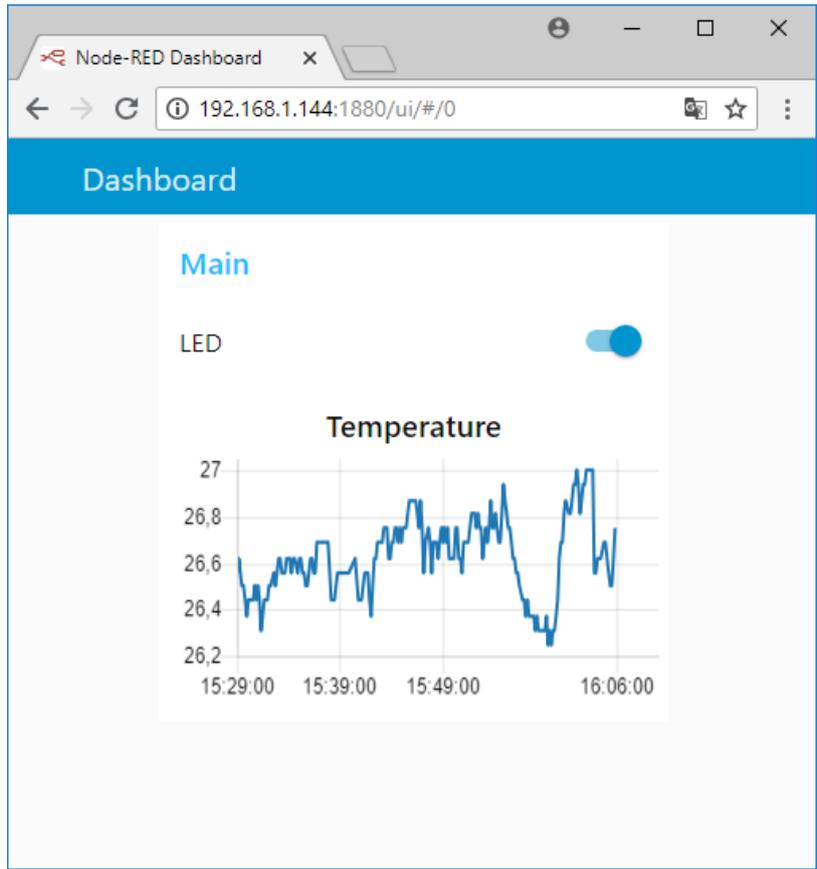
http://Your_RPi_IP_address:1880/ui

Your application should look as shown in the following figure.

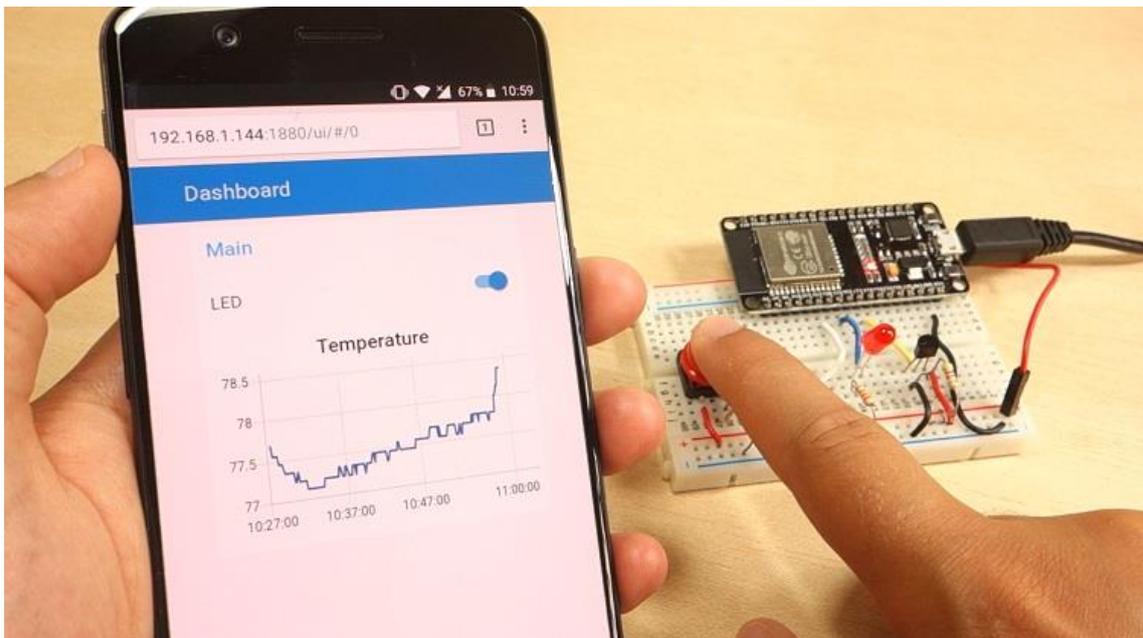


Demonstration

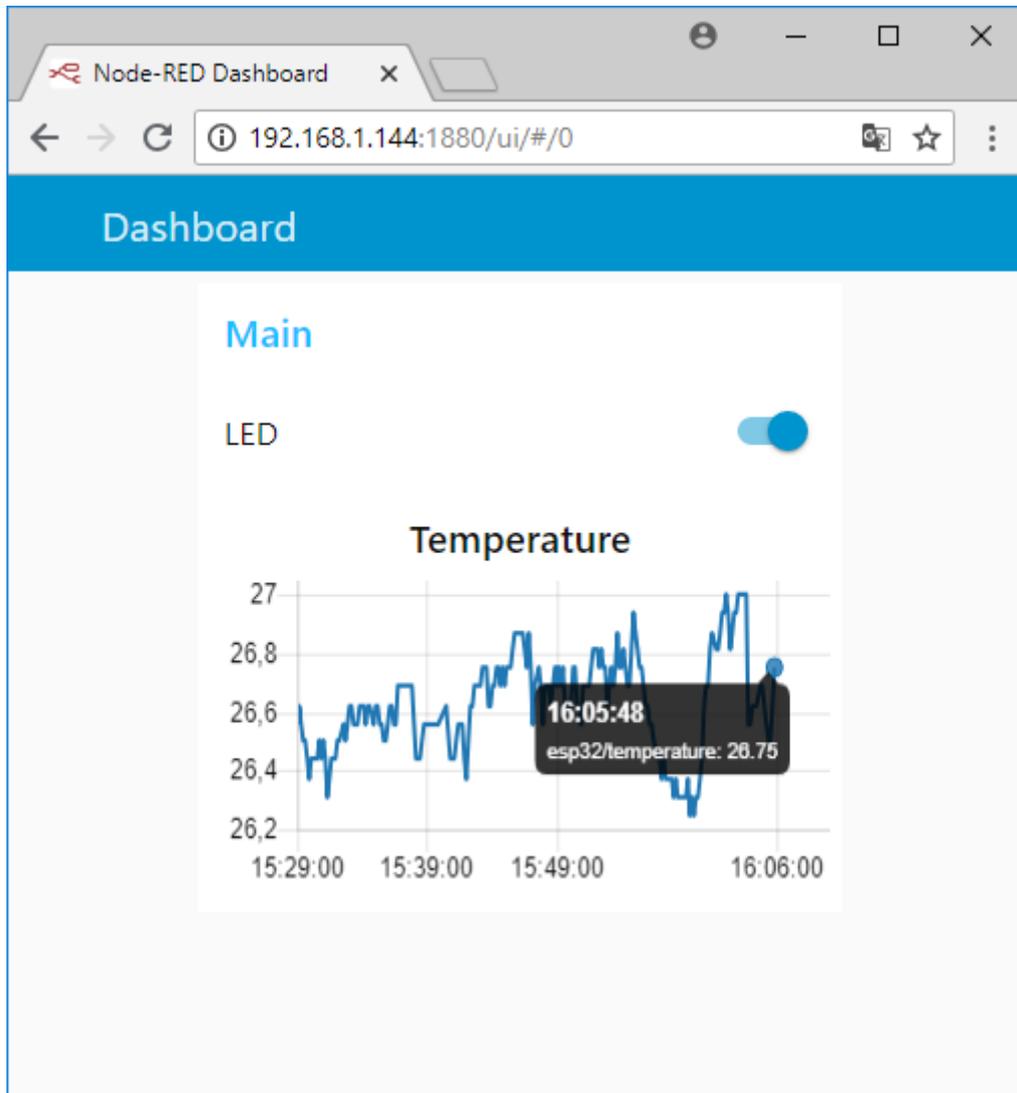
You can turn the LED on or off by pressing the switch.



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 how to 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.



MODULE 8

ESP-NOW Communication Protocol

Unit 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."*



This means that after pairing a device with each other, the connection is persistent. In other words, if suddenly one of your boards loses power or resets, when it restarts, it will automatically connect to its peer to continue the communication.

ESP-NOW supports the following features:

- Encrypted and unencrypted unicast communication;
- Mixed encrypted and unencrypted peer devices;
- Up to 250-byte payload can be carried;

- Sending callback function that can be set to inform the application layer of transmission success or failure.
- ESP-NOW technology also has the following limitations:
- Limited encrypted peers. 10 encrypted peers at the most are supported in Station mode; 6 at the most in SoftAP or SoftAP + Station mode;
- Multiple unencrypted peers are supported, however, their total number should be less than 20, including encrypted peers;
- **Payload is limited to 250 bytes.**

In simple words, ESP-NOW is a fast communication protocol that can be used to exchange small messages (up to 250 bytes) between ESP32 boards.

ESP-NOW is very versatile and you can have one-way or two-way communication in different setups.

ESP-NOW One-Way Communication

For example, in one-way communication, you can have scenarios like this:

- **One ESP32 board sending data to another ESP32 board**

This configuration is very easy to implement and it is great to send data from one board to the other like sensor readings or ON and OFF commands to control GPIOs.



- **A “master” ESP32 sending data to multiple ESP32 “slaves”**

One ESP32 board sending the same or different commands to different ESP32 boards. This configuration is ideal to build something like a remote control. You can have several ESP32 boards around the house that are controlled by one main ESP32 board.



- One ESP32 "slave" receiving data from multiple "masters"

This configuration is ideal if you want to collect data from several sensors nodes into one ESP32 board. This can be configured as a web server to display data from all the other boards, for example.



Note: in the ESP-NOW documentation there isn't such thing as "sender/master" and "receiver/slave". Every board can be a sender or receiver. However, to keep things clear we'll use the terms "sender" and "receiver" or "master" and "slave".

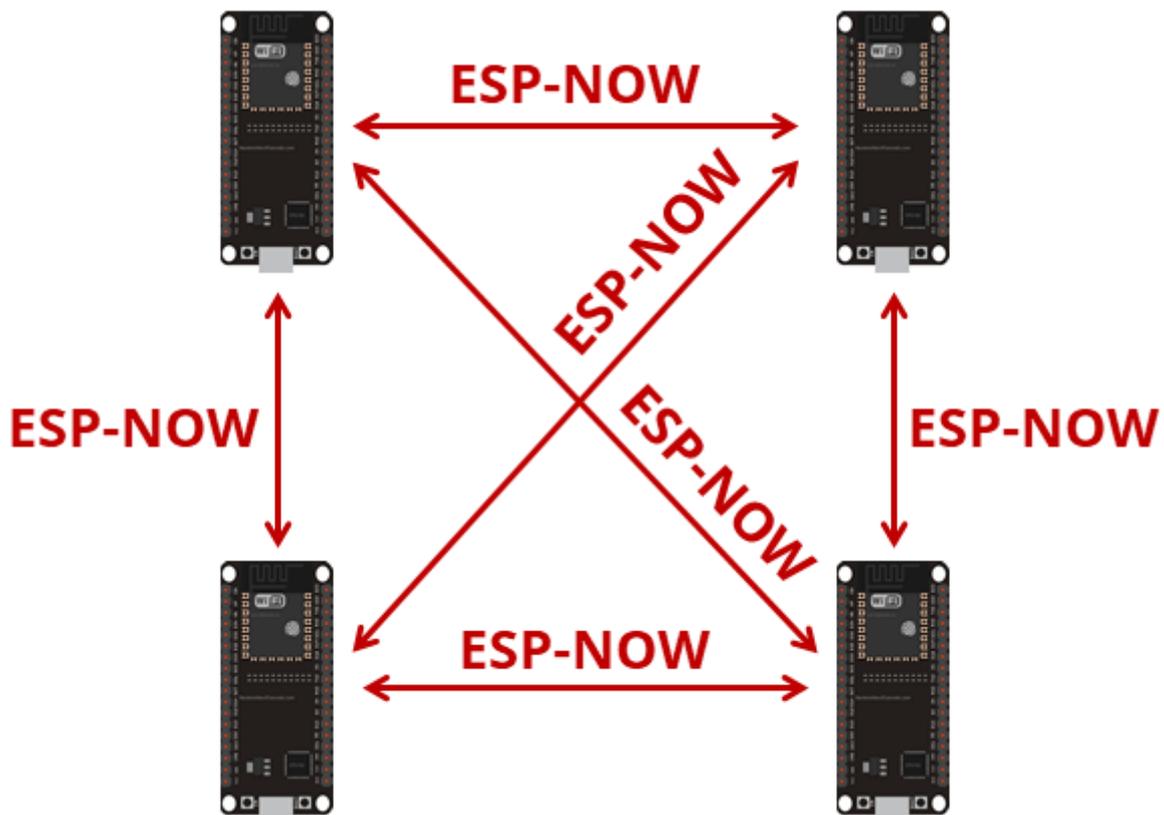
ESP-NOW Two-Way Communication

With ESP-NOW, each board can be a sender and a receiver at the same time. So, you can establish 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 to build 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.

SOURCE CODE

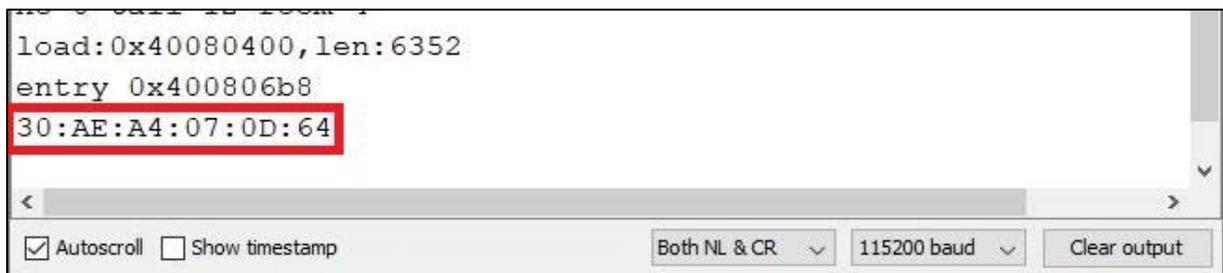
https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Get_MAC_Address/Get_MAC_Address.ino

```
#include "WiFi.h"

void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println(WiFi.macAddress());
}

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 should be printed as follows:



The screenshot shows the Serial Monitor window with the following output:

```
load:0x40080400,len:6352
entry 0x400806b8
30:AE:A4:07:0D:64
```

The MAC address `30:AE:A4:07:0D:64` is highlighted with a red box. The Serial Monitor settings at the bottom are: Autoscroll, Show timestamp, Both NL & CR, 115200 baud, and Clear output.

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, String 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 together. 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 delivered or not;
3. Add a peer device (the receiver). For this, you need to know the 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 (you get if the message was successfully delivered or if it failed).

ESP-NOW Useful Functions

Here's a summary of 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 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.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_1/ESP32_Sender/ESP32_Sender.ino

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

// REPLACE WITH YOUR RECEIVER MAC Address
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

// Structure example to send data
// Must match the receiver structure
typedef struct struct_message {
  char a[32];
```

```

int b;
float c;
String d;
bool e;
} 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
    esp_now_peer_info_t peerInfo;
    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 = "Hello";
    myData.e = 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:A4:07:0D:64, but you need to replace that variable with your own MAC address.

Then, create a structure that contains the type of data we want to send. We called this structure `struct_message` and it contains 5 different variable types. You can change this to send whatever variable types you want.

```
typedef struct struct_message {
    char a[32];
    int b;
    float c;
    String d;
    bool e;
} struct_message;
```

Then, create a new variable of type `struct_message` that is called `myData` that will store the variables values.

```
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 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");
}
```

In the `setup()`, initialize the serial monitor for debugging purposes:

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

Set the device as a Wi-Fi station:

```
WiFi.mode(WIFI_STA);
```

Initialize ESP-NOW:

```
if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
}
```

After successfully initializing ESP-NOW, register the callback function that will be called when a message is sent. In this case, we register for the `OnDataSent()` function created previously.

```
esp_now_register_send_cb(OnDataSent);
```

After that, we need to pair with another ESP-NOW device to send data. That's what we do in the next lines:

```
esp_now_peer_info_t peerInfo;
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 = "Hello";
myData.e = 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, a String 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:

```
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.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_1/ESP32_Receiver/ESP32_Receiver.ino

```
#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;
    String d;
    bool e;
} 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("String: ");
    Serial.println(myData.d);
    Serial.print("Bool: ");
    Serial.println(myData.e);
    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(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;
    String d;
    bool e;
} 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("String: ");
Serial.println(myData.d);
Serial.print("Bool: ");
Serial.println(myData.e);
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:

```
if (esp_now_init() != ESP_OK) {  
    Serial.println("Error initializing ESP-NOW");  
    return;  
}
```

Register for a callback function that will be called when data is received. In this case, we register for the `OnDataRecv()` function that was created previously.

```
esp_now_register_recv_cb(OnDataRecv);
```

Testing ESP-NOW Communication

Upload the sender sketch to the sender ESP32 board and the receiver sketch to the receiver ESP32 board.

Now, open two Arduino IDE windows. One for the receiver, and another for the sender. Open the Serial Monitor for each board. It should be a different COM port for each board.

This is what you should get on the sender side.



```
Last Packet Send Status:      Delivery Success  
Sent with success  
  
Last Packet Send Status:      Delivery Success  
Sent with success  
  
Last Packet Send Status:      Delivery Success
```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output

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 in the sender side).



```
COM3  
Bytes received: 56  
Char: THIS IS A CHAR  
Int: 5  
Float: 1.20  
String: Hello  
Bool: 0  
  
Bytes received: 56  
Char: THIS IS A CHAR  
Int: 15  
Float: 1.20  
String: Hello  
Bool: 0
```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output

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

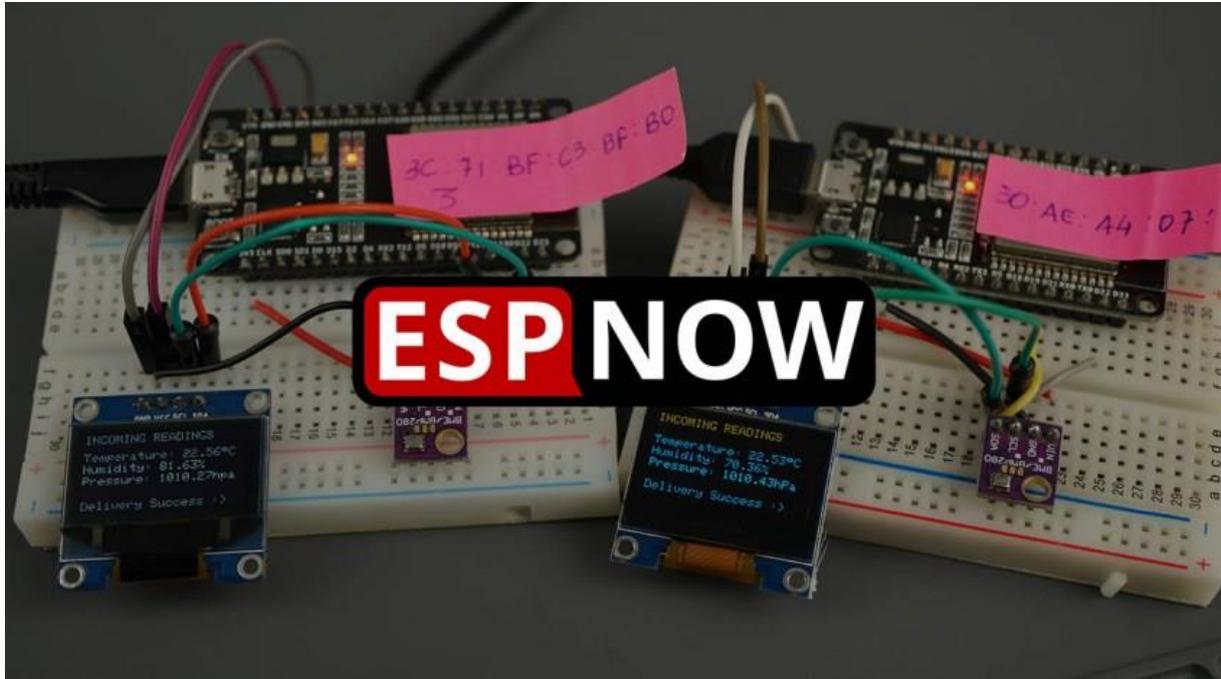
We tried to keep this example as simple as possible so that you better understand how everything works. There are more ESP-NOW related functions that can be useful in your projects, like: managing peers, deleting peers, scanning for slave devices, etc...

For a complete example, in your Arduino IDE, you can go to **File ▶ Examples ▶ ESP32 ▶ ESPNow** 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 by 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.

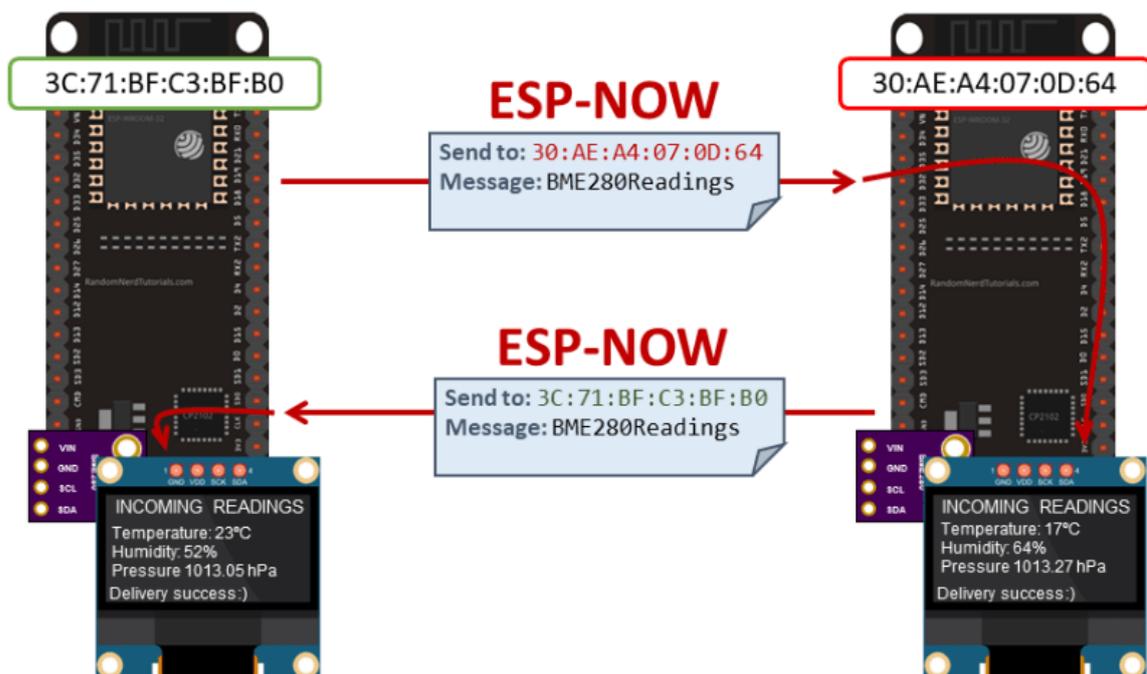
Unit 2 - ESP-NOW Two-Way Communication Between ESP32



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 successfully delivered;
- Each board needs to know the other board MAC address in order 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 having all boards communicating with each other.

Install Libraries

Before proceeding with this project, make sure you check the following prerequisites.

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 library name.

- 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 address. Each board has a unique MAC address.

Upload the following code to each of your boards to get their MAC addresses.

SOURCE CODE

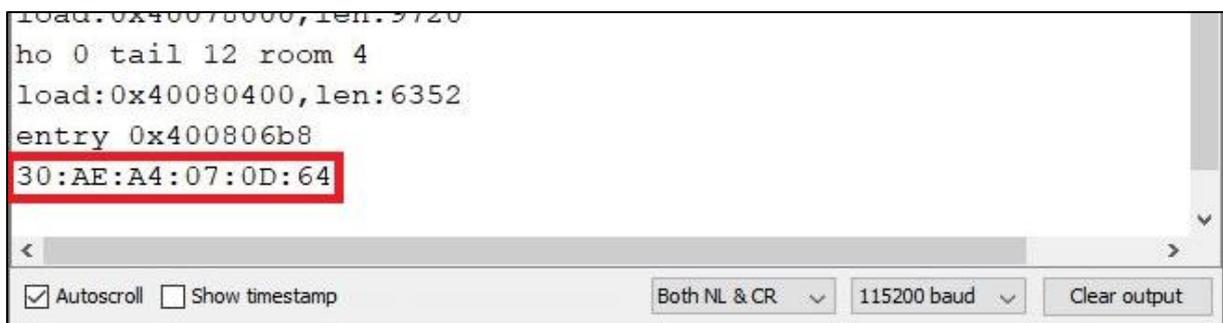
https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Get_MAC_Address/Get_MAC_Address.ino

```
#include "WiFi.h"

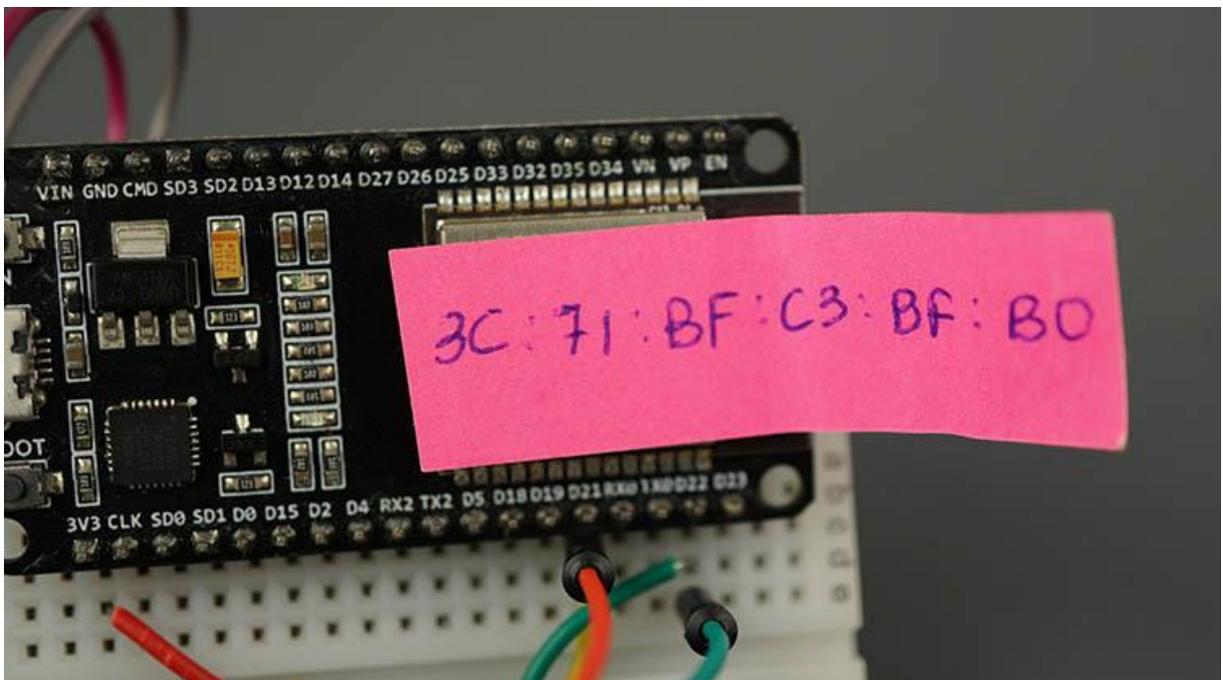
void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println(WiFi.macAddress());
}

void loop() {
}
```

After uploading the code, press the RST/EN button, and the MAC address should be displayed on the Serial Monitor.

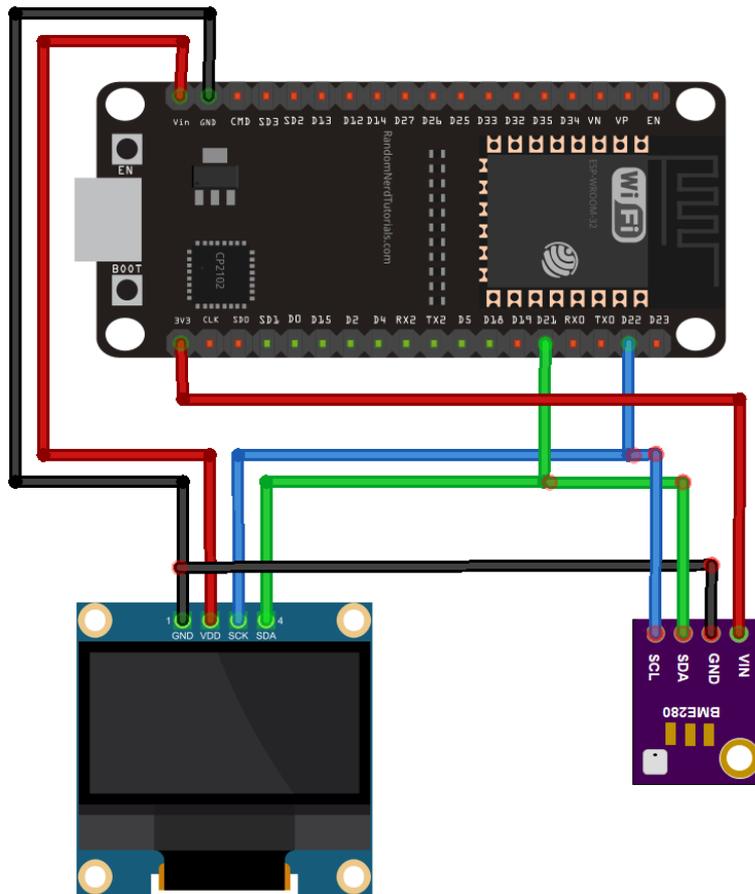


Write down the MAC address of each board to clearly identify them.



Schematic Diagram

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).

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_2/ESP_NOW_Sender_Receiver_Two_Way/ESP_NOW_Sender_Receiver_Two_Way.ino

```
#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;
```

```

// 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
    esp_now_peer_info_t peerInfo;
    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(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 the code works

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:

```

uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

```

Create variables to store temperature, humidity and pressure readings from the BME280 sensor. These readings will be sent to the other board:

```

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 delivered successfully to the other board.

```

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 delivered or not. If the message is delivered successfully, the status variable returns 0, so we can set our success message to "Delivery 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 ? "Delivery Success"
: "Delivery 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
esp_now_peer_info_t peerInfo;
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(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.

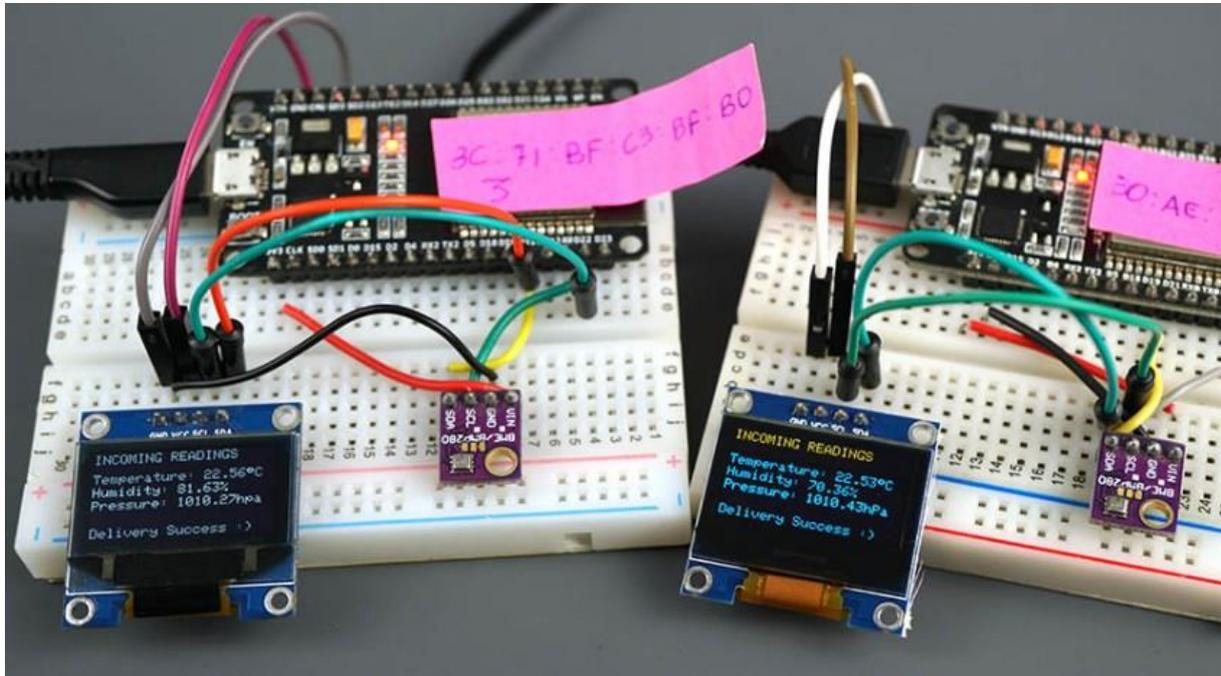
```
delay(10000);
```

That's pretty much how the code works.

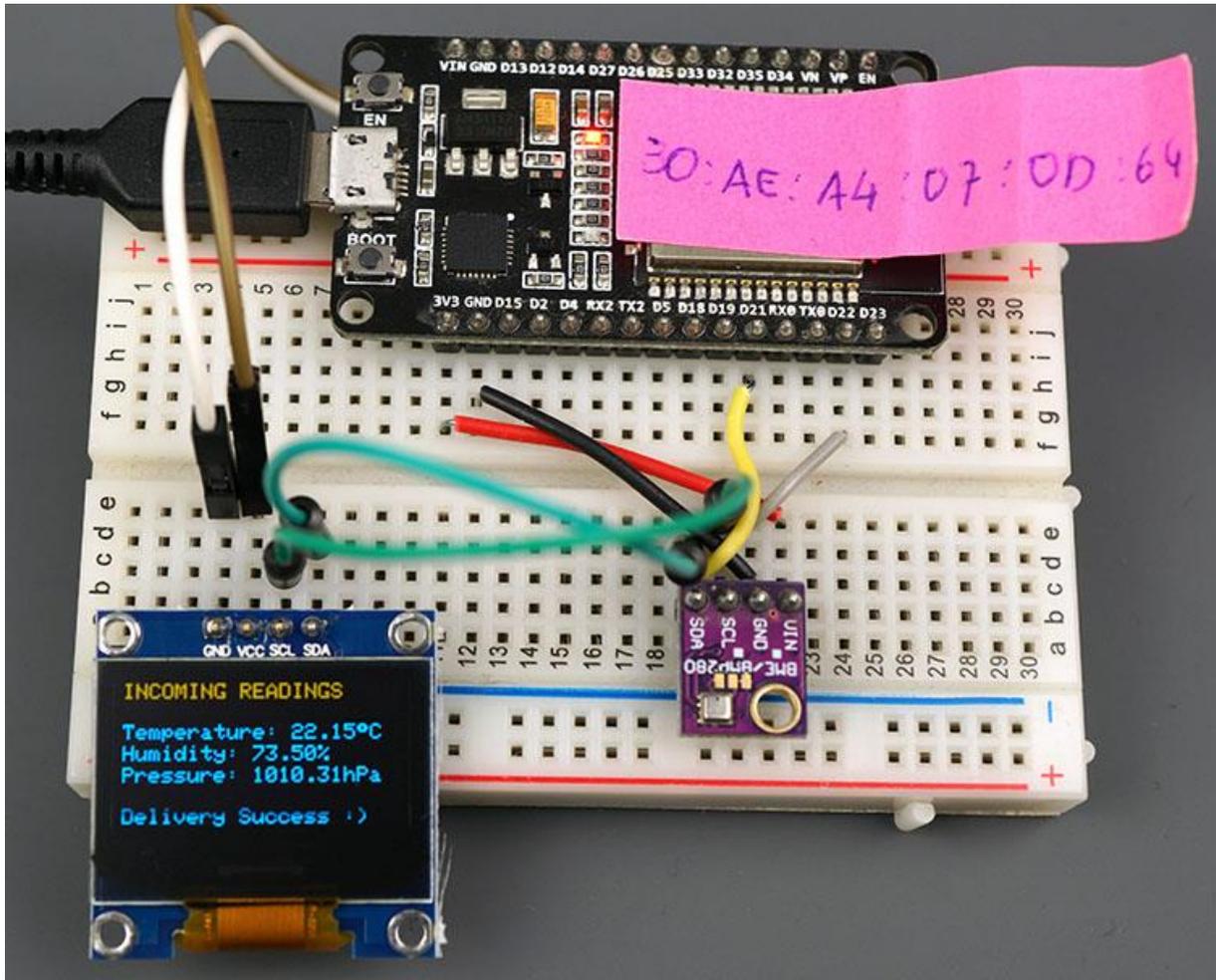
Now, upload the code to both of your boards. You just need to modify the code with the MAC address of the board you're sending data to.

Demonstration

After uploading the code to both boards, you should see the OLED displaying the sensor readings from the other board, as well as a success delivery message.



As you can see, it's working as expected:



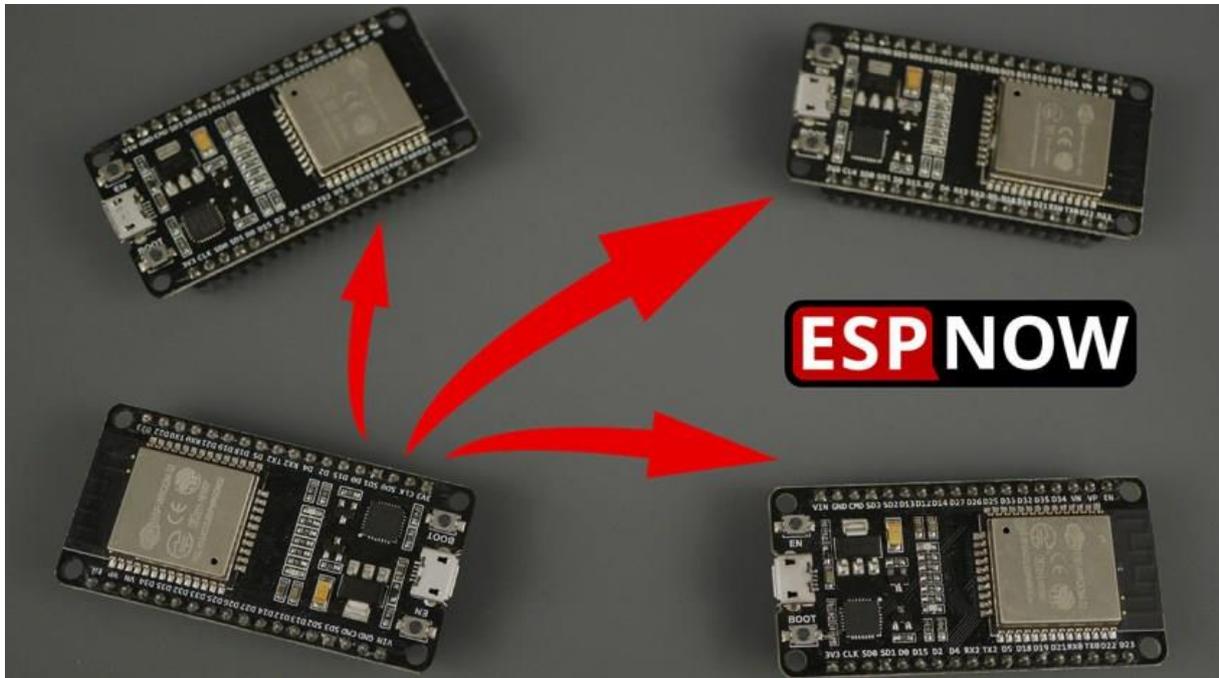
Wrapping Up

In this unit we've shown you how to establish a two-way communication with two ESP32 board 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).

Unit 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 to send messages via ESP-NOW, you need to know the receiver boards' MAC address.

Upload the following code to each of your receiver boards to get their MAC addresses.

SOURCE CODE

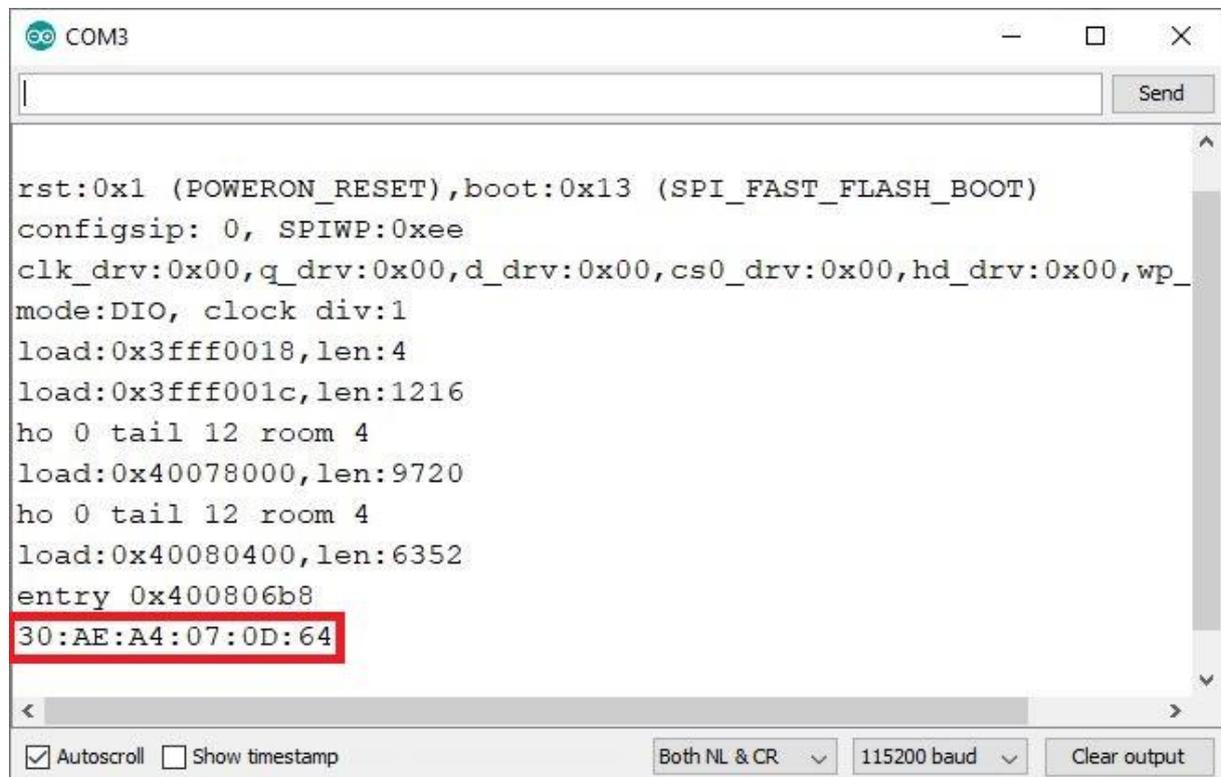
[https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP NOW/Get MAC Address/Get MAC Address.ino](https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP%20NOW/Get%20MAC%20Address/Get%20MAC%20Address.ino)

```
#include "WiFi.h"

void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println(WiFi.macAddress());
}

void loop() {
}
```

After uploading the code, press the RST/EN button, and the MAC address should be displayed on the Serial Monitor.



```
COM3
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
config:0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
entry 0x400806b8
30:AE:A4:07:0D:64
```

We suggest that you write down the boards' MAC address on a label to clearly 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.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_3/ESP_NOW_Sender_Multiple_Boards_Same_Message/ESP_NOW_Sender_Multiple_Boards_Same_Message.ino

```
#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, , , , , };

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
    snprintf(macStr, sizeof(macStr), "%02x:%02x:%02x:%02x:%02x:%02x",
             mac_addr[0], mac_addr[1], mac_addr[2], mac_addr[3], mac_addr[4],
             mac_addr[5]);
    Serial.print(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
    esp_now_peer_info_t peerInfo;
    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 the code works

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. In our example, we're sending data to three boards.

```

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.

```
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
    snprintf(macStr, sizeof(macStr), "%02x:%02x:%02x:%02x:%02x:%02x",
mac_addr[0], mac_addr[1], mac_addr[2], mac_addr[3], mac_addr[4],
mac_addr[5]);
    Serial.print(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:

```
esp_now_peer_info_t peerInfo;
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 duplicate 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 with 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 result = esp_now_send(broadcastAddress1, (uint8_t *)
&test, sizeof(test_struct));

if (result == ESP_OK) {
    Serial.println("Sent with success");
}
else {
    Serial.println("Error sending the data");
}
delay(2000);
```

Do the same for the other boards. For the second board send the `test2` structure:

```
esp_err_t result = esp_now_send(broadcastAddress2, (uint8_t *)
&test2, sizeof(test_struct));

if (result == ESP_OK) {
    Serial.println("Sent with success");
}
else {
    Serial.println("Error sending the data");
}
delay(2000);
```

And finally, for the third board, send the `test3` structure:

```

esp_err_t result = esp_now_send(broadcastAddress2, (uint8_t *)
&test2, sizeof(test_struct));

if (result == ESP_OK) {
    Serial.println("Sent with success");
}
else {
    Serial.println("Error sending the data");
}
delay(2000);

```

Here's the complete code that sends a different message to each board.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_3/ESP_NOW_Sender_Multiple_Boards_Different_Message/ESP_NOW_Sender_Multiple_Boards_Different_Message.ino

ESP32 Receiver Code (ESP-NOW)

Upload the next code to the receiver boards (in our example, we've used three receiver boards).

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_3/ESP_NOW_Receiver/ESP_NOW_Receiver.ino

```

#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 OnDataRcv(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(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 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 a 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:

```
if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
}
```

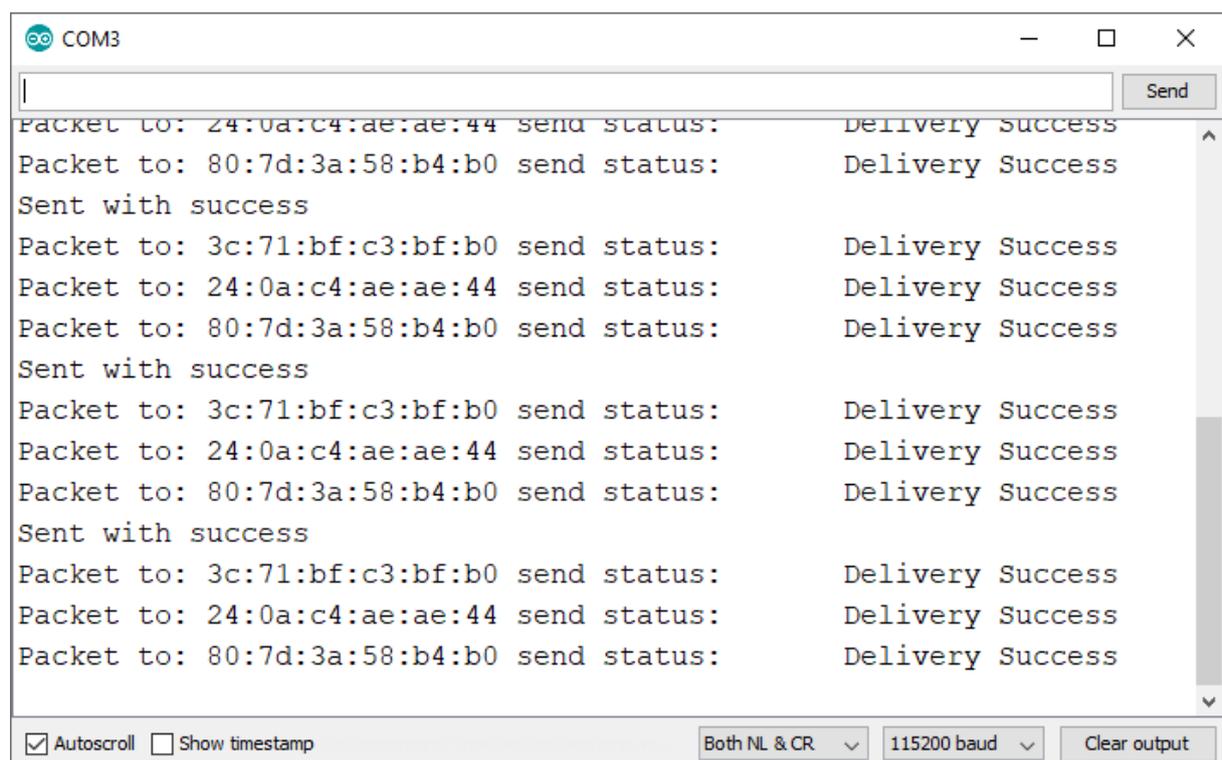
Register for a callback function that will be called when data is received. In this case, we register for the `OnDataRecv()` function that was created previously.

```
esp_now_register_recv_cb(OnDataRecv);
```

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.

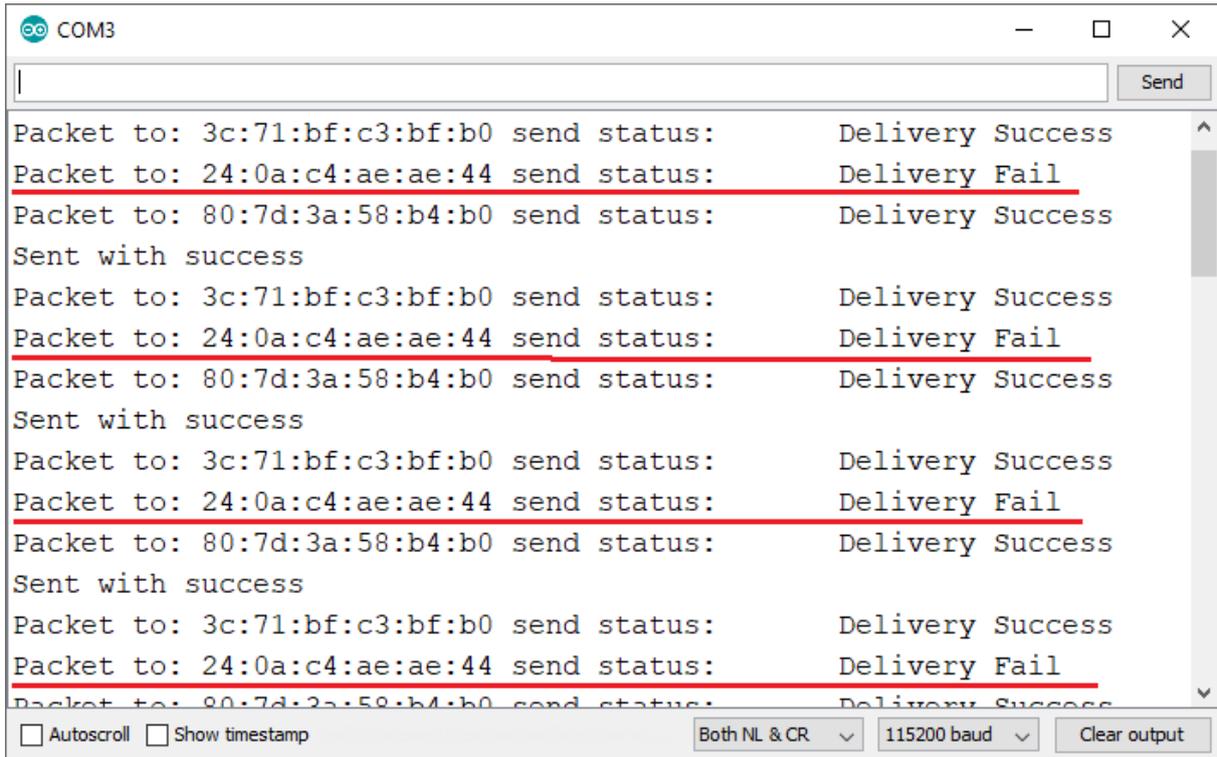


The screenshot shows the Arduino IDE Serial Monitor window for COM3. The window title is "COM3" and it has standard window controls (minimize, maximize, close). At the top right, there is a "Send" button. The main area displays the following text:

```
Packet to: 24:0a:c4:ae:ae:44 send status: Delivery Success
Packet to: 80:7d:3a:58:b4:b0 send status: Delivery Success
Sent with success
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Packet to: 24:0a:c4:ae:ae:44 send status: Delivery Success
Packet to: 80:7d:3a:58:b4:b0 send status: Delivery Success
Sent with success
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Packet to: 24:0a:c4:ae:ae:44 send status: Delivery Success
Packet to: 80:7d:3a:58:b4:b0 send status: Delivery Success
Sent with success
Packet to: 3c:71:bf:c3:bf:b0 send status: Delivery Success
Packet to: 24:0a:c4:ae:ae:44 send status: Delivery Success
Packet to: 80:7d:3a:58:b4:b0 send status: Delivery Success
```

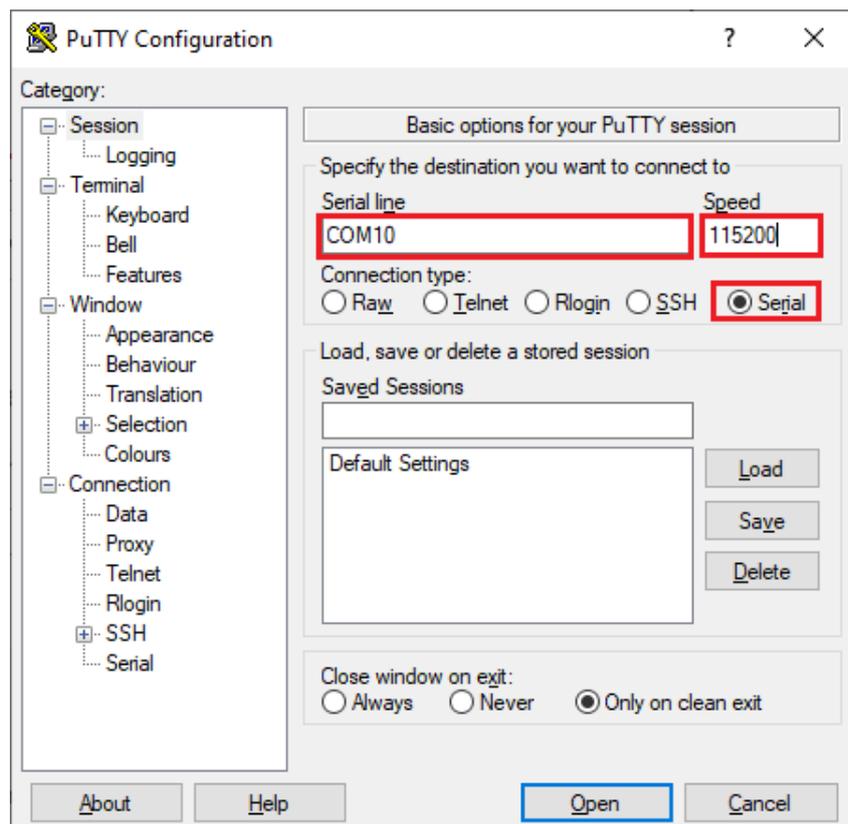
At the bottom of the window, there are several controls: a checked "Autoscroll" checkbox, an unchecked "Show timestamp" checkbox, a dropdown menu set to "Both NL & CR", a dropdown menu set to "115200 baud", and a "Clear output" button.

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.

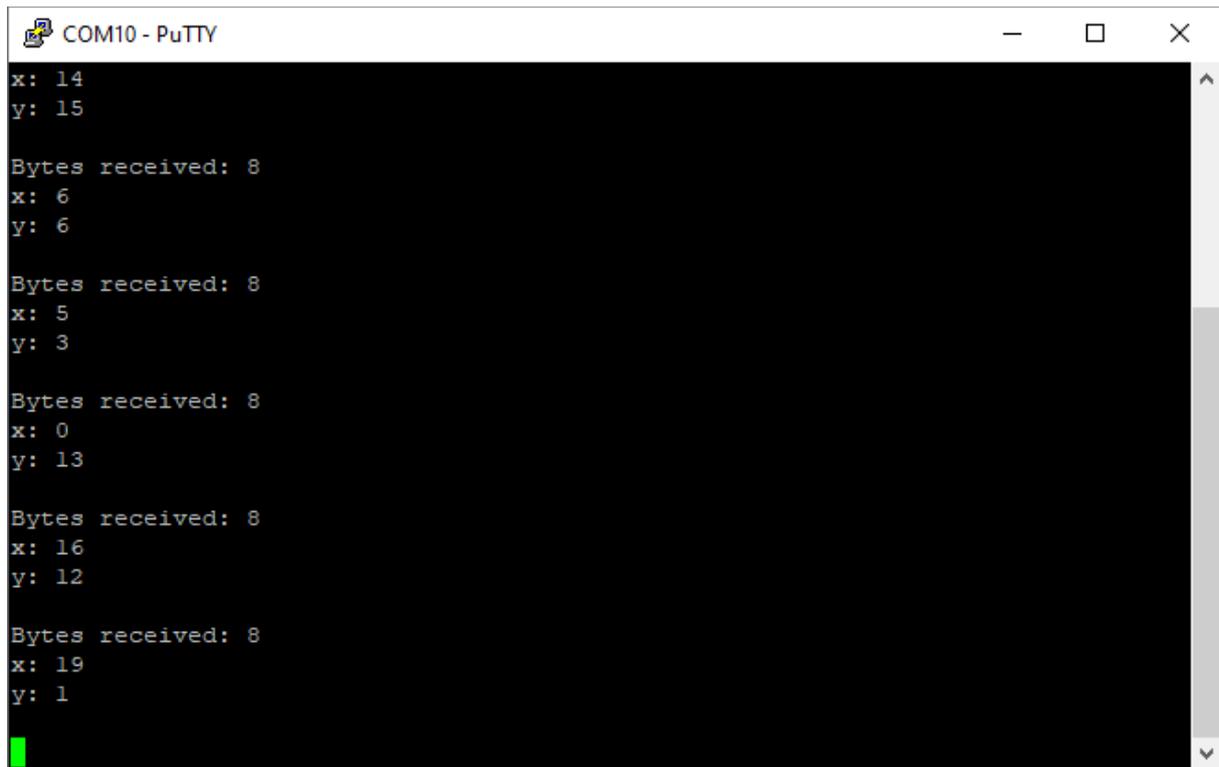


If you want to check if the boards are actually receiving the messages, you can open the Serial Monitor for the COM port they are connected to, or you can use [PuTTY](#) to establish a serial communication with your boards.

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.



```
COM10 - PuTTY
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
█
```

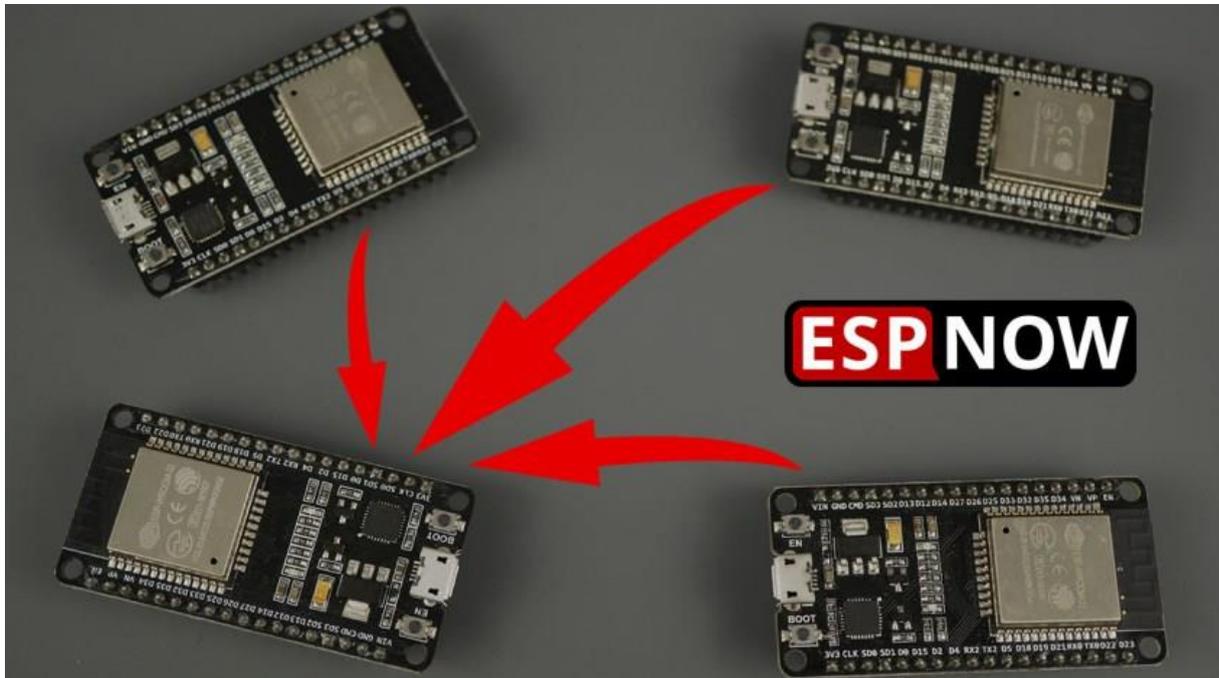
Open a serial communication for each of your boards and check that they are receiving the messages.

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.

Unit 4 - ESP-NOW Receive Data from Multiple Boards (many-to-one)



This Unit shows how to setup 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.

SOURCE CODE

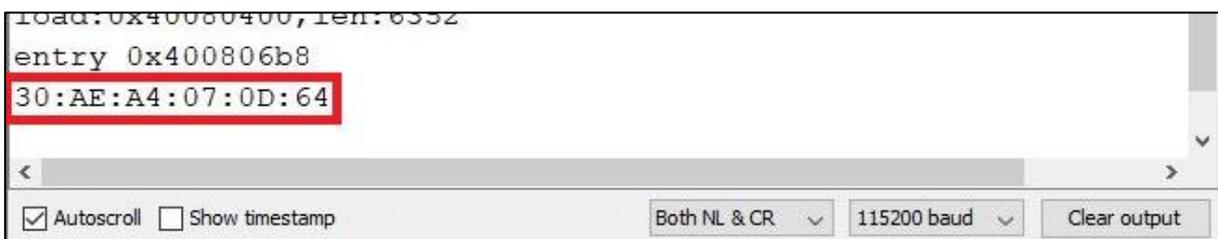
https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Get_MAC_Address/Get_MAC_Address.ino

```
#include "WiFi.h"

void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println(WiFi.macAddress());
}

void loop() {
}
```

After uploading the code, press the RST/EN button, and the MAC address should be displayed on the Serial Monitor.

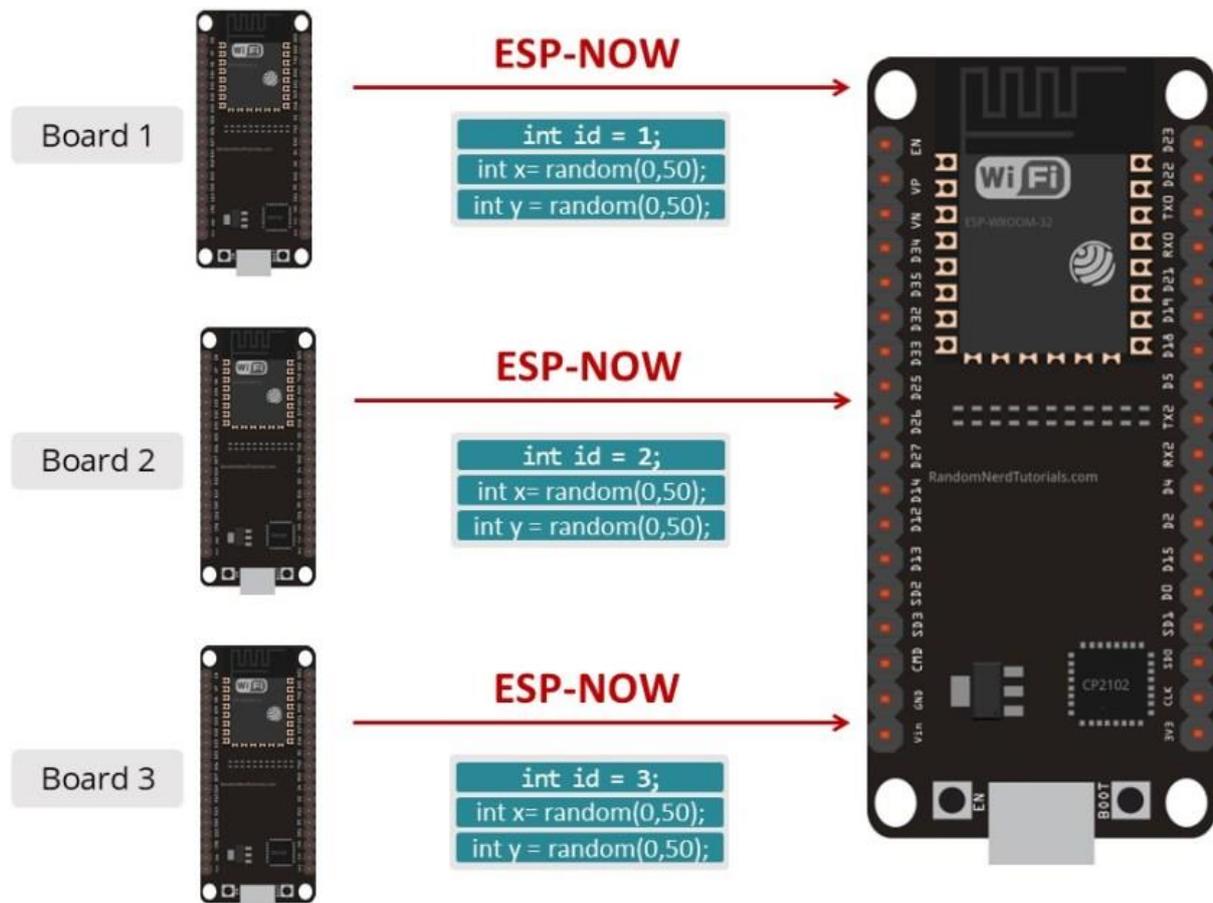


ESP32 Sender Code (ESP-NOW)

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 a little 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.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_4/ESP_NOW_Sender/ESP_NOW_Sender.ino

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

// REPLACE WITH THE RECEIVER'S MAC Address
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
```

```

// 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 Trasmitted packet
    esp_now_register_send_cb(OnDataSent);

    // Register peer
    esp_now_peer_info_t peerInfo;
    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};
```

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
esp_now_peer_info_t peerInfo;
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.

```
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.

```
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.

SOURCE CODE

[https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP NOW/Unit 4/ESP NOW Receiver Multiple Boards/ESP NOW Receiver Multiple Boards.ino](https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP%20NOW/Unit%204/ESP%20NOW%20Receiver%20Multiple%20Boards/ESP%20NOW%20Receiver%20Multiple%20Boards.ino)

```
#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) {
    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(&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(OnDataRecv);
}
void loop() {
  // Access 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) {
  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);

```

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]
<pre> structure board1 int id = 1; int x= random(0,50); int y = random(0,50); </pre>	<pre> structure board2 int id = 2; int x= random(0,50); int y = random(0,50); </pre>	<pre> structure board3 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(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:

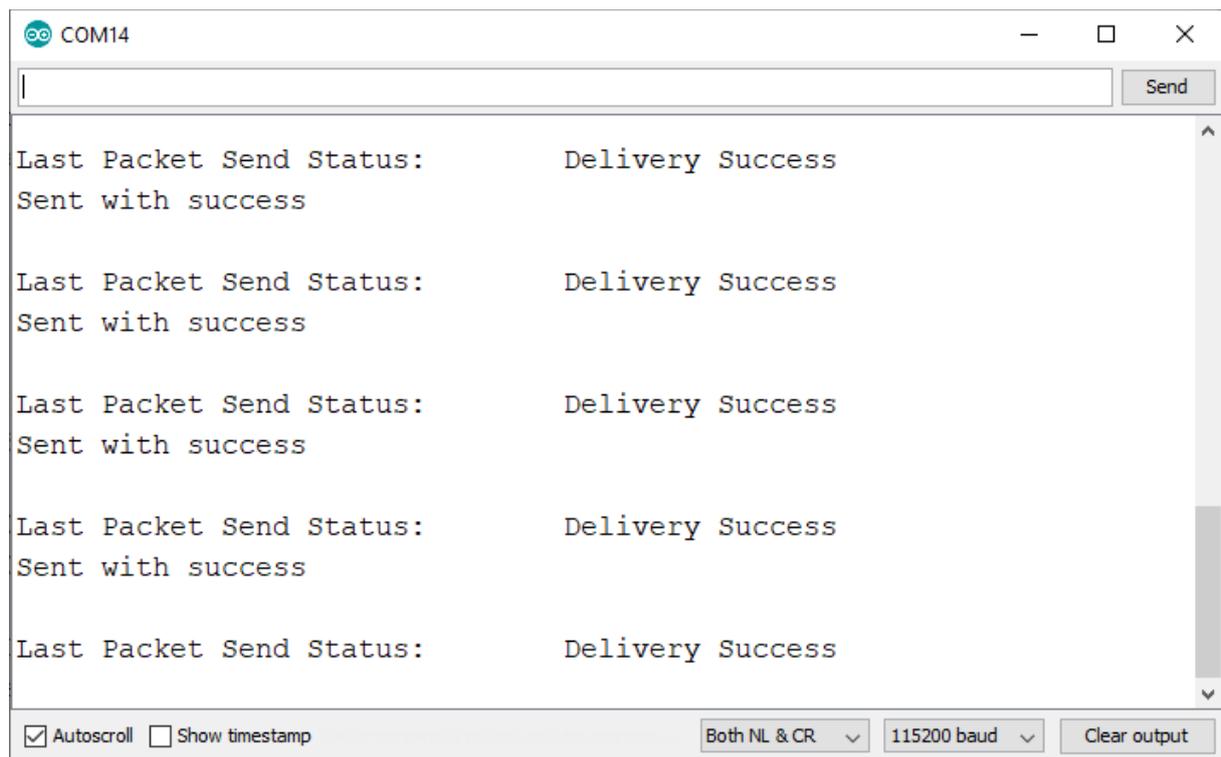
```
int board1X = boardsStruct[0].x;
```

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 properly.



On the receiver board, you should be receiving the packets from all the other boards. In this test, we were receiving data from 5 different boards.

```
board1X = 4
board1Y = 8
board2X = 17
board2Y = 13
board3X = 21
board3Y = 23
board4X = 32
board4Y = 34
board5X = 49
board5Y = 48
Packet received from: 84:0d:8e:d0:70:f0
Board ID 4: 12 bytes
x value: 37
y value: 34
```

Wrapping Up

In this tutorial you've learned how setup 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 should 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 displays the received messages – that's what we're going to do in the next Unit.

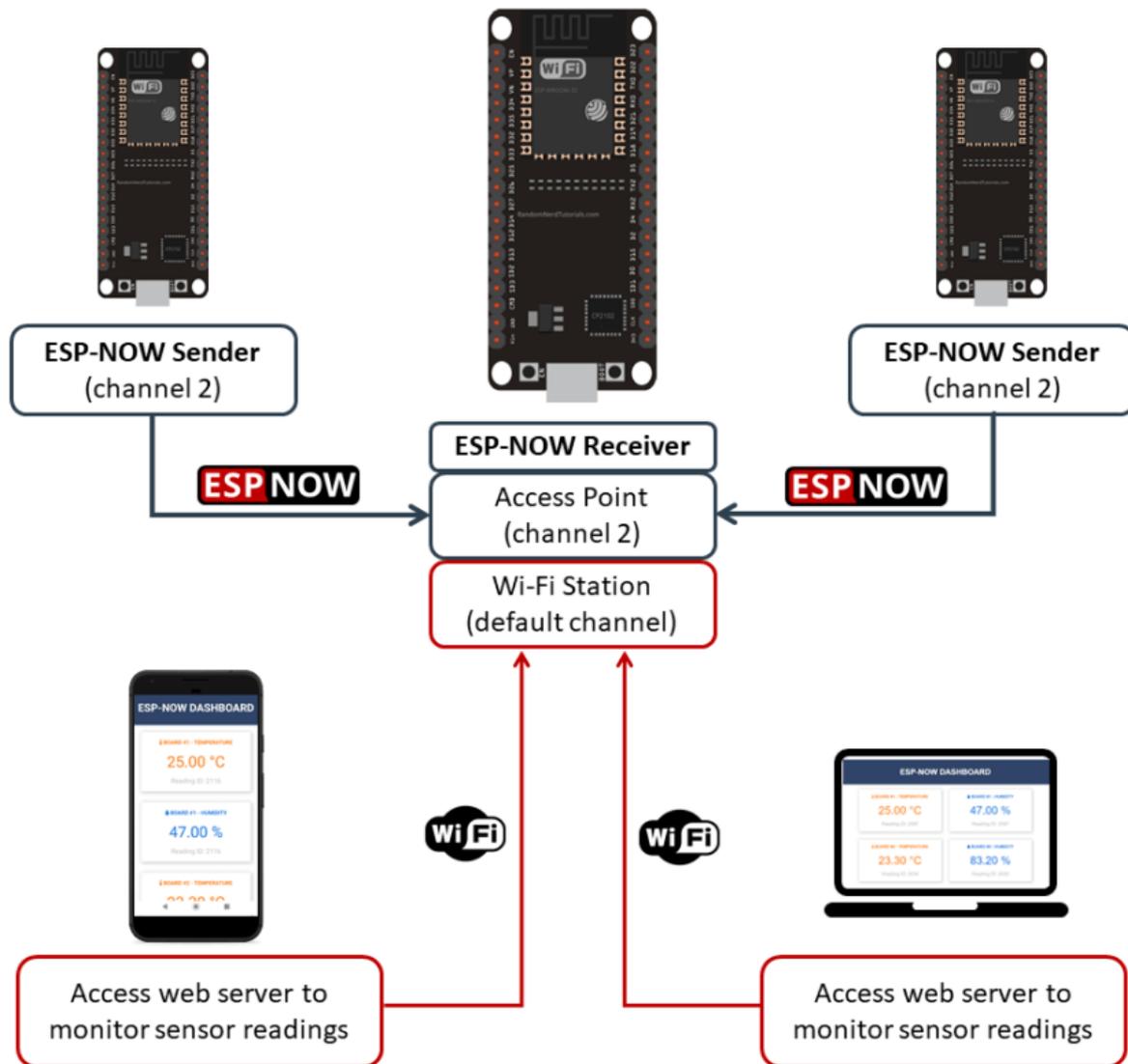
Unit 5 - ESP-NOW Web Server Sensor Dashboard (ESP-NOW + Wi-Fi)



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



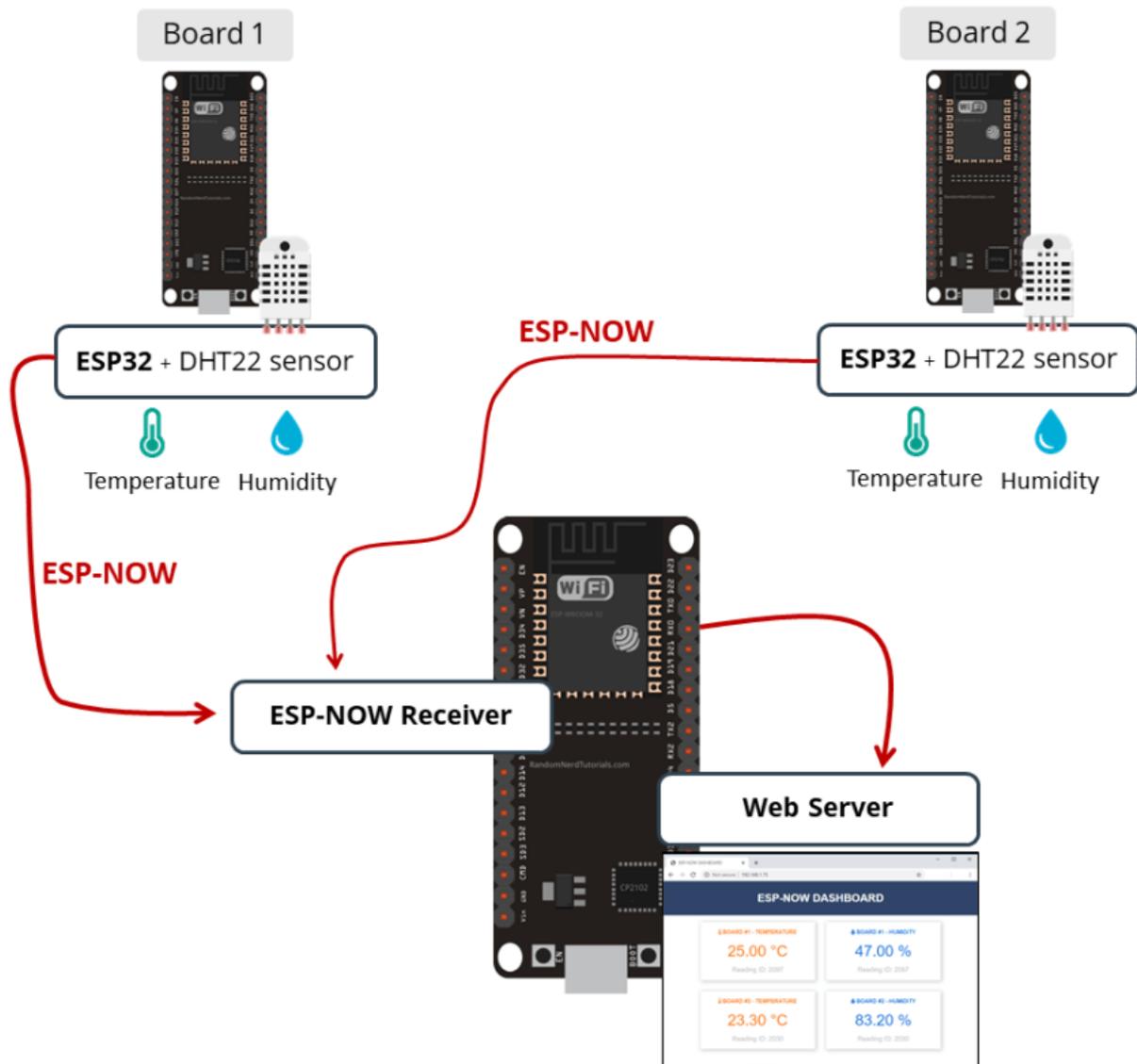
There are a few things you need to take into account if you want to use Wi-Fi to host a web server and use ESP-NOW simultaneously to receive sensor readings from other boards:

- The ESP32 receiver board must be set both as a station and as an access point;
- The ESP32 should use a different Wi-Fi channel for station mode and different channel for the access point;
- We'll set channel 2 for the access point. This should be the same Wi-Fi channel defined in the ESP32 sender boards;
- The sender boards should be connected to the receiver via access point on channel 2.

We're not sure if there's better way to use ESP-NOW and Wi-Fi simultaneously. This solution worked reliably for us. If you have another working solution, you can share it with us.

Project Overview

The following diagram shows a high-level overview of the project we'll build.

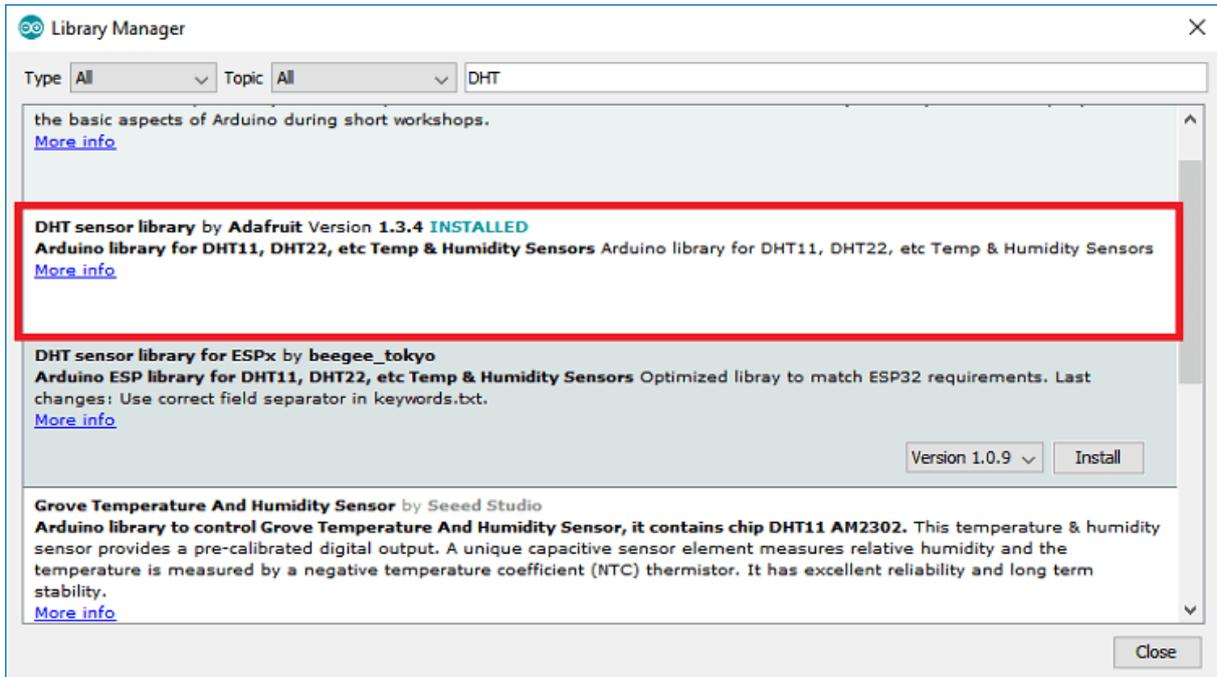


- There are two ESP32 sender boards that send DHT22 temperature and humidity readings via ESP-NOW to one ESP32 receiver board (ESP-NOW many to one configuration);
- The ESP32 receiver board receives the packets and displays the readings on a web server;
- The web server is updated automatically every time it receives a new reading using Server-Sent Events (SSE).

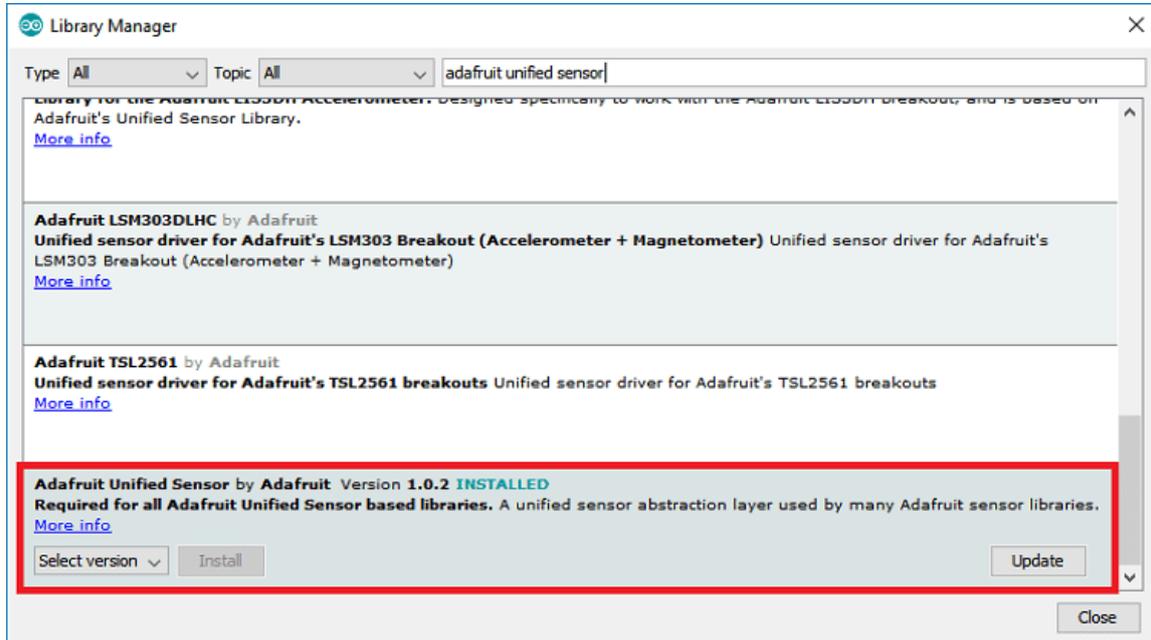
DHT Libraries

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](#). To use this library you also need to install the [Adafruit Unified Sensor library](#). Follow the next steps to install those libraries.

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.



3. After installing the DHT library from Adafruit, type “**Adafruit Unified Sensor**” in the search box. Scroll all the way down to find the library and install it.



After installing the libraries, restart your Arduino IDE.

Async Web Server Libraries

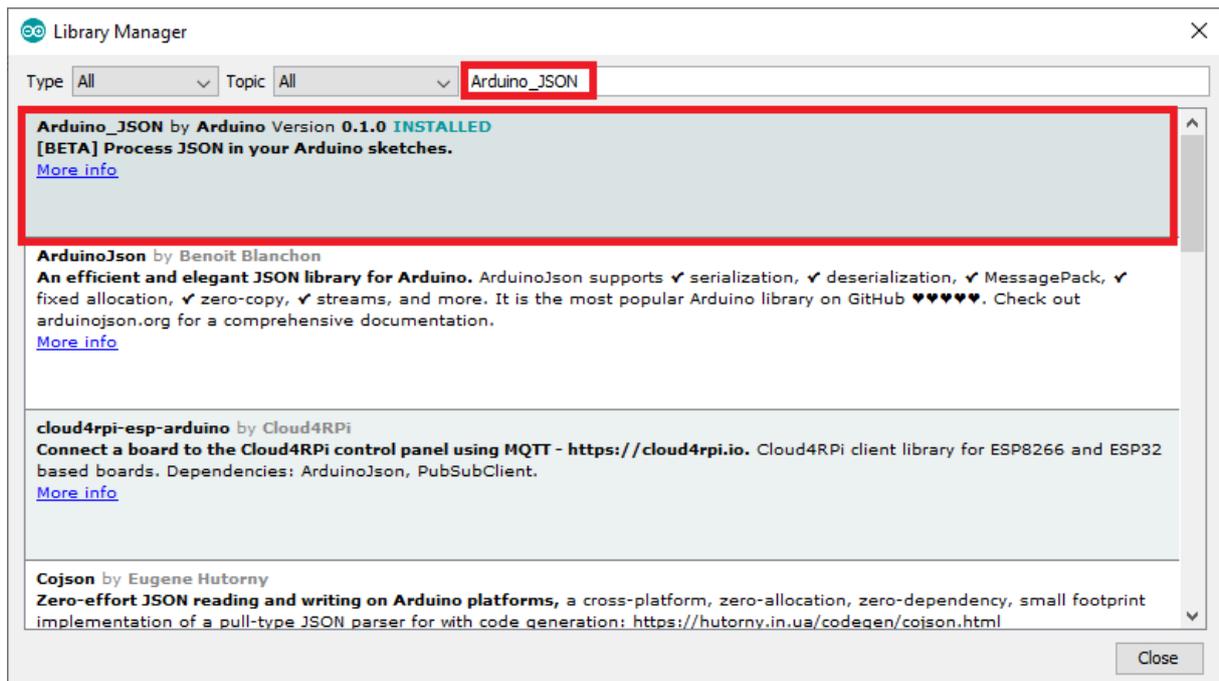
To build the web server you need to install the following libraries:

- [ESPAsyncWebServer](#)
- [AsyncTCP](#)

These libraries aren't available to install through the Arduino Library Manager, so you need to copy the library files to the Arduino Installation Libraries folder. Alternatively, in your Arduino IDE, you can 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:



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 know the receiver board's MAC address. Upload the following code to get its MAC address.

SOURCE CODE

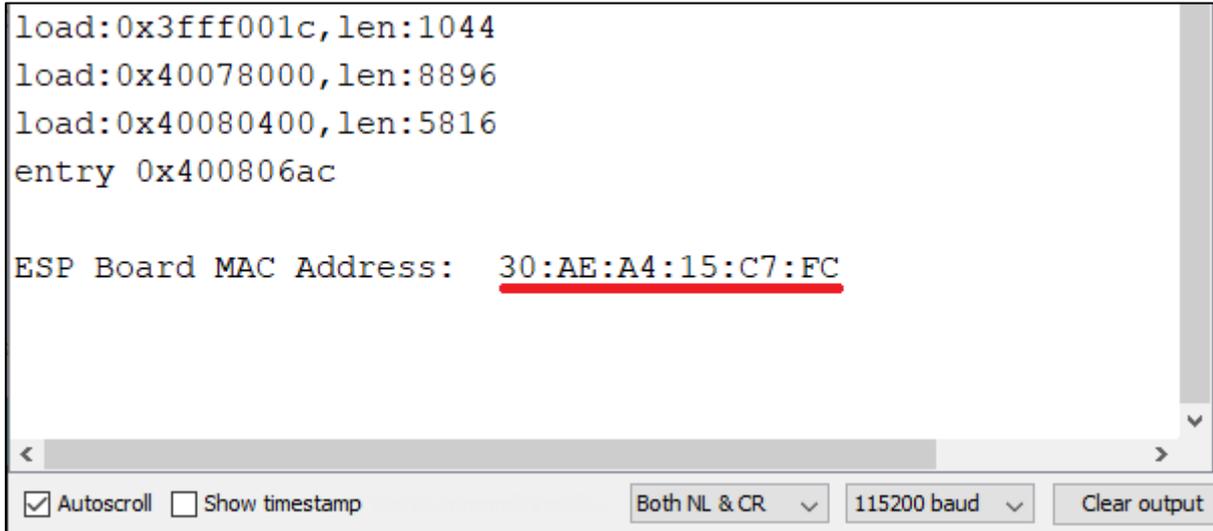
https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Get_MAC_Address/Get_MAC_Address.ino

```
#include "WiFi.h"

void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println(WiFi.macAddress());
}

void loop() {
}
```

After uploading the code, press the RST/EN button, and the MAC address should be displayed on the Serial Monitor.



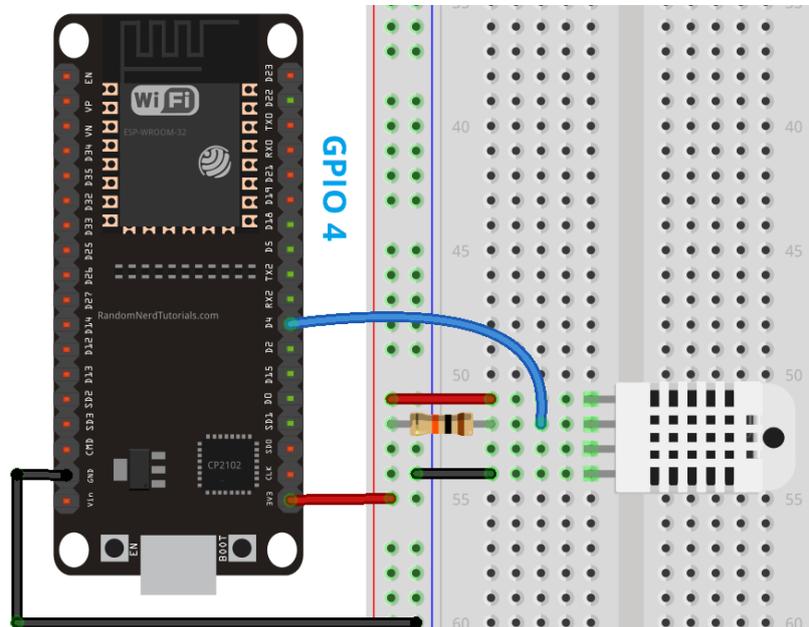
```
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5816
entry 0x400806ac

ESP Board MAC Address: 30:AE:A4:15:C7:FC
```

Autoscroll Show timestamp Both NL & CR 115200 baud Clear output

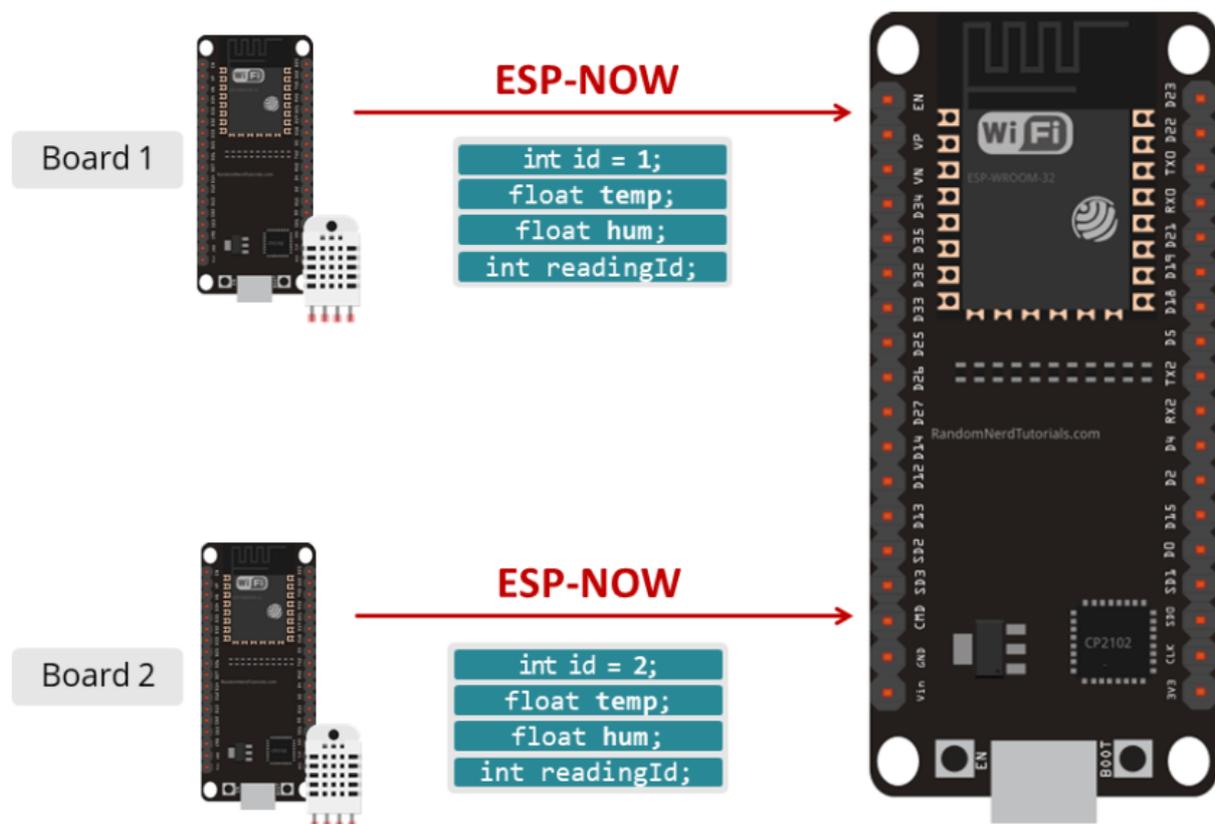
ESP32 Sender Circuit

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.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_5/ESP_NOW_Sender/ESP_NOW_Sender.ino

```
#include <esp_now.h>
#include <WiFi.h>
#include <Adafruit_Sensor.h>
#include <DHT.h>

//The gateway access point credentials
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";

// 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[] = {0x30, 0xAE, 0xA4, 0x15, 0xC7, 0xFC};

//Wi-Fi channel (must match the gateway wi-fi channel as an access
point)
#define CHAN_AP 2

//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;

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
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.println("Connecting to Access Point...");
    }

    //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
esp_now_peer_info_t peerInfo;
memcpy(peerInfo.peer_addr, broadcastAddress, 6);
peerInfo.channel = CHAN_AP;
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 = 2;
        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 the Code Works

Start by importing the required libraries:

```

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

```

Receiver's Access Point Credentials

As we've seen previously, the sender board needs to connect to the receiver board access point. The receiver board access point credentials are the ones defined in the next lines. If you change the access point credentials, you need to change them for every board.

```

//The gateway access point credentials
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";

```

Note: these are not your local network credentials. These are the receiver board access point credentials.

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:

```
uint8_t broadcastAddress[] = {0x30, 0xAE, 0xA4, 0x15, 0xC7, 0xFC};
```

ESP-NOW Wi-Fi Channel

As mentioned previously, we'll use Wi-Fi channel 2 to communicate via ESP-NOW.

```
#define CHAN_AP 2
```

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;
    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;
const long interval = 10000;
```

Initialize the `readingId` variable – it keeps track of the number of readings sent.

```
unsigned int readingId = 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 and connect it to the receiver's access point.

```
WiFi.mode(WIFI_STA);
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(1000);
  Serial.println("Connecting to Access Point...");
}
```

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

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
esp_now_peer_info_t peerInfo;
memcpy(peerInfo.peer_addr, broadcastAddress, 6);
peerInfo.channel = CHAN_AP;
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.

```
unsigned long currentMillis = millis();
if (currentMillis - previousMillis >= interval) {
  // Save the last time a new reading was published
  previousMillis = currentMillis;
```

Send ESP-NOW Message

Finally, send the message structure 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 (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.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP_NOW/Unit_5/ESP_NOW_Receiver_Web_Server/ESP_NOW_Receiver_Web_Server.ino

```

#include <esp_now.h>
#include <WiFi.h>
#include "ESPAsyncWebServer.h"
#include <Arduino_JSON.h>

// Replace with your network credentials (STATION)
const char* ssidStation = "MEO-D32A40";
const char* passwordStation = "384e6d3cec";

// ACCESS POINT credentials
const char* ssidAP = "ESP32-Access-Point";
const char* passwordAP = "123456789";

// Wi-Fi channel for the access point (must match the sender channel)
#define CHAN_AP 2

// 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
    href="https://use.fontawesome.com/releases/v5.7.2/css/all.css"
    integrity="sha384-
    fnmOCqBtLWI1j8LyTjo7mOUSTjsKC4pOpQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr"
    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> &percent;</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>
</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(ssidStation, passwordStation);
    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());
    // Set device as an access point
    WiFi.softAP(ssidAP, passwordAP, CHAN_AP, true);
    // 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(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 the Code Works

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 needed 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* ssidStation = "REPLACE_WITH_YOUR_SSID";
const char* passwordStation = "REPLACE_WITH_YOUR_PASSWORD";

```

As explained previously, this board needs to be set as a Wi-Fi station and access point simultaneously. In the following lines insert the credentials for the access point (you can leave them as they are).

```

const char* ssidAP = "ESP32-Access-Point";
const char* passwordAP = "123456789";

```

Define the ESP-NOW Wi-Fi channel (channel 2).

```

#define CHAN_AP 2

```

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.

Important: Server-sent events are not supported on Internet Explorer.

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) {
```

Inside that function, print the sender's MAC address:

```
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);
```

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;
```

Basically, 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);
// Set device as a Wi-Fi Station
WiFi.begin(ssidStation, passwordStation);
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());
```

Create an access point with the SSID, password and channel defined earlier.

```
WiFi.softAP(ssidAP, passwordAP, CHAN_AP, true);
```

Initialize 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(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);  
}
```

Finally, start the server.

```
server.begin();
```

loop()

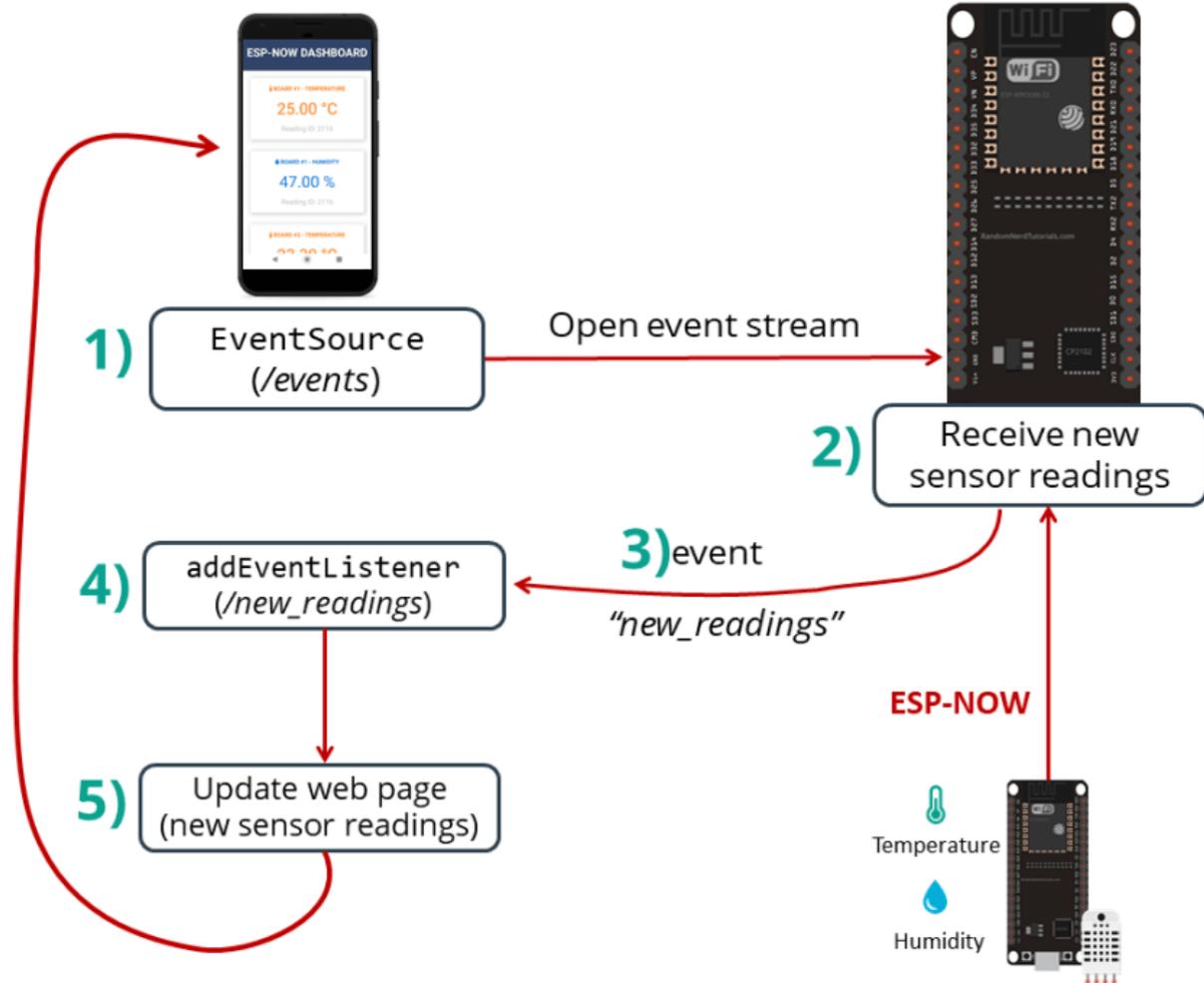
In the `loop()`, send a ping every 5 seconds. This is used to check on the client side, if the server is still running.

```
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)

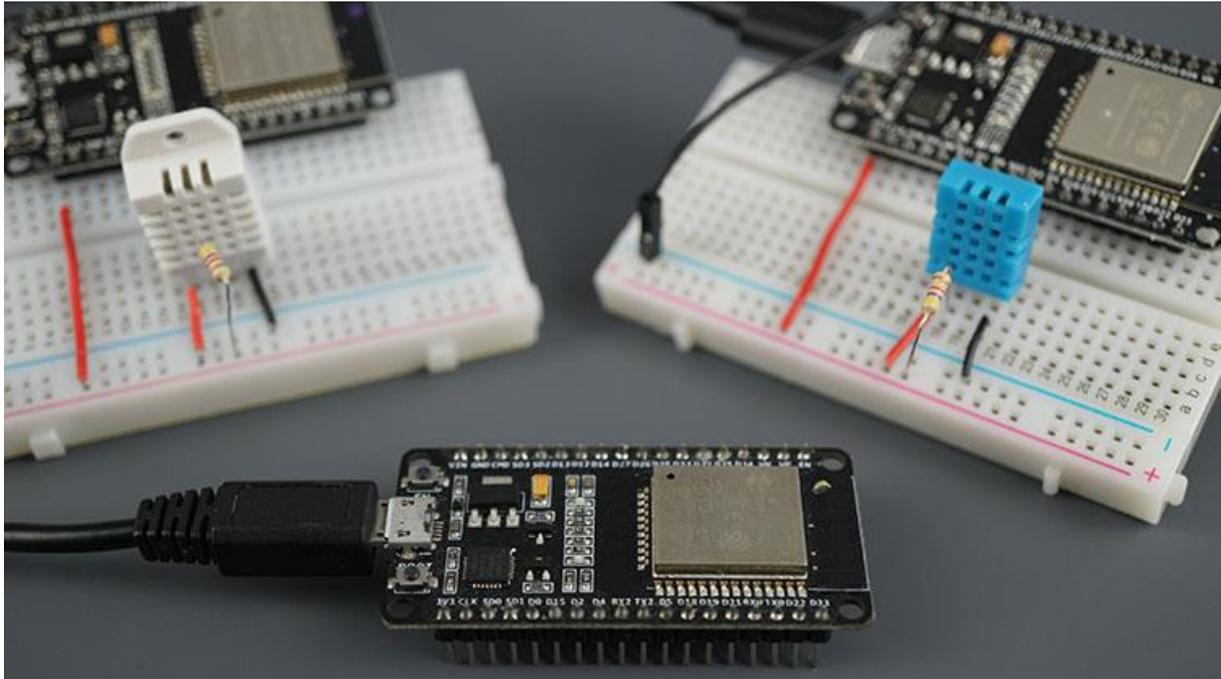


Demonstration

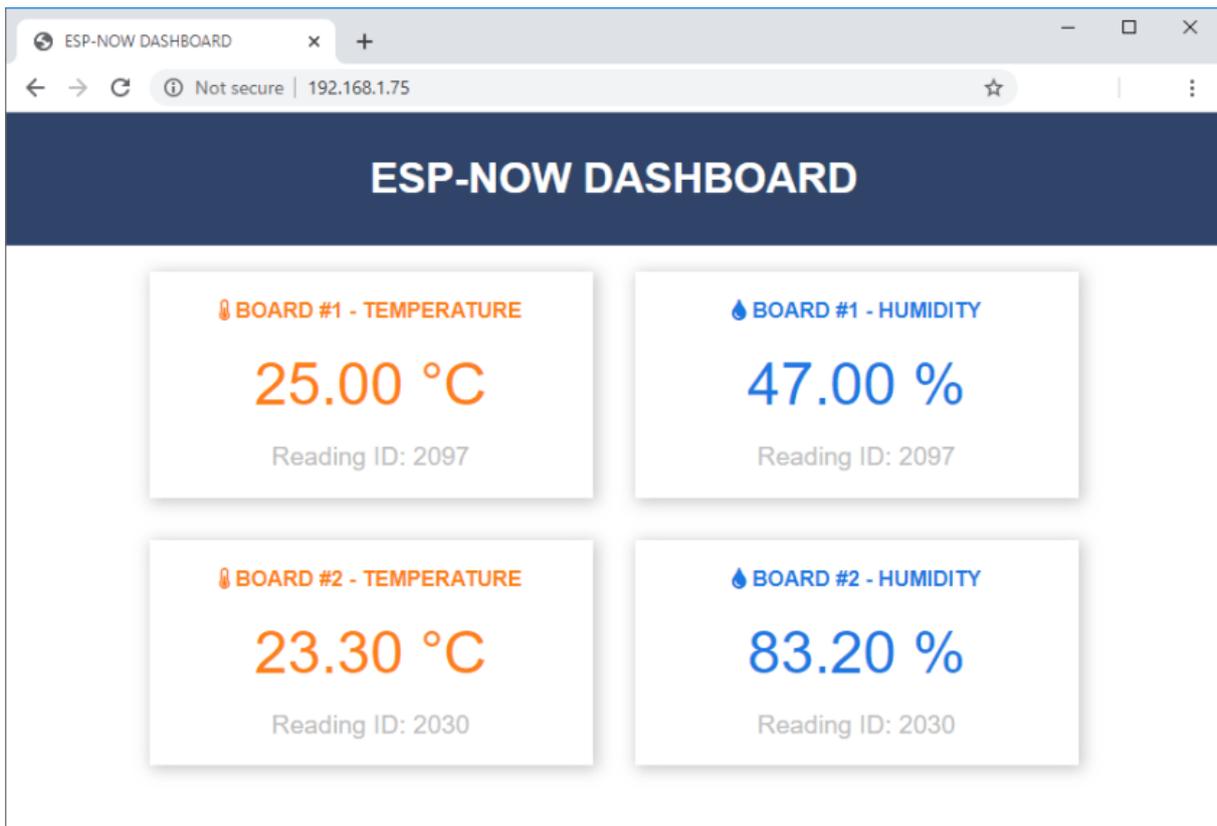
After uploading the code to the receiver board, press the on-board EN/RST button. The ESP IP address should be printed on the Serial Monitor.

```
COM3
load:0x40080400,len:6352
entry 0x400806b8
Setting as a Wi-Fi Station..
Station IP Address: 192.168.1.75
Wi-Fi Channel: 6
Packet received from: 30:ae:a4:f6:7d:4c
Board ID 2: 16 bytes
t value: 23.90
h value: 85.40
readingID value: 14
```

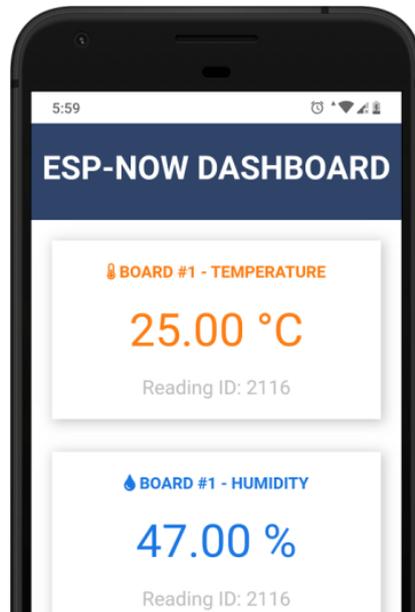
If everything is going as expected, the ESP32 receiver board should start receiving sensor readings from the other boards.



Open a browser on your local network and type the ESP32 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 setup a web server to receive ESP-NOW packets from multiple boards (many-to-one configuration).

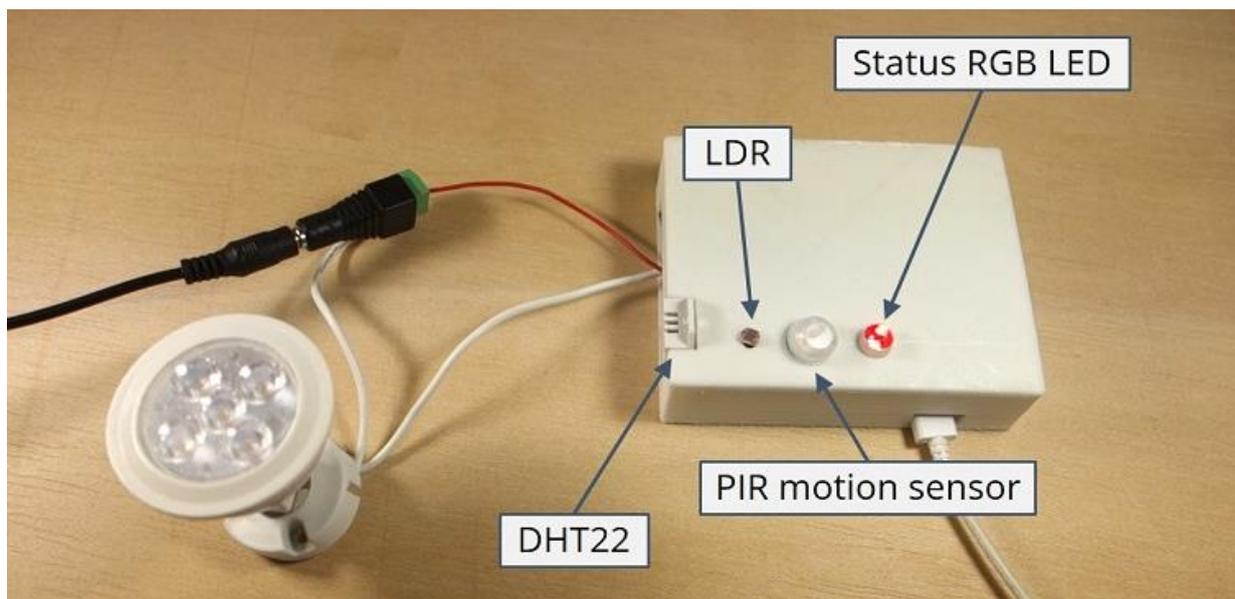
Additionally, you also used Server-Sent Events to automatically update the web page every time a new packet is received without refreshing the web page.

PROJECT 1

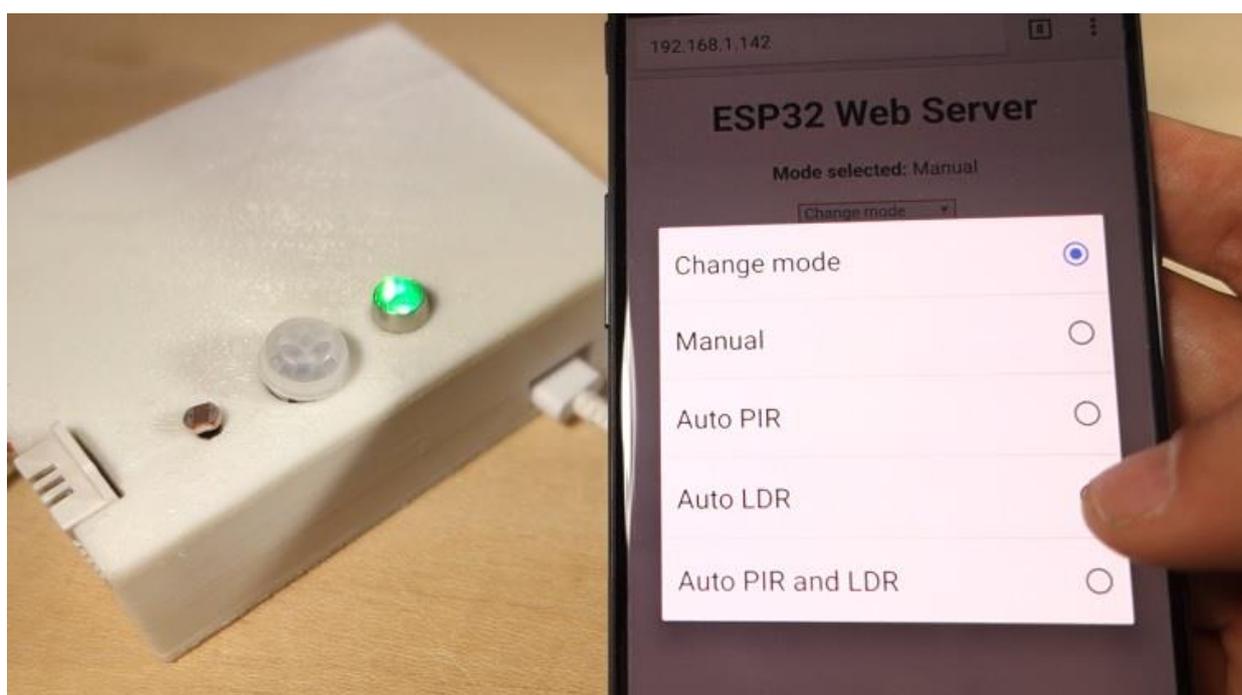
**ESP32 Wi-Fi Multisensor:
Temperature, Humidity, Motion,
Luminosity, and Relay Control**

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 in 2 separate 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.

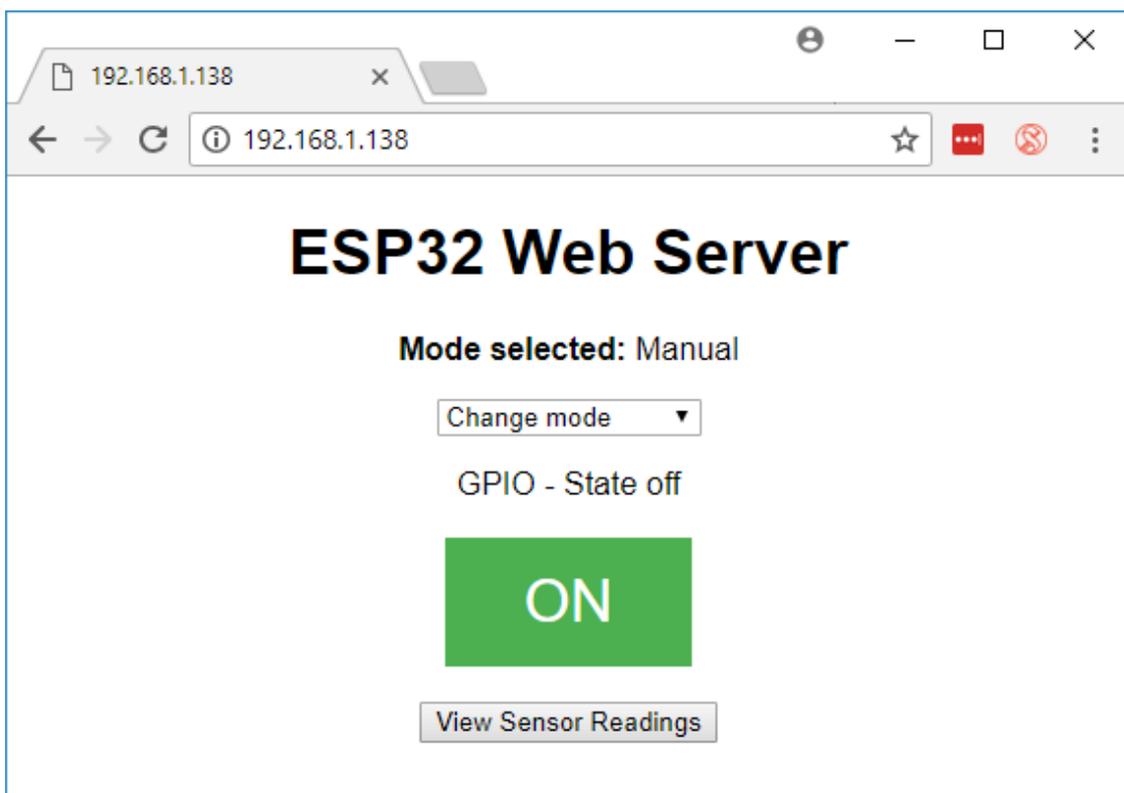
To better understand some of the concepts used throughout this project, first you may want to take a look at the following Units:

- ESP32 Web Server – Control Outputs
- ESP32 Web Server – Control Outputs (Relay)
- ESP32 with PIR Motion Sensor – Interrupts and Timers
- ESP32 Flash Memory – Store Permanent Data (Write and Read)

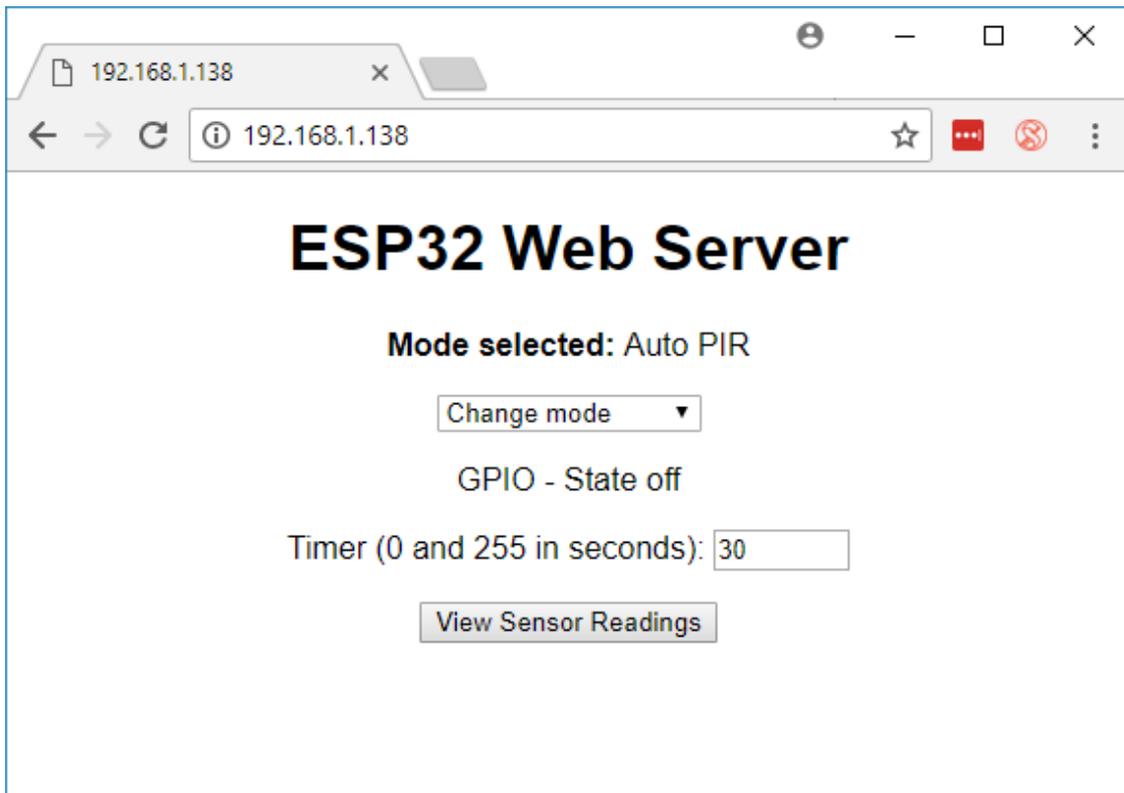
Project Overview

The web server for this project allows you to choose between 4 different modes to control the relay:

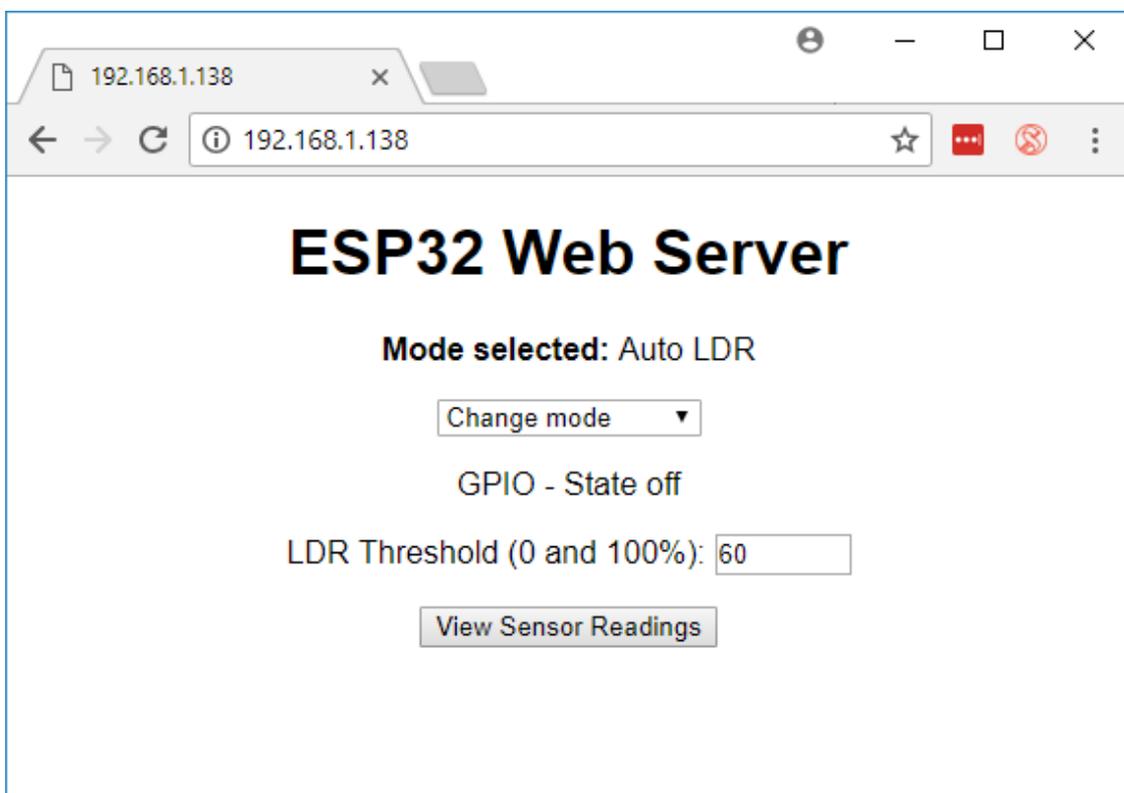
- **Manual** (mode 0): in which you have a button to turn the relay on and off.



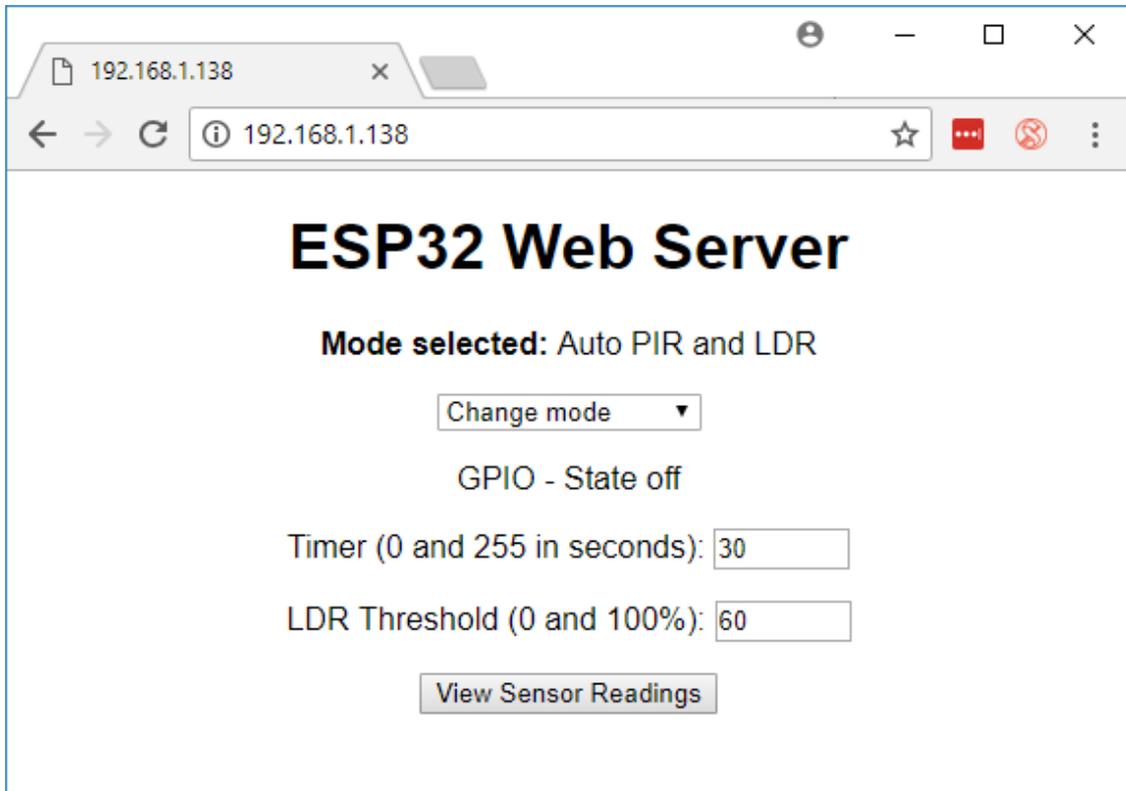
- **Auto PIR** (mode 1): turns the relay on when motion is detected. In this mode there is a field in which you can set the number of seconds the output will be on after motion is detected.



- **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%.

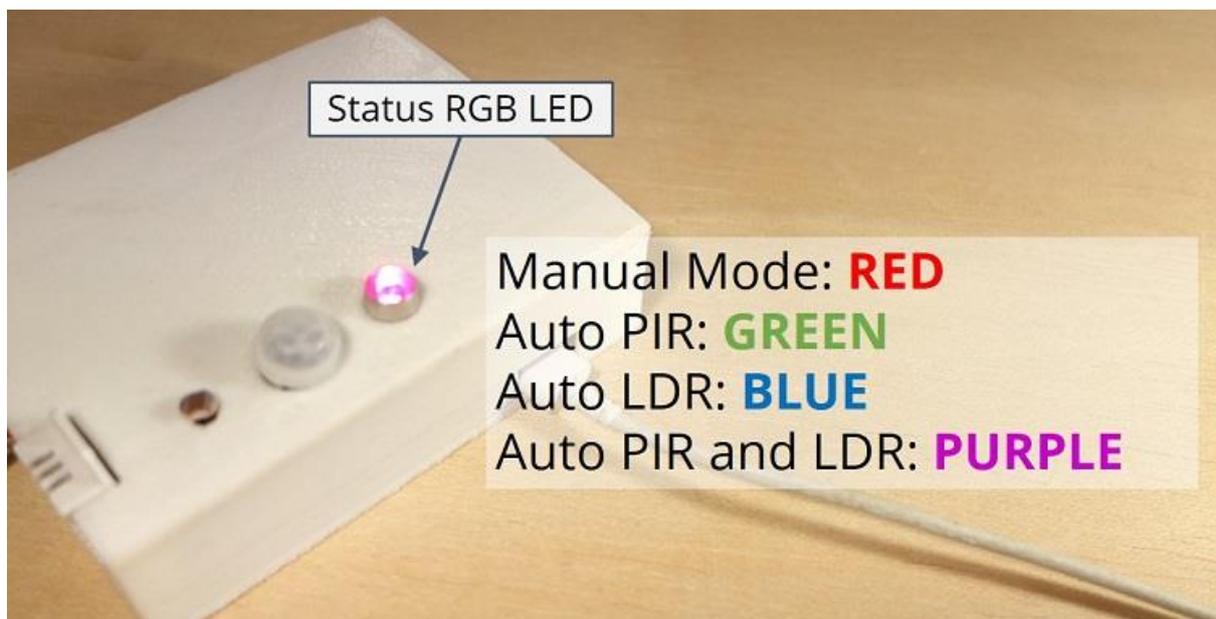


- **Auto PIR and LDR** (mode 3): this mode combines the PIR motion sensor and the LDR. When this mode is selected, the relay turns on when the PIR sensor detects motion and if the luminosity value is below the threshold. In this mode you can set the timer and the LDR threshold value.



Status indicator

The device contains an RGB LED that changes color accordingly to the selected mode:



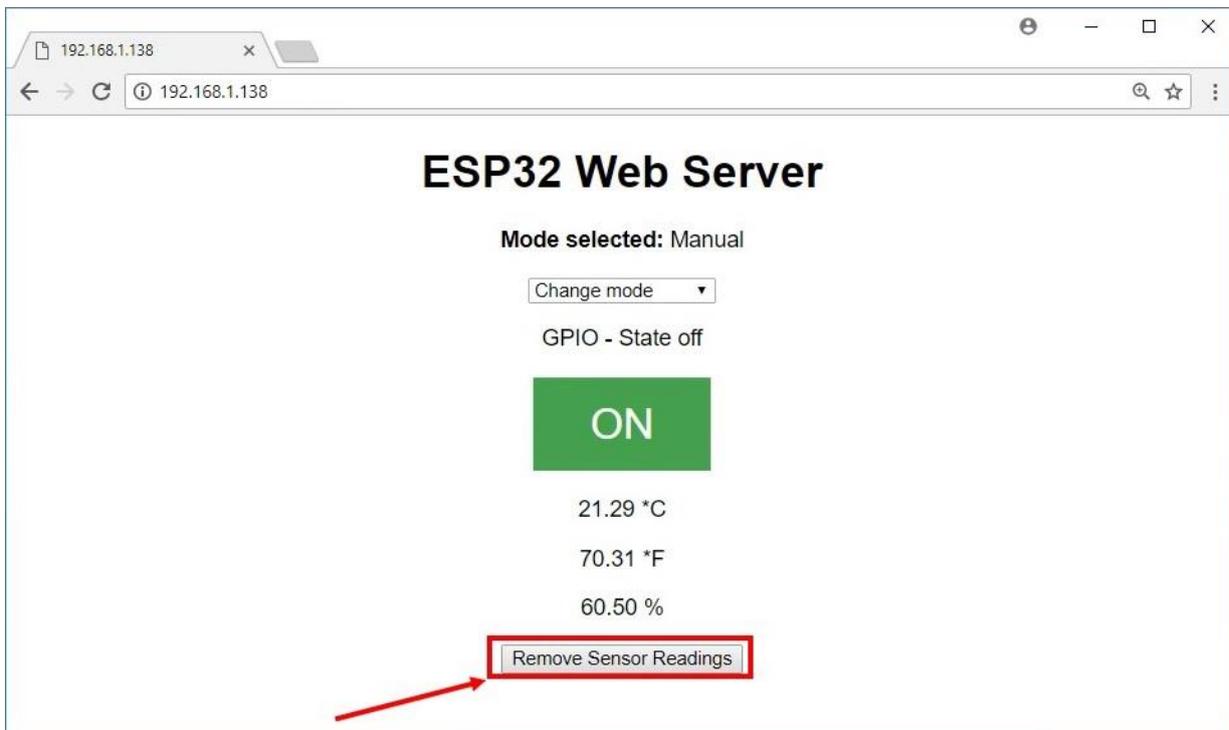
- Manual: red
- Auto PIR: green
- Auto LDR: blue
- Auto PIR and LDR: purple

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 "Hide Sensor Readings" button to hide the readings to optimize the web server performance.



In every mode there's a label that shows the selected mode, as well as 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.

Note: if you want to learn how to permanently save data on the ESP32 flash memory, read the following Unit: ESP32 Flash Memory – Store Permanent Data (Write and Read).

Building the Sensor Node

Now that you know what you're going to build, let's create the ESP32 Wi-Fi Multisensor.

Parts Required:

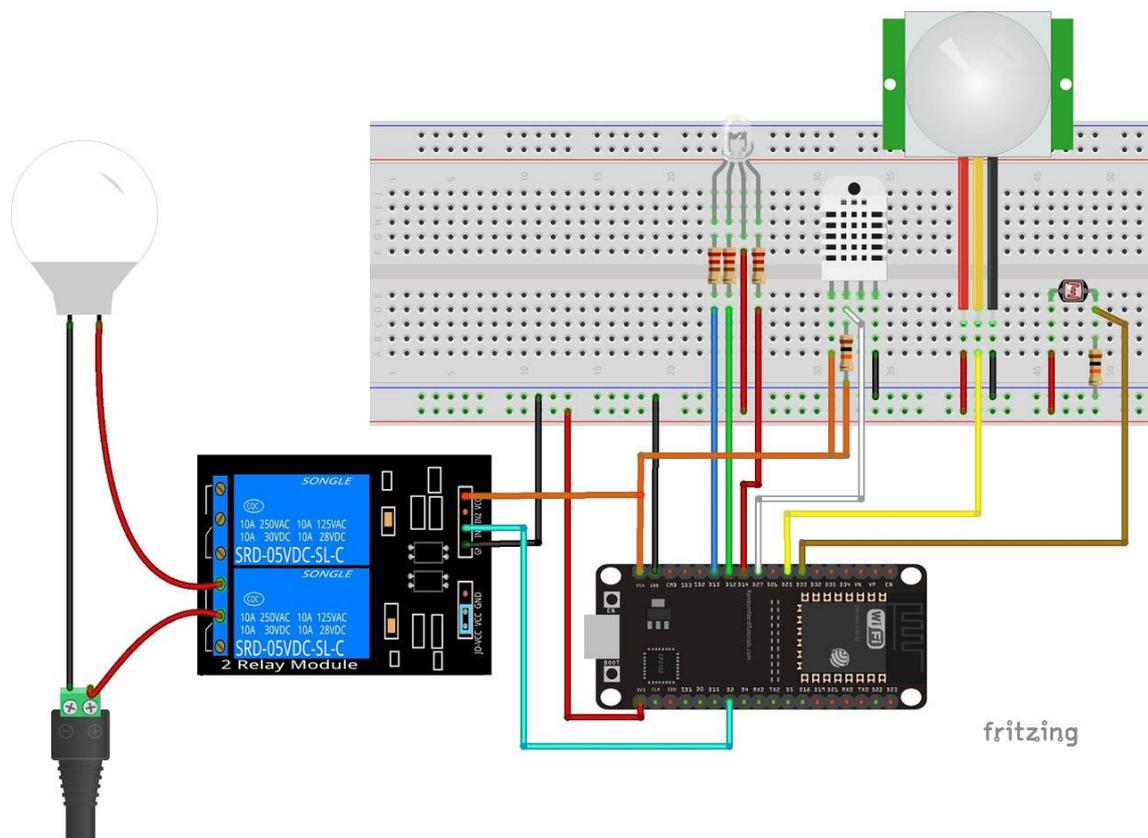
Here's a list of the parts required to build this project:

- [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](#)
- [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 needs.



(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.)

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

Installing the DHT Sensor library

- 1) [Click here to download the DHT sensor library](#). You should have a .zip folder in your Downloads
- 2) Unzip the .zip folder and you should get DHT-sensor-library-master folder
- 3) Rename your folder from ~~DHT-sensor-library-master~~ to DHT
- 4) Move the DHT folder to your Arduino IDE installation libraries folder
- 5) Then re-open your Arduino IDE

Installing the Adafruit_Sensor library

- 1) [Click here to download the Adafruit_Sensor library](#). You should have a .zip folder in your Downloads folder
- 2) Unzip the .zip folder and you should get Adafruit_Sensor-master folder
- 3) Rename your folder from ~~Adafruit_Sensor-master~~ to Adafruit_Sensor
- 4) Move the Adafruit_Sensor folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Uploading the Sensor Node Code

After installing the required libraries, copy the following code to your Arduino IDE.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_WiFi_Multisensor/ESP32_WiFi_Multisensor.ino

```
// Load libraries
#include <WiFi.h>
#include <EEPROM.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];

// EEPROM size
```

```

// Address 0: Last output state (0 = off or 1 = on)
// Address 1: Selected mode (0 = Manual, 1 = Auto PIR,
// 2 = Auto LDR, or 3 = Auto PIR and LDR)
// Address 2: Timer (time 0 to 255 seconds)
// Address 3: LDR threshold value (luminosity in percentage 0 to 100%)
#define EEPROM_SIZE 4

// 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...");
if(!EEPROM.begin(EEPROM_SIZE)) {
  Serial.println("failed to initialise EEPROM");
  delay(1000);
}

// Uncomment the next lines to test the values stored in the flash memory
Serial.println(" bytes read from Flash . Values are:");
for(int i = 0; i < EEPROM_SIZE; i++) {
  Serial.print(byte(EEPROM.read(i)));
  Serial.print(" ");
}

// 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(!EEPROM.read(0)) {
  outputState = "off";
  digitalWrite(output, HIGH);
}
else {
  outputState = "on";
  digitalWrite(output, LOW);
}
selectedMode = EEPROM.read(1);
timer = EEPROM.read(2);
ldrThreshold = EEPROM.read(3);
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
    while (client.connected() && currentTime - previousTime <= timeoutTime)
    { // loop while the client's connected
      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();
EEPROM.write(1, selectedMode);
EEPROM.commit();
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();
    EEPROM.write(2, timer);
    EEPROM.commit();
    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();
    EEPROM.write(3, ldrThreshold);
    EEPROM.commit();
    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") {

```



```

        // You can delete the following Serial.prints, it s just for
        debugging purposes
        /*Serial.print("Humidity:          ");          Serial.print(h);
Serial.print(" %\t Temperature: ");
        Serial.print(t); Serial.print(" *C "); Serial.print(f);
        Serial.print(" *F\t Heat index: "); Serial.print(hic);
Serial.print(" *C ");
        Serial.print(hif);          Serial.print("          *F");
Serial.print("Humidity: ");
        Serial.print(h);  Serial.print(" %\t Temperature: ");
Serial.print(t);
        Serial.print(" *C "); Serial.print(f); Serial.print(" *F\t
Heat index: ");
        Serial.print(hic); Serial.print(" *C "); Serial.print(hif);
Serial.println(" *F");*/
    }
    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);
  EEPROM.write(0, 1);
  EEPROM.commit();
}
void outputOff() {
  Serial.println("GPIO off");
  outputState = "off";
  digitalWrite(output, HIGH);
  EEPROM.write(0, 0);
  EEPROM.commit();
}

```

This code is quite long to explain. You can simply replace the following two variables with your network credentials and the code will work straight away.

```

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

```

We explain how the code works in a separated Unit. So, read the next Unit to learn how the code works.

After adding 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.

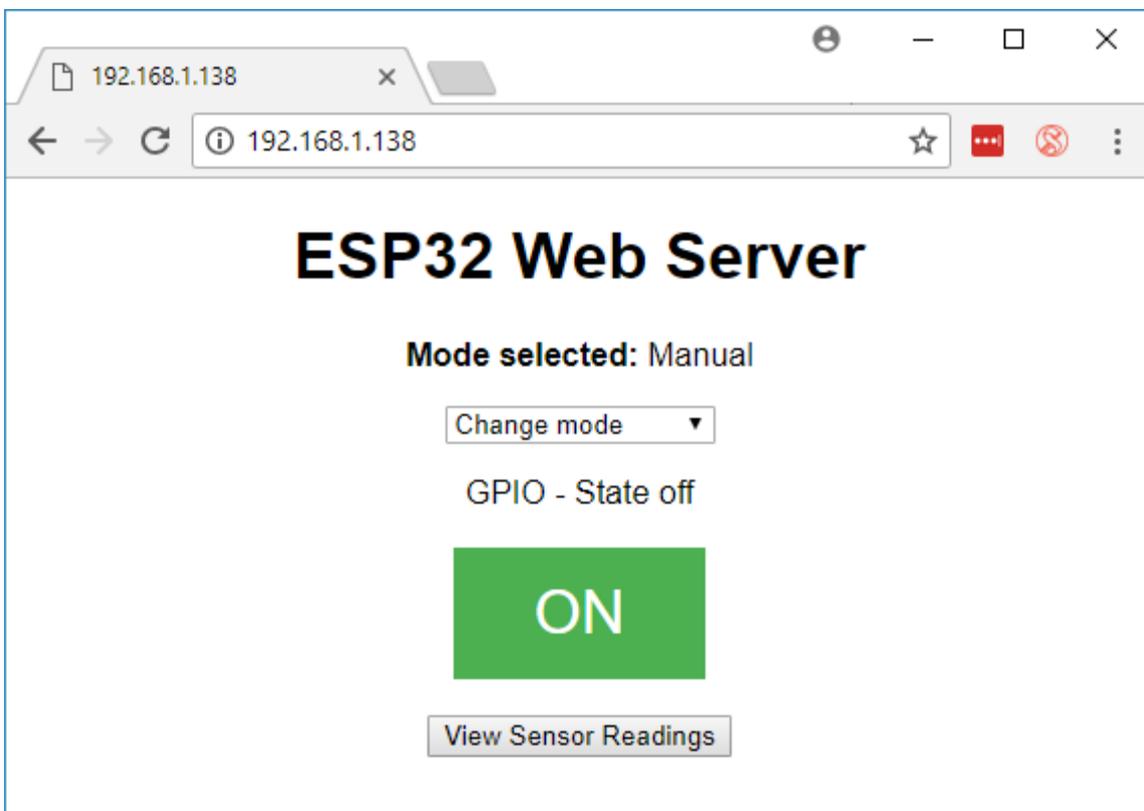


Press the ESP32 enable button to print the ESP32 IP address.

```
load:0x40078000, len:11392
entry 0x40078a9c
start...
  bytes read from Flash . Values are:
0 3 30 60 GPIO off
Connecting to MEO-620B4B
MOTION DETECTED!!!
..
WiFi connected.
IP address:
192.168.1.138
GPIO off
```

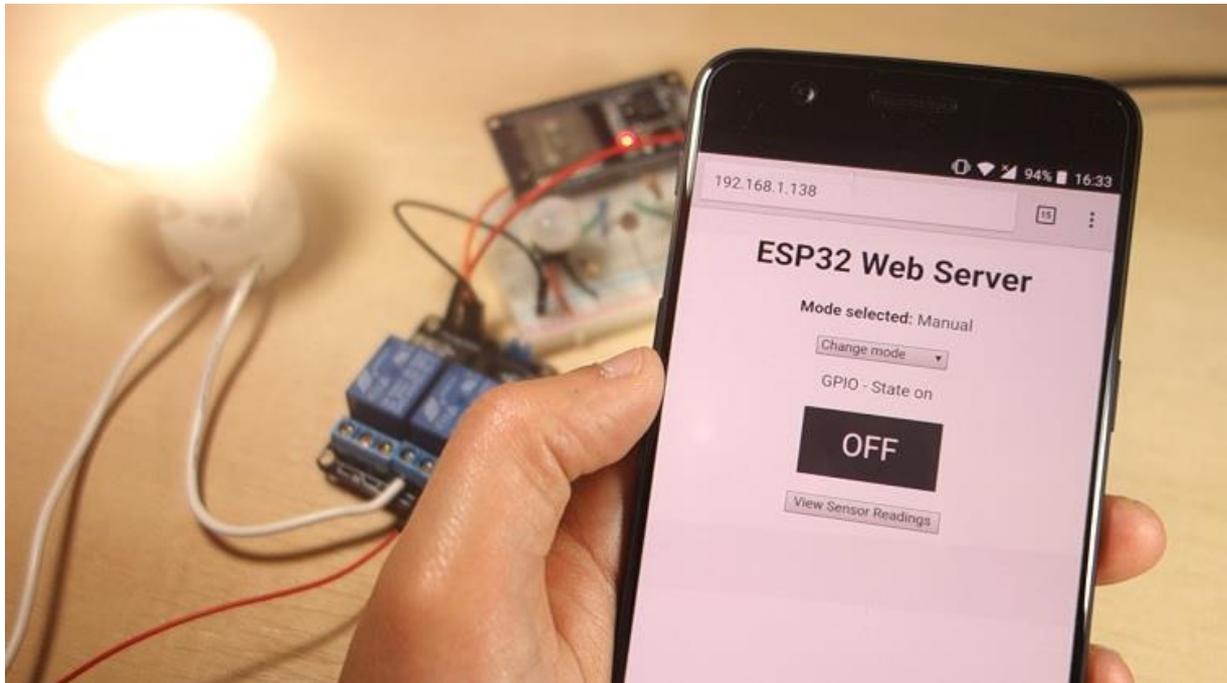
Autoscroll | No line ending | 115200 baud | Clear output

Open your browser and type the ESP32 IP address. The following page should load.

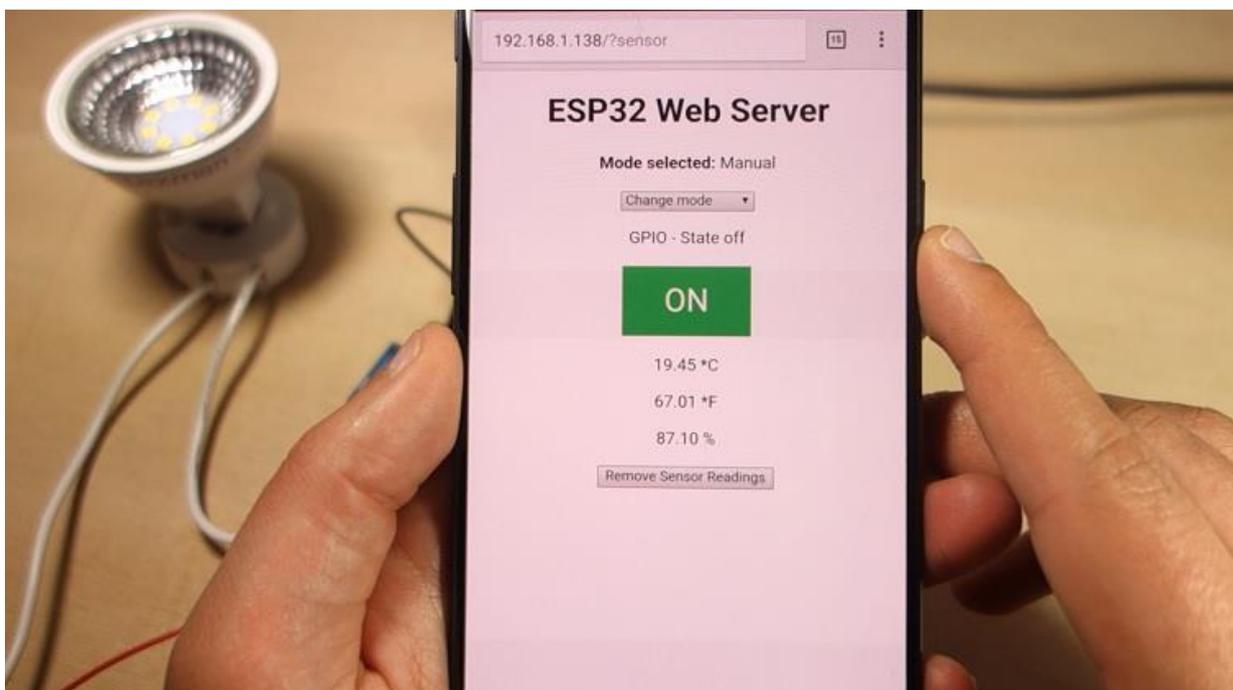


Now, select each mode, try to set different settings to check if everything is working properly.

For example, select Manual mode and turn the lamp on and off.



Click the **“View Sensor Readings”** button to request the latest sensor readings.

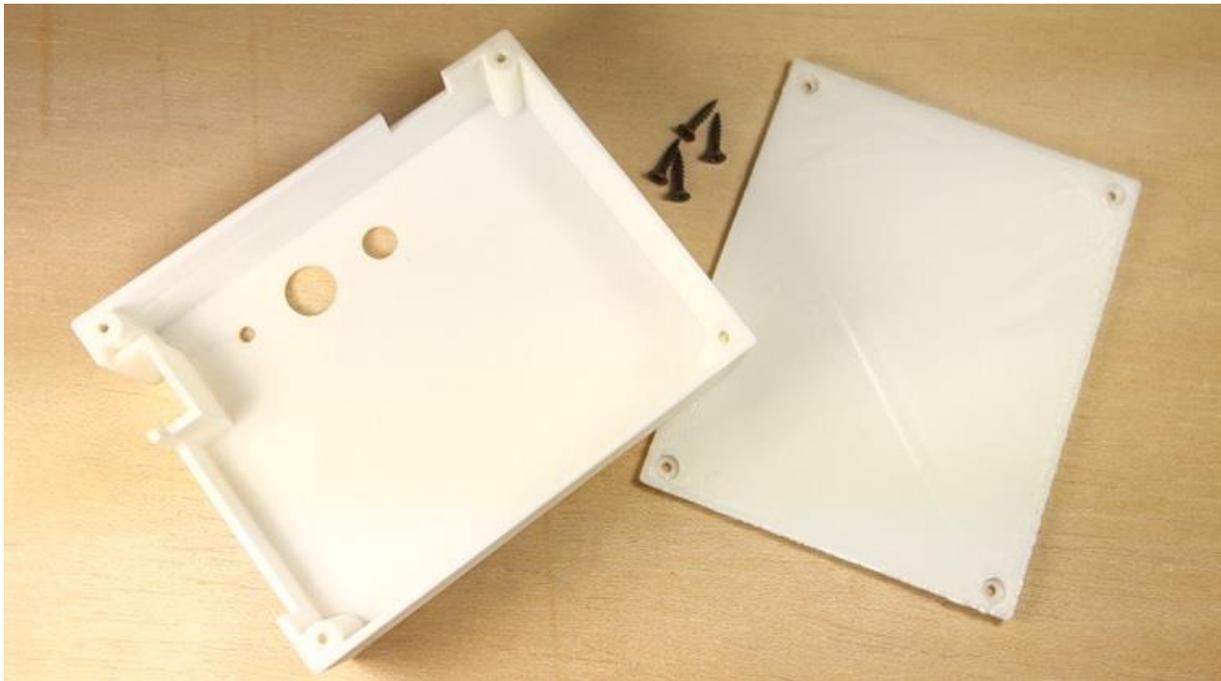


Select other modes and see the RGB LED color changing. Configure the multisensor with your own settings to check how it works.

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](#)).

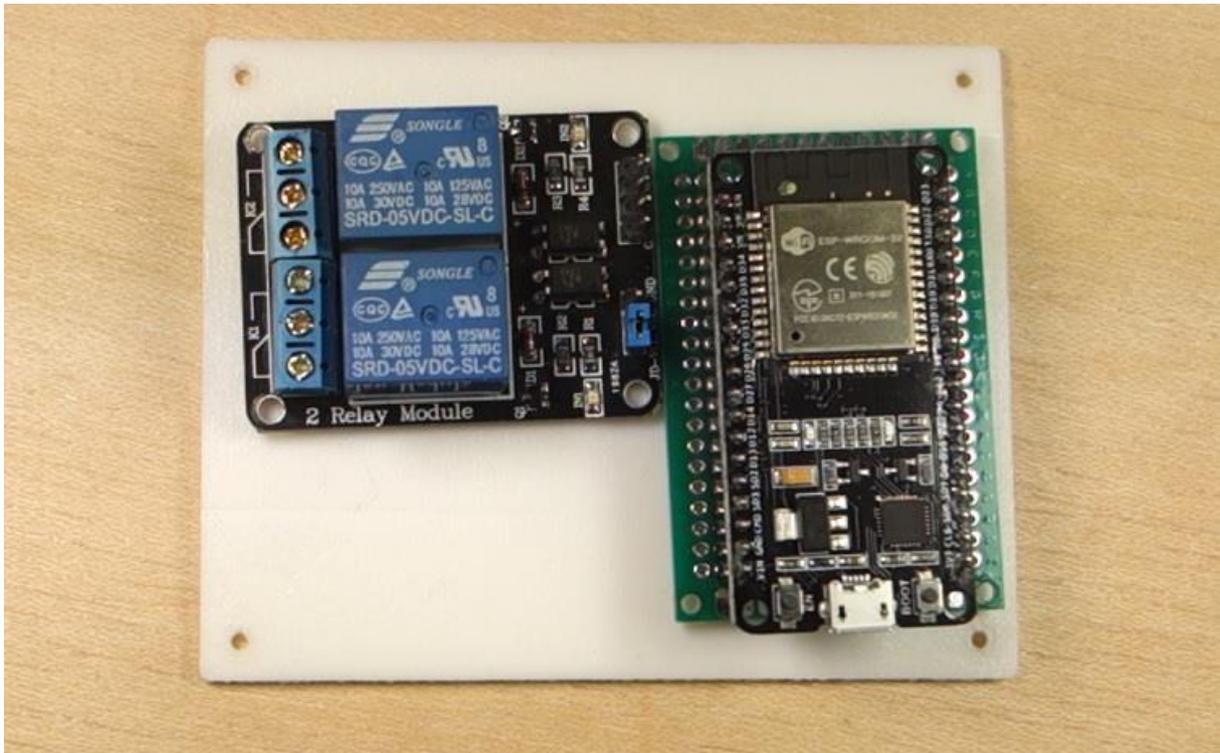
The enclosure consists of two pieces: the bottom and the lid. The lid has three holes on top.



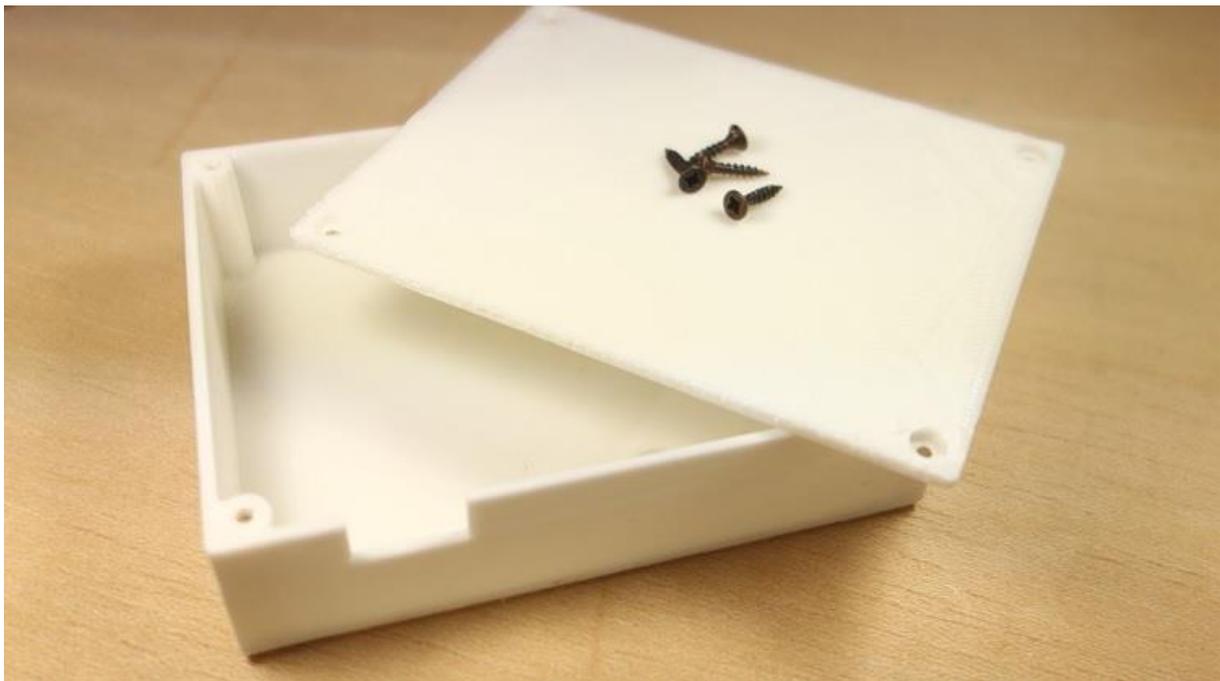
One to place the LDR, another to place the mini PIR motion sensor and another for the RGB LED holder. There's a slot at the side to place the DHT. There's a space for the relay wires and another for the USB cable to go through and power the ESP32.



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.



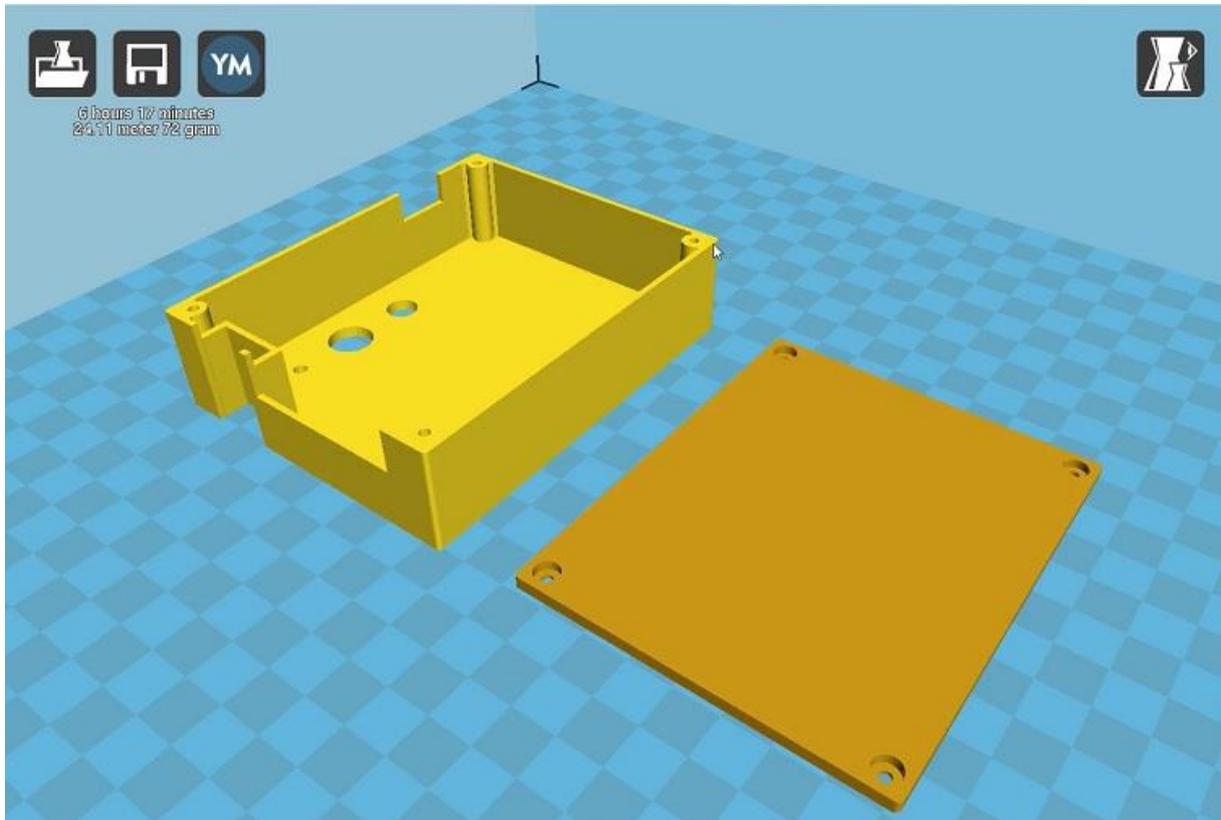
Finally, there are 4 holes to fix the lid with the bottom using 4 screws.



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](#)

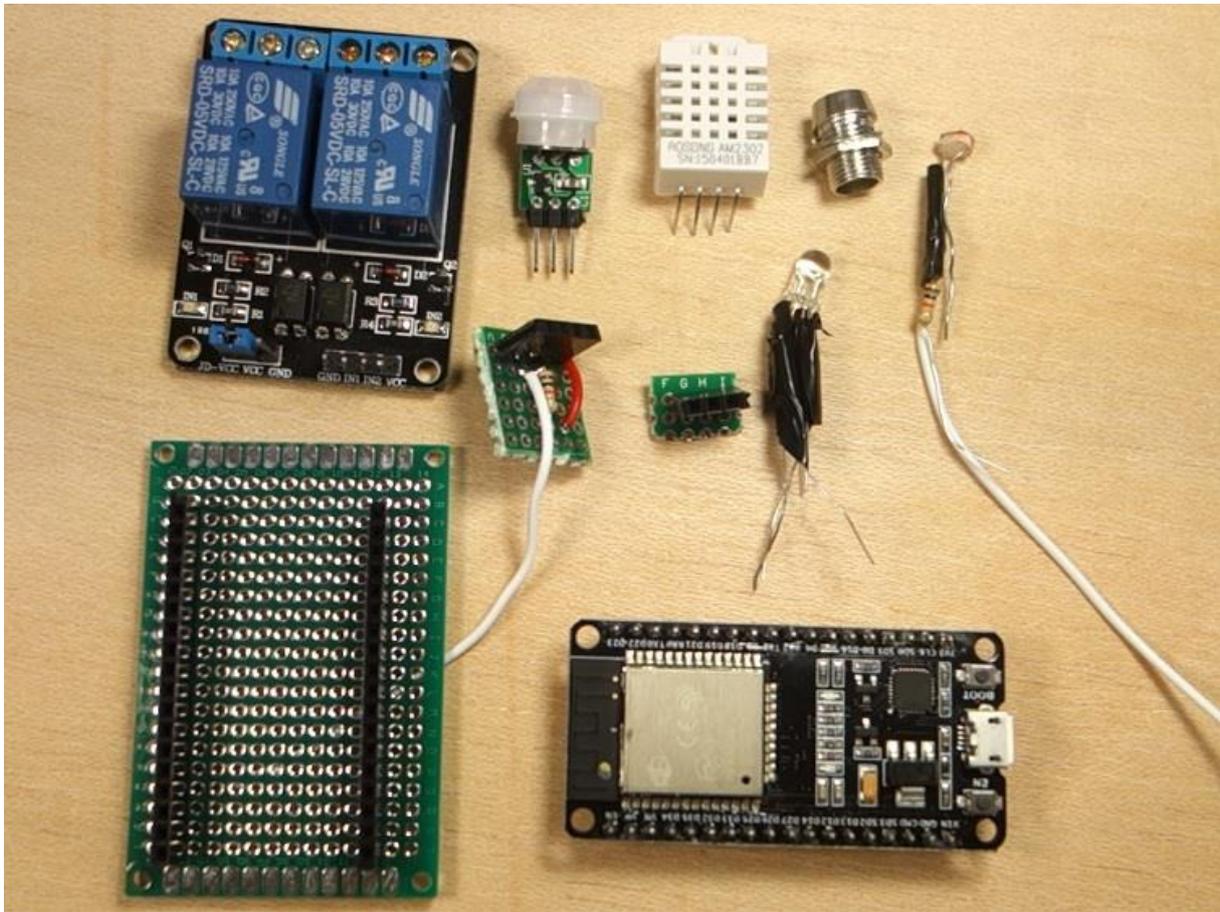
Note: we advise you to increase the size of the enclosure, as it will be easier to place all the components inside.



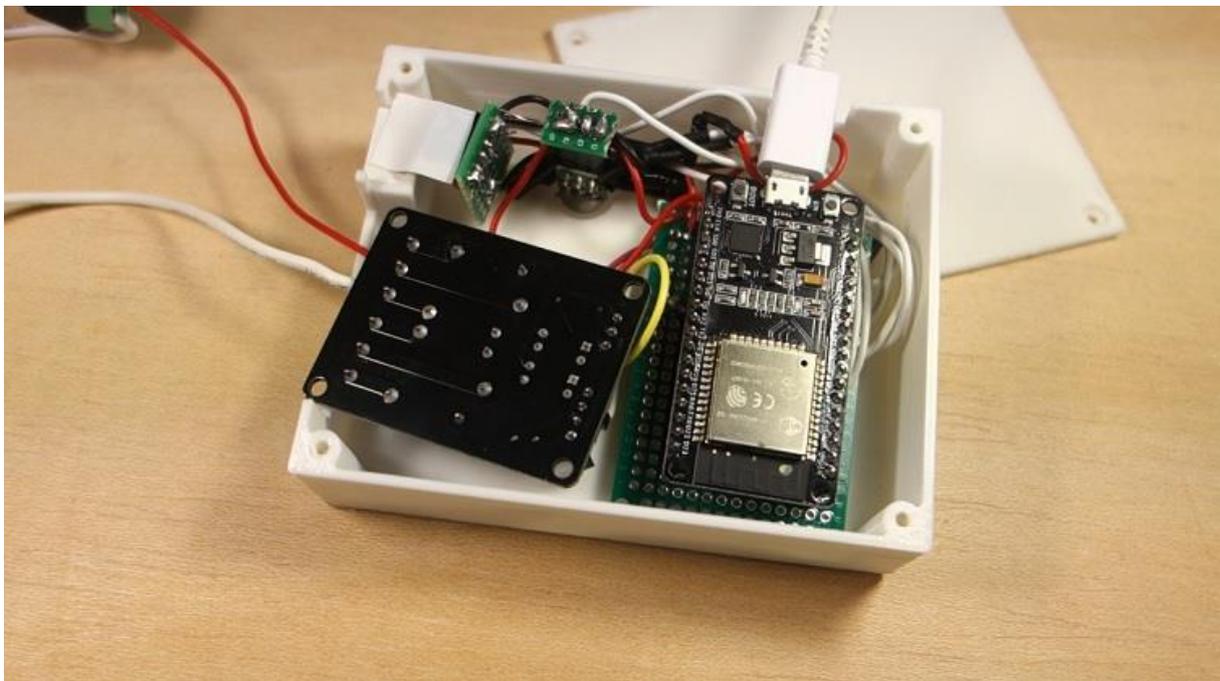
Alternatively, you can also use any other plastic enclosure and make some holes to place all the circuitry.



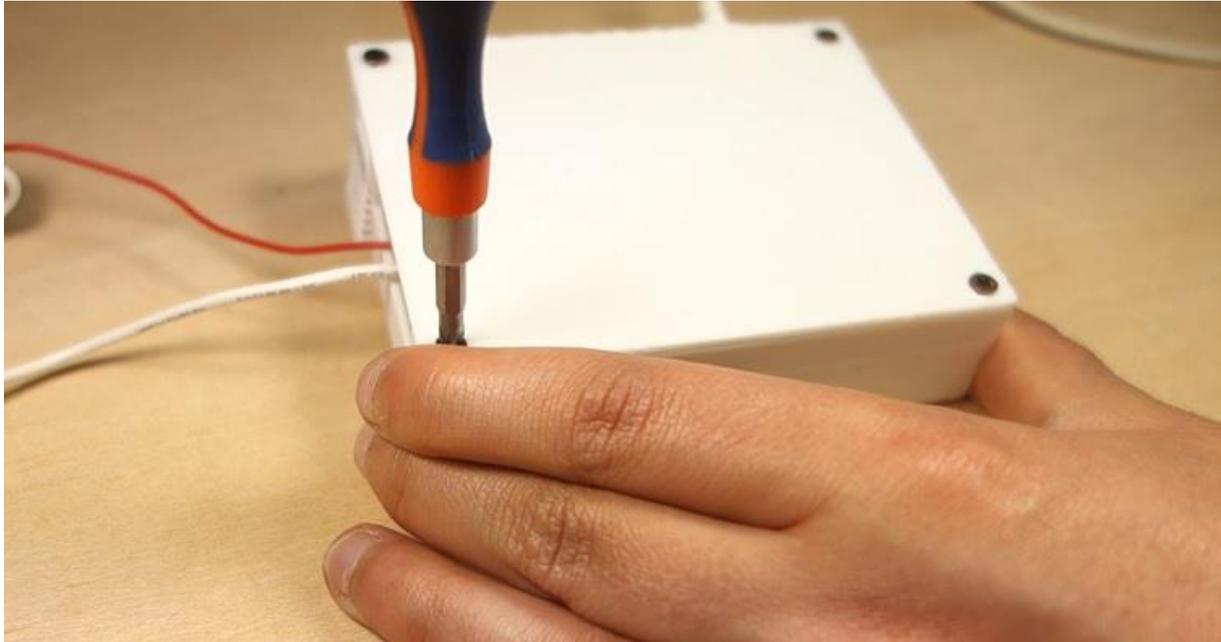
We've built a semi-permanent circuit using small pieces of prototyping board to place the ESP32, the PIR motion sensor, and the DHT. We've also soldered wires directly to the cheaper components like the LDR and the RGB LED.



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 then place it inside a different enclosure.

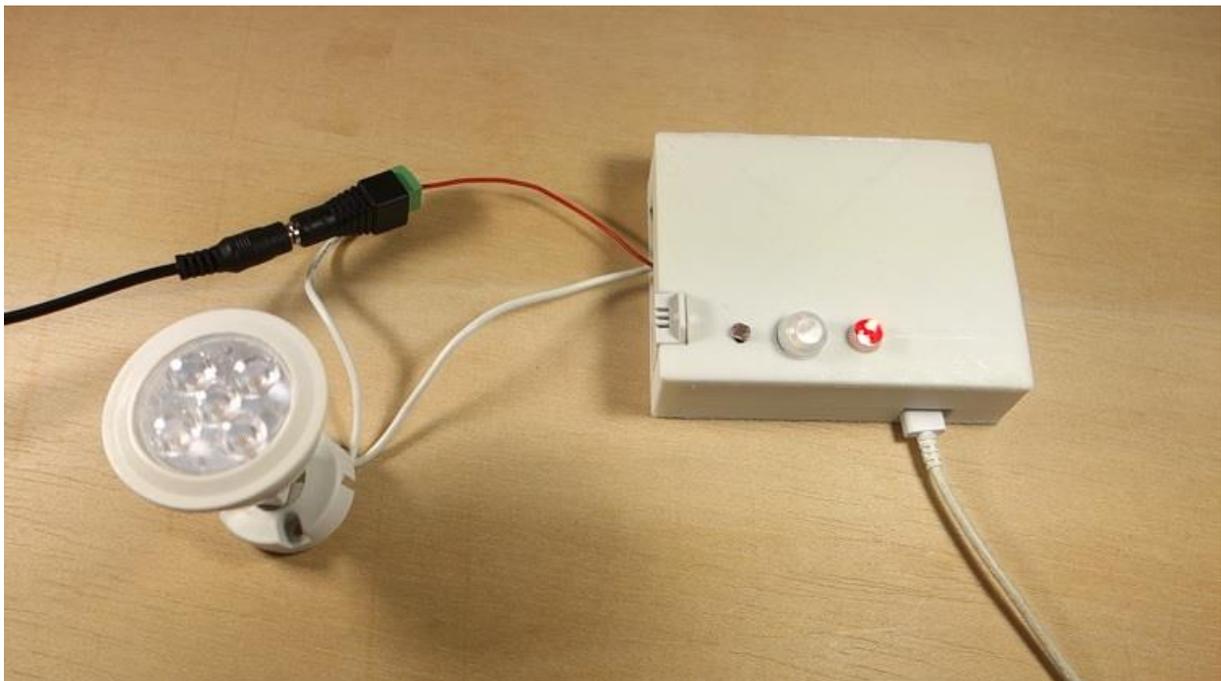


After powering the ESP32 using a USB cable, and connecting the relay to the lamp, we can close the enclosure using 4 screws.

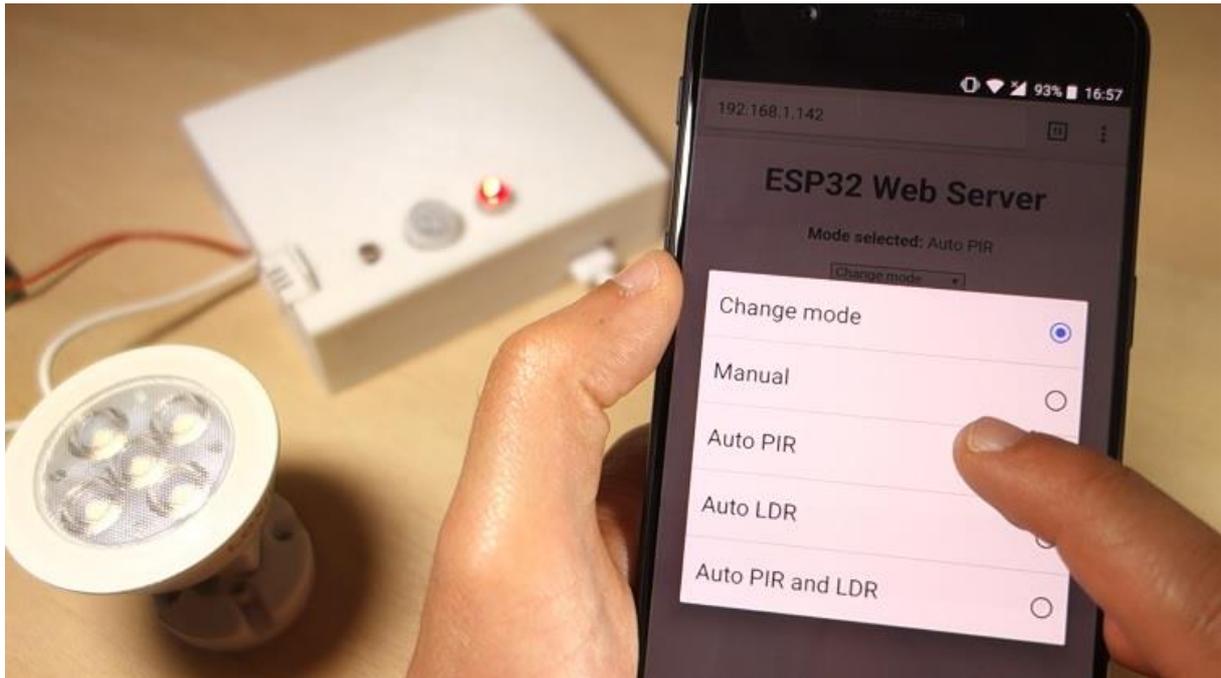


Demonstration

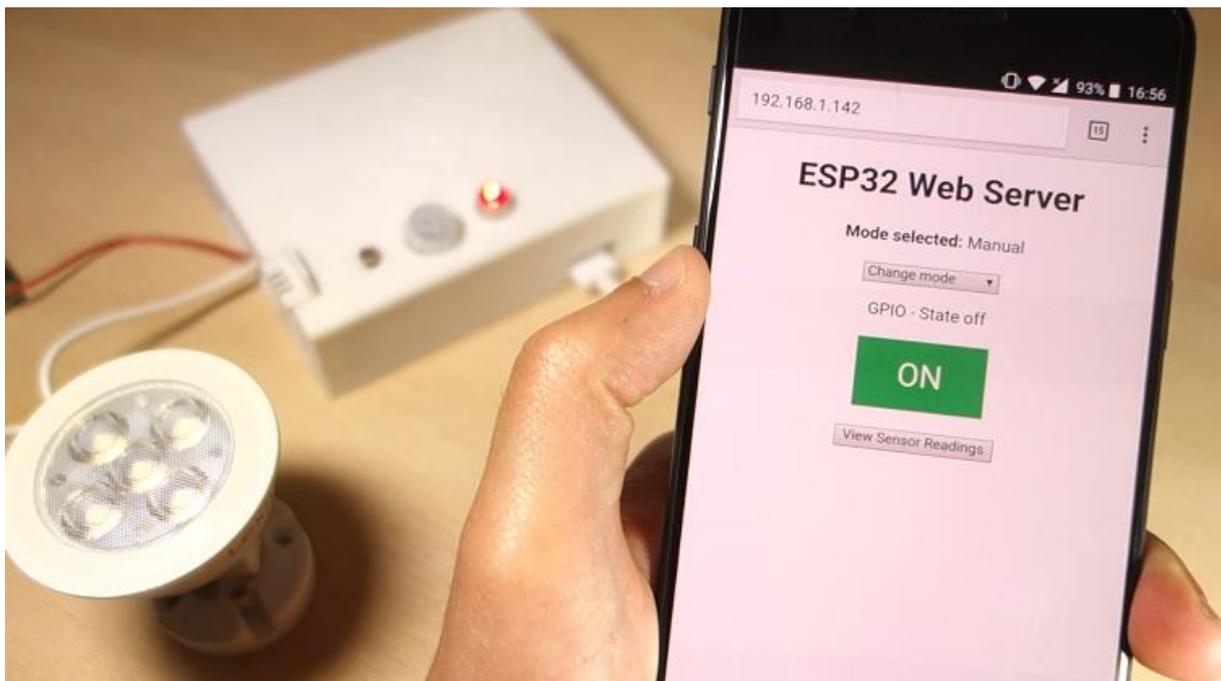
The following figure shows how the Wi-Fi Multisensor looks like.



Now, you can access your web server to control the output in different modes and configure the modes with your own settings.



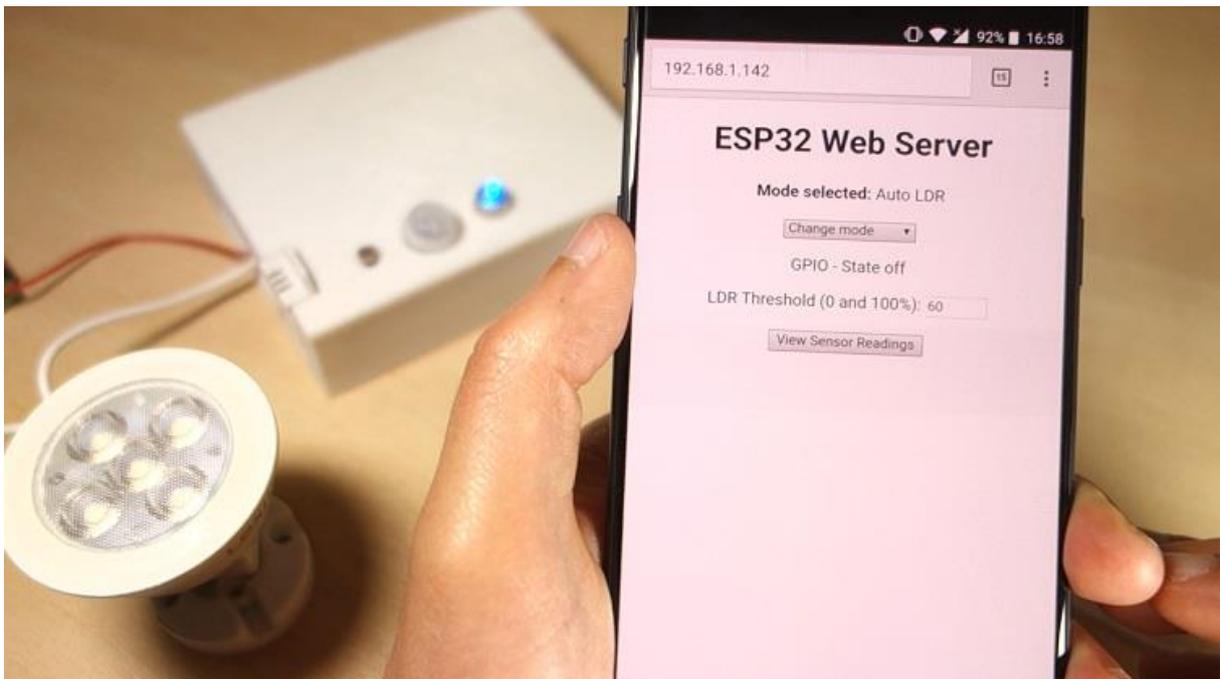
On manual mode, press the buttons to turn the lamp on and off.



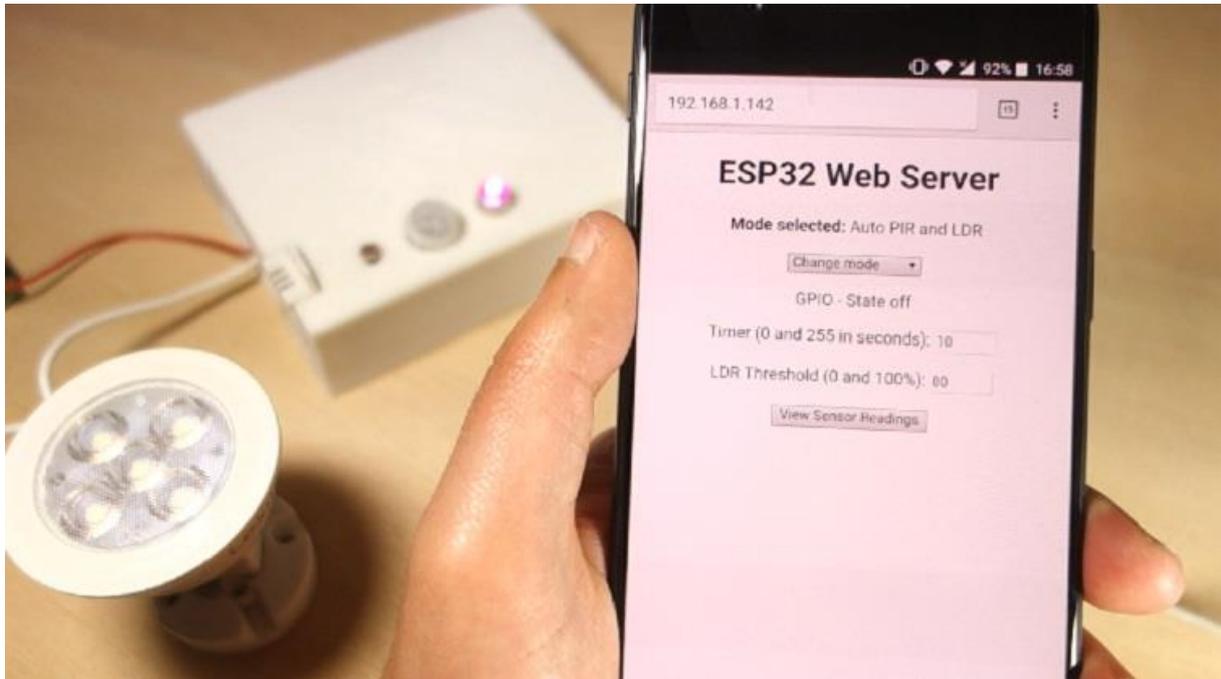
Select the Auto PIR mode. In this mode the lamp turns on for the number of seconds you set, when motion is detected.



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.

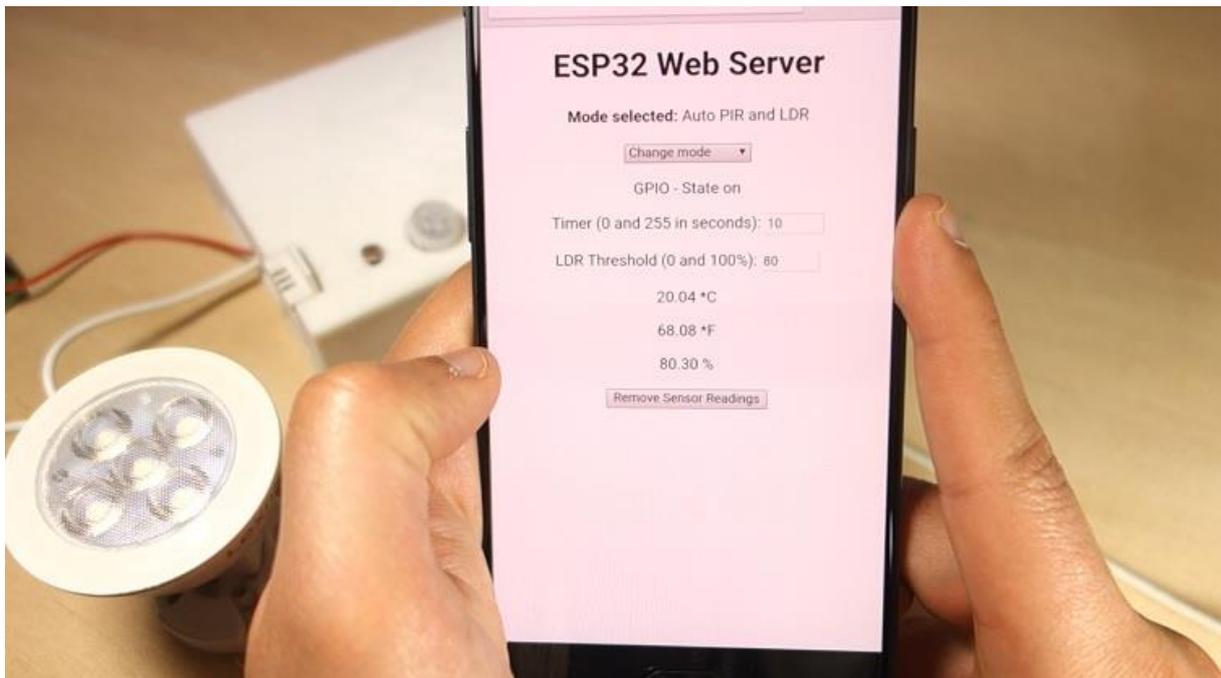


In Auto PIR and LDR mode, I can set the timer and the LDR threshold.



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?

This Unit explains the code used in the ESP32 Wi-Fi Multisensor project. This code is easier to understand if you've followed previous Units:

- ESP32 Web Server – Control Outputs
- ESP32 Web Server – Control Outputs (Relay)
- ESP32 with PIR Motion Sensor – Interrupts and Timers
- ESP32 Flash Memory – Store Permanent Data (Write and Read)

Code

This is the code used in the ESP32 Wi-Fi Multisensor project.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_WiFi_Multisensor/ESP32_WiFi_Multisensor.ino

```
// Load libraries
#include <WiFi.h>
#include <EEPROM.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];

// EEPROM size
// Address 0: Last output state (0 = off or 1 = on)
// Address 1: Selected mode (0 = Manual, 1 = Auto PIR,
// 2 = Auto LDR, or 3 = Auto PIR and LDR)
// Address 2: Timer (time 0 to 255 seconds)
// Address 3: LDR threshold value (luminosity in percentage 0 to 100%)
#define EEPROM_SIZE 4
```

```

// 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);

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...");
  if(!EEPROM.begin(EEPROM_SIZE)) {
    Serial.println("failed to initialise EEPROM");
    delay(1000);
  }

  // Uncomment the next lines to test the values stored in the flash memory
  Serial.println(" bytes read from Flash . Values are:");
  for(int i = 0; i < EEPROM_SIZE; i++) {
    Serial.print(byte(EEPROM.read(i)));
    Serial.print(" ");
  }

  // 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 and store the values in auxiliary variables
  // Set output to last state (saved in the flash memory)
  if(!EEPROM.read(0)) {
    outputState = "off";
  }
}

```

```

    digitalWrite(output, HIGH);
}
else {
    outputState = "on";
    digitalWrite(output, LOW);
}
selectedMode = EEPROM.read(1);
timer = EEPROM.read(2);
ldrThreshold = EEPROM.read(3);
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,
        Serial.println("New Client."); // print a message out in the serial port
        String currentLine = ""; // make a String to hold incoming data
        while (client.connected()) { // loop while the client's connected
            if (client.available()) { // if there's bytes to read
                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, there's 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
                        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
                        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();
        EEPROM.write(1, selectedMode);
        EEPROM.commit();
        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();
        EEPROM.write(2, timer);
        EEPROM.commit();
        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();
        EEPROM.write(3, ldrThreshold);
        EEPROM.commit();
        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(EEPROM.read(2)) + \"\"
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(EEPROM.read(3)) + "\"
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(EEPROM.read(2)) + "\"
onchange=\"setTimer(this.value)\" min=\"0\" max=\"255\"></p>");
        client.println("<p>LDR Threshold (0 and 100%): <input
type=\"number\" name=\"txt\" value=\"\" +
            String(EEPROM.read(3)) + "\"
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'
        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
            float hic = dht.computeHeatIndex(t, h, false);
            dtostrf(hic, 6, 2, celsiusTemp);
            float hif = dht.computeHeatIndex(f, h);
            dtostrf(hif, 6, 2, fahrenheitTemp);
            dtostrf(h, 6, 2, humidityTemp);
            // You can delete the following Serial.prints
            /*Serial.print("Humidity: "); Serial.print(h);
Serial.print(" %\t Temperature: ");
Serial.print(t); Serial.print(" *C "); Serial.print(f);
Serial.print(" *F\t Heat index: "); Serial.print(hic);
Serial.print(" *C ");
Serial.print(hif); Serial.print(" *F");
Serial.print("Humidity: ");
Serial.print(h); Serial.print(" %\t Temperature: ");
Serial.print(t);
Serial.print(" *C "); Serial.print(f); Serial.print(" *F\t
Heat index: ");
Serial.print(hic); Serial.print(" *C "); Serial.print(hif);
Serial.println(" *F");*/
            client.println("<p>");
            client.println(celsiusTemp);
            client.println("<*/p><p>");
            client.println(fahrenheitTemp);
            client.println("<*/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(); location.reload(true); } ");
    client.println("function setTimer(value) { var xhr = new
XMLHttpRequest();");
    client.println("xhr.open('GET', \"/?timer=\" + value + \"&\",
true);");
    client.println("xhr.send(); location.reload(true); } ");
    client.println("function setThreshold(value) { var xhr = new
XMLHttpRequest();");
    client.println("xhr.open('GET', \"/?ldrthreshold=\" + value +
\"&\", true);");
    client.println("xhr.send(); location.reload(true); } ");
    client.println("function outputOn() { var xhr = new
XMLHttpRequest();");
    client.println("xhr.open('GET', \"/?state=on\", true);");
    client.println("xhr.send(); location.reload(true); } ");
    client.println("function outputOff() { var xhr = new
XMLHttpRequest();");
    client.println("xhr.open('GET', \"/?state=off\", true);");
    client.println("xhr.send(); location.reload(true); } ");
    client.println("function updateSensorReadings() { var xhr = new
XMLHttpRequest();");
    client.println("xhr.open('GET', \"/?sensor\", true);");
    client.println("xhr.send(); location.reload(true);
}</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') {
    currentLine += c;
}
}
}
// 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);
    EEPROM.write(0, 1);
    EEPROM.commit();
}
void outputOff() {
    Serial.println("GPIO off");
    outputState = "off";
    digitalWrite(output, HIGH);
    EEPROM.write(0, 0);
    EEPROM.commit();
}
}

```

Including Libraries

You start by including the necessary libraries.

```

#include <WiFi.h>
#include <EEPROM.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.

```

//#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];
```

EEPROM

Next, we define the flash memory size we want to access.

```
#define EEPROM_SIZE 4
```

We'll need to save four values in the flash memory: the last output state on address 0, the selected mode on address 1, the timer value on address 2, and the LDR threshold value on address 3. So, we need 4 bytes in the flash memory.

- **Address 0:** Last output state (0 = off or 1 = on)
- **Address 1:** Selected mode (0 = Manual, 1 = Auto PIR, 2 = Auto LDR, or 3 = Auto PIR and LDR)
- **Address 2:** Timer (time 0 to 255 seconds)
- **Address 3:** LDR threshold value (luminosity in percentage 0 to 100%)

We recommend reading the following Unit to learn more about ESP32 flash memory: **ESP32 Flash Memory – Store Permanent Data (Write and Read)**.

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;
```

Note: Read the next Unit to learn more about interrupts and timers with the ESP32: **ESP32 with PIR Motion Sensor – Interrupts and Timers**.

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.

```
// 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);
```

To learn more about web servers take a look at the following Unit: **ESP32 Web Server – Control Outputs.**

setup()

In the `setup()`, start by initializing the DHT sensor.

```
dht.begin();
```

Initialize the Serial Port 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);
```

Flash memory

This part of the code initializes the flash memory with the EEPROM size defined earlier, and reads the bytes stored.

```
Serial.println("start...");
if(!EEPROM.begin(EEPROM_SIZE)) {
  Serial.println("failed to initialise EEPROM");
  delay(1000);
}

Serial.println(" bytes read from Flash . Values are:");
for(int i = 0; i < EEPROM_SIZE; i++) {
  Serial.print(byte(EEPROM.read(i)));
  Serial.print(" ");
}
}
```

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 flash memory

Read the flash memory and set the output to the last state saved. The output state is saved on position 0, so use `EEPROM.read(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(!EEPROM.read(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 = EEPROM.read(1);
timer = EEPROM.read(2);
ldrThreshold = EEPROM.read(3);
```

Then, we call the `configureMode()` function to assign the right values for each mode.

```
configureMode();
```

configureMode()

Let's take a look on 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.

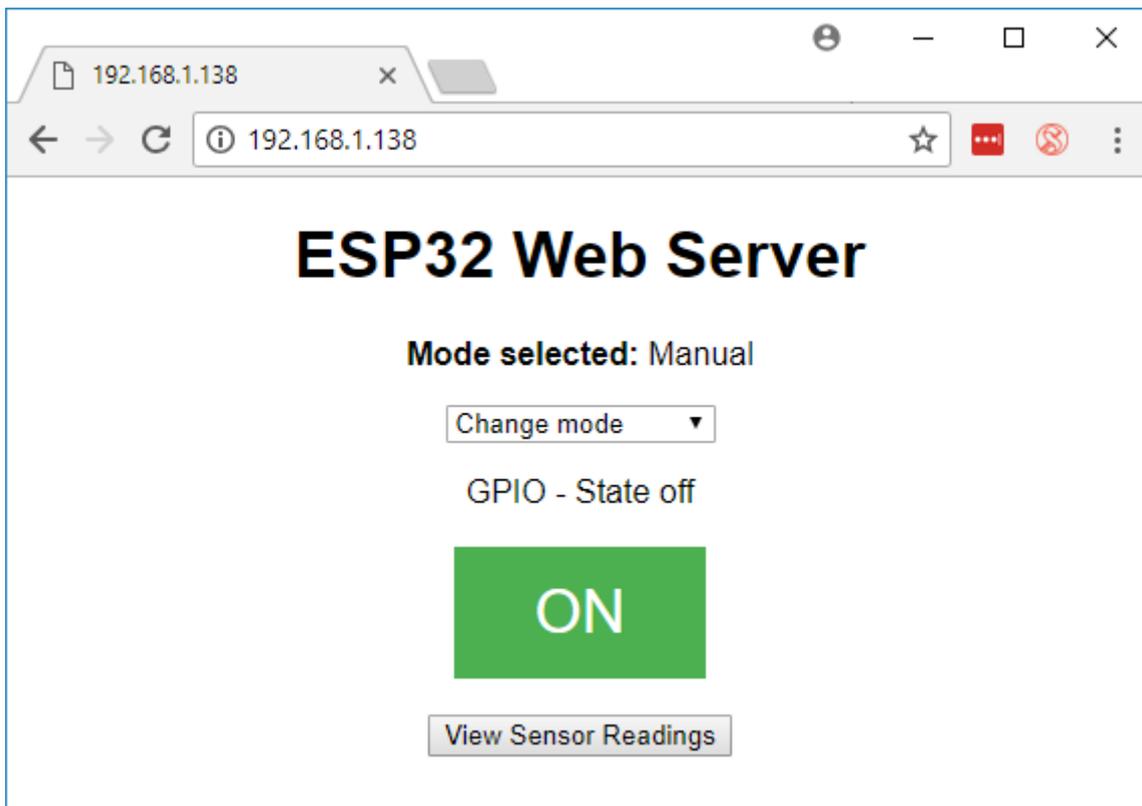
```
// Connect to Wi-Fi network with SSID and password
Serial.print("Connecting to ");
Serial.println(ssid);
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
}
// Print local IP address and start web server
Serial.println("");
Serial.println("WiFi 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 more relevant for this project. This part of the code is easier to understand if we explain what's happening with a live demonstration.

Note: this is Unit is a transcript from the video. To better understand the code, we recommend watching the video.

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 the 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();
  EEPROM.write(1, selectedMode);
  EEPROM.commit();
  configureMode();
}
```

It saves the selected mode in the `selectedMode` variable and stores it in the flash memory with `EEPROM.write(1, 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.

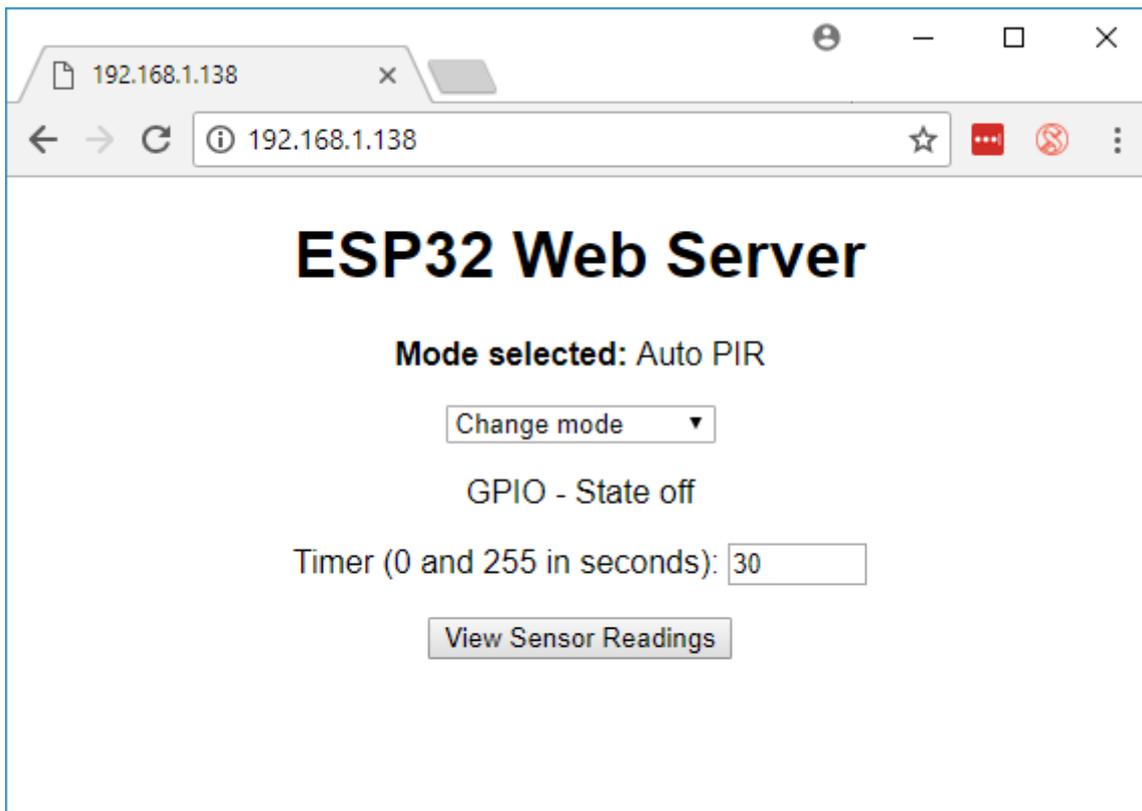
```
if(selectedMode == 0) {
  if(outputState == "off") {
    client.println("<p><button class=\"button\" onclick=\"outputOn()\">ON</button></p>");
  }
  else {
    client.println("<p><button class=\"button button2\" onclick=\"outputOff()\">OFF</button></p>");
  }
}
```

When you click the on and off buttons, in the background, the following code runs and one of these two `else if` statements actually turn the output on or 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.

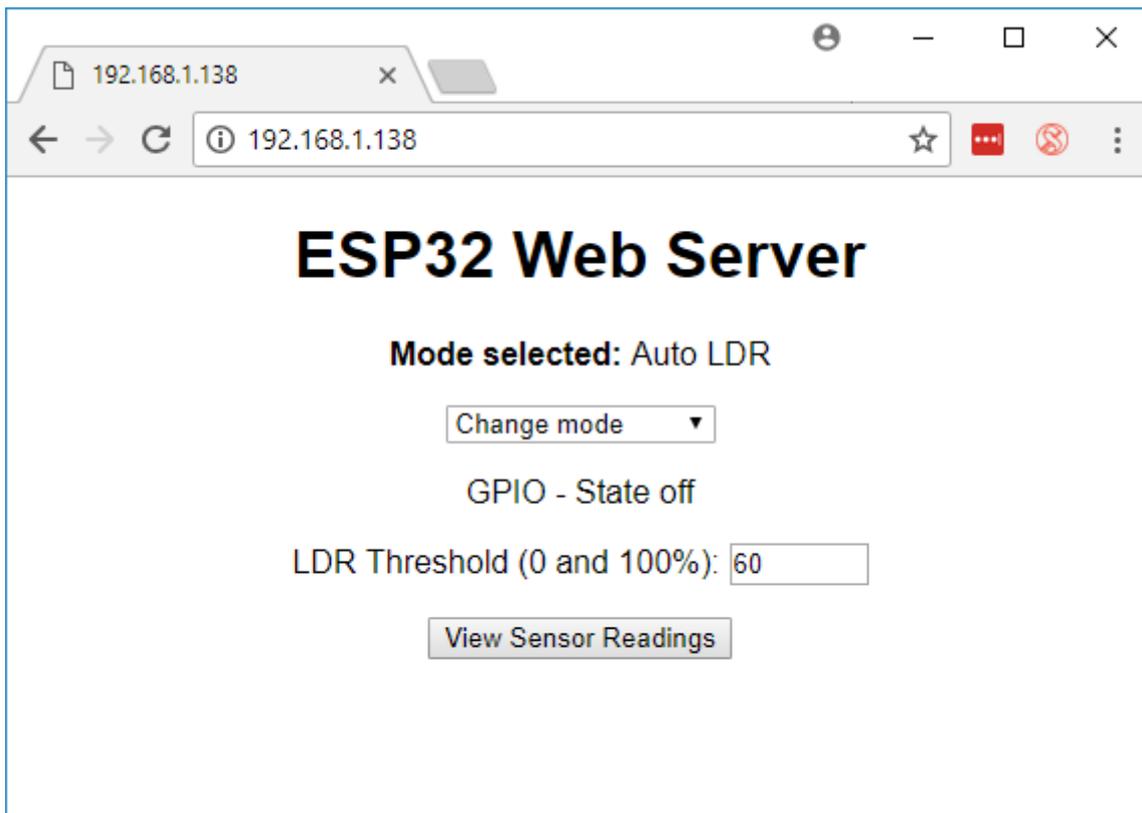
```
// Set timer value
else if(header.indexOf("GET /?timer=") >= 0) {
  pos1 = header.indexOf('=');
  pos2 = header.indexOf('&');
  valueString = header.substring(pos1+1, pos2);
  timer = valueString.toInt();
  EEPROM.write(2, timer);
  EEPROM.commit();
  Serial.println(valueString);
}
```

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(EEPROM.read(2)) + \"\"
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:

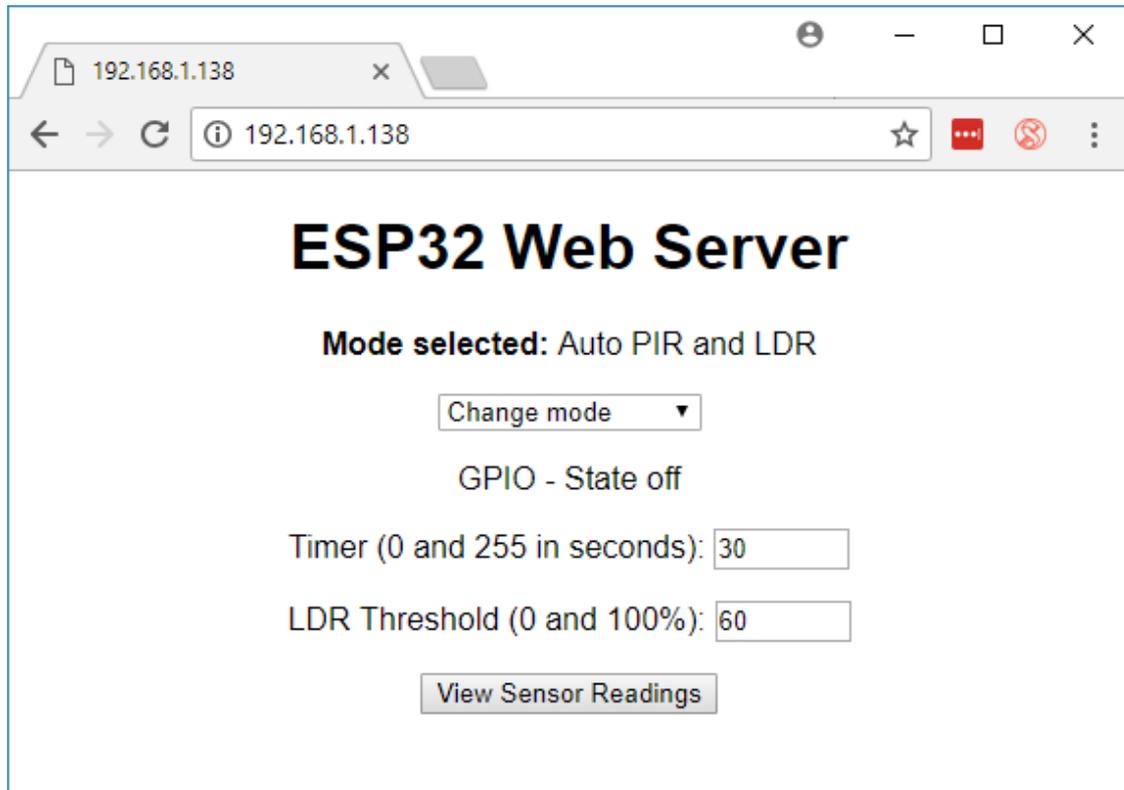
```
// Set LDR Threshold value
else if(header.indexOf("GET /?ldrthreshold=") >= 0) {
  pos1 = header.indexOf('=');
  pos2 = header.indexOf('&');
  valueString = header.substring(pos1+1, pos2);
  ldrThreshold = valueString.toInt();
  EEPROM.write(3, ldrThreshold);
  EEPROM.commit();
  Serial.println(valueString);
}
```

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%): <input type=\"number\"
name=\"txt\" value=\"" + String(EEPROM.read(3)) + "\"
onchange=\"setThreshold(this.value)\" min=\"0\" max=\"100\"></p>");
}
```

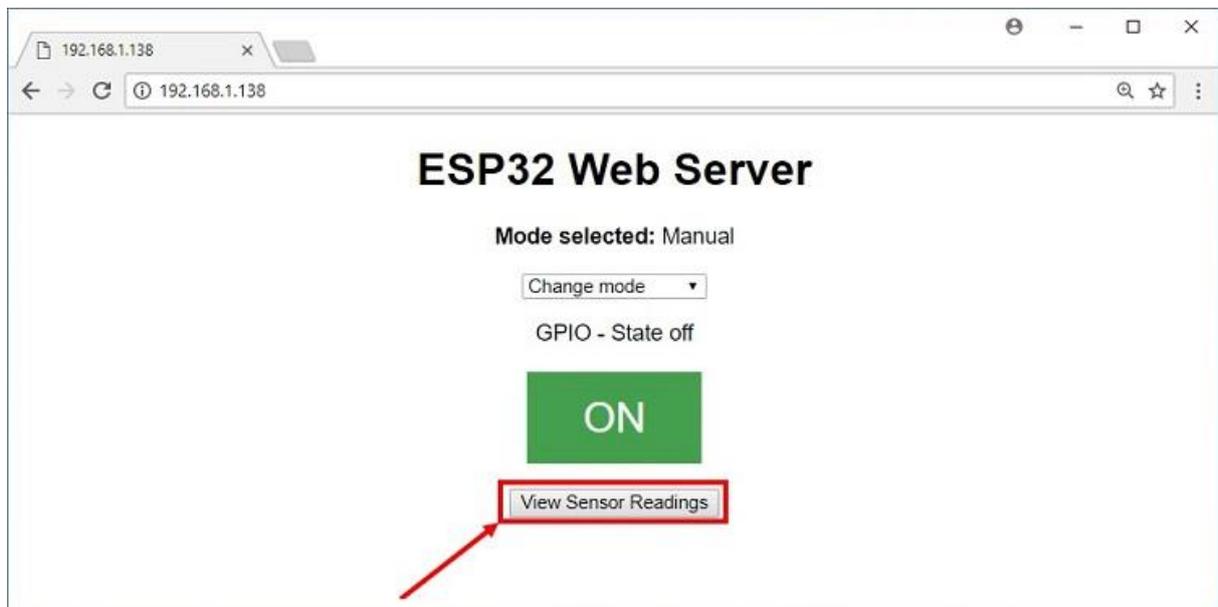
Auto PIR and LDR mode

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.

```
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 {
  float hic = dht.computeHeatIndex(t, h, false);
  dtostrf(hic, 6, 2, celsiusTemp);
  float hif = dht.computeHeatIndex(f, h);
  dtostrf(hif, 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, accordingly 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 accordingly 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;
  }
}

```

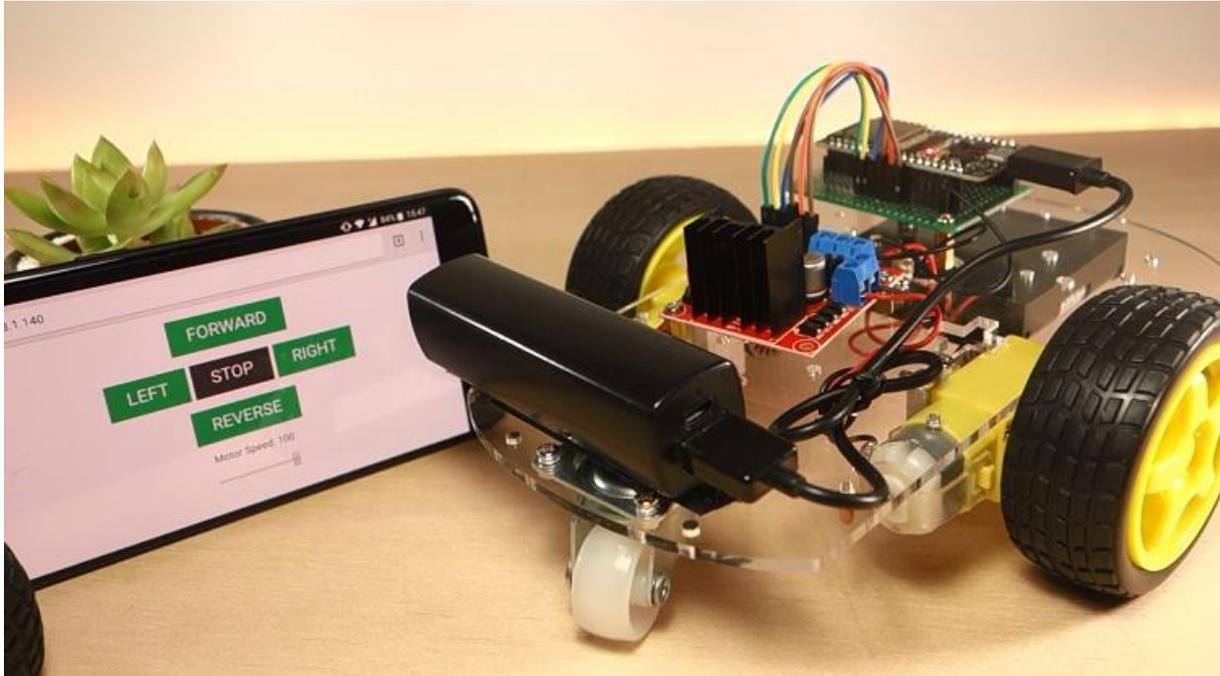
That's pretty much how the code works, we've also put an effort to write a bunch of comments in the code to make it easier to understand.

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 step by step how to create an ESP32 Wi-Fi remote controlled car robot.



This project is quite long, so it is divided into two parts.

- First, we'll take a look on how to control DC motors with the ESP32 and assemble the circuit;
- Then, we'll program the ESP32 with a Web Server to control the Robot.

This project applies several concepts addressed throughout the course. For better understanding this project we recommend taking a look at the following units if you haven't already:

- **ESP32 Pulse-Width Modulation (PWM):** Module 2, Unit 3
- **ESP32 Web Server – Control Outputs:** Module 4, Unit 2
- **Control Servo Motor Remotely (Web Server):** Module 4, Unit 8

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

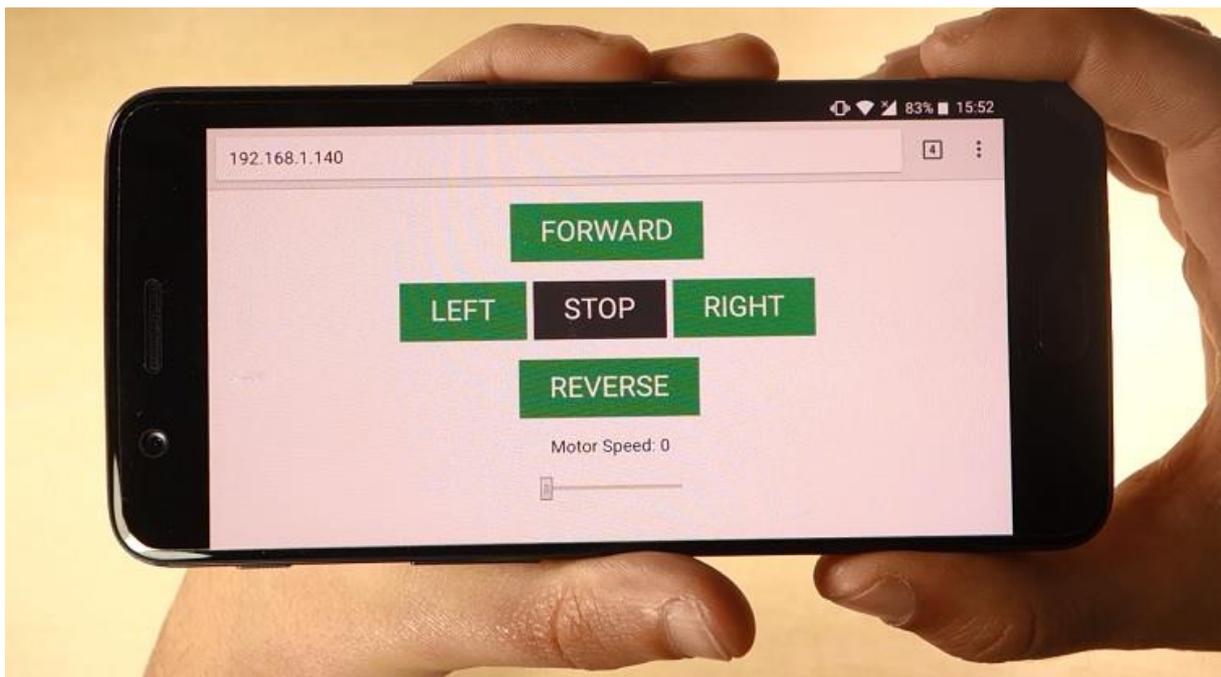
Before starting the project, we'll highlight the most important features and components used to make the robot work.

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 in any device inside your local network.

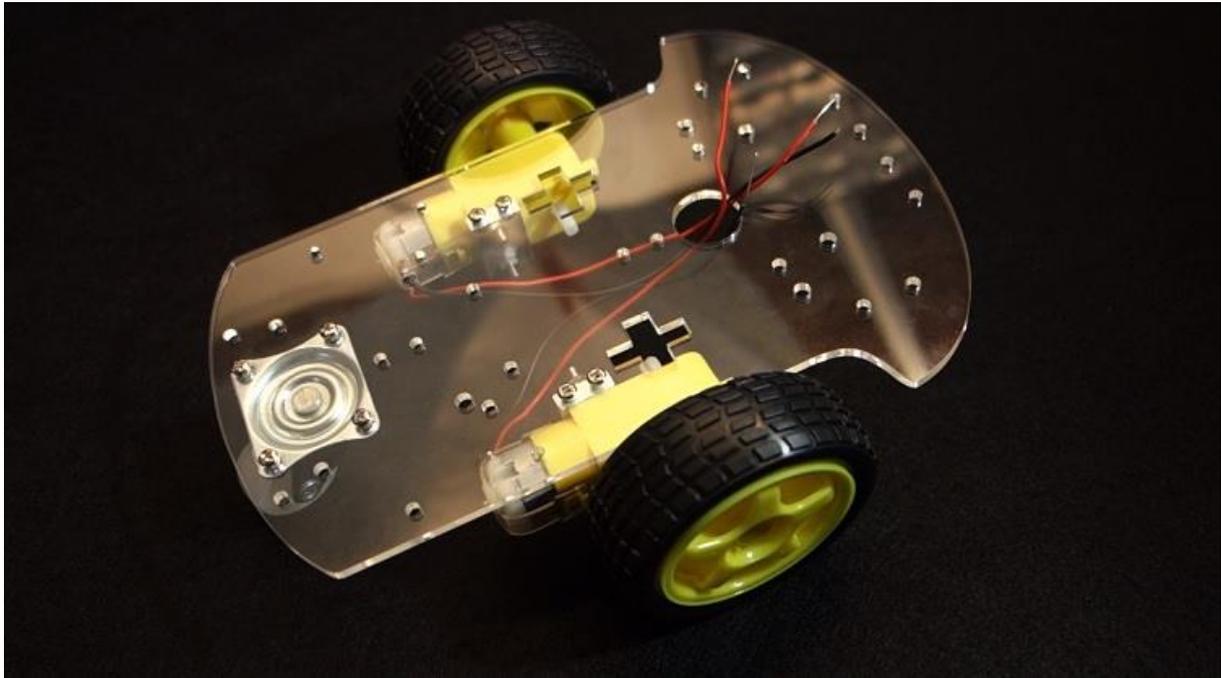
Robot Controls

The web server has 5 control options: forward, reverse, right, left, and stop. There's also a slider to control the speed.



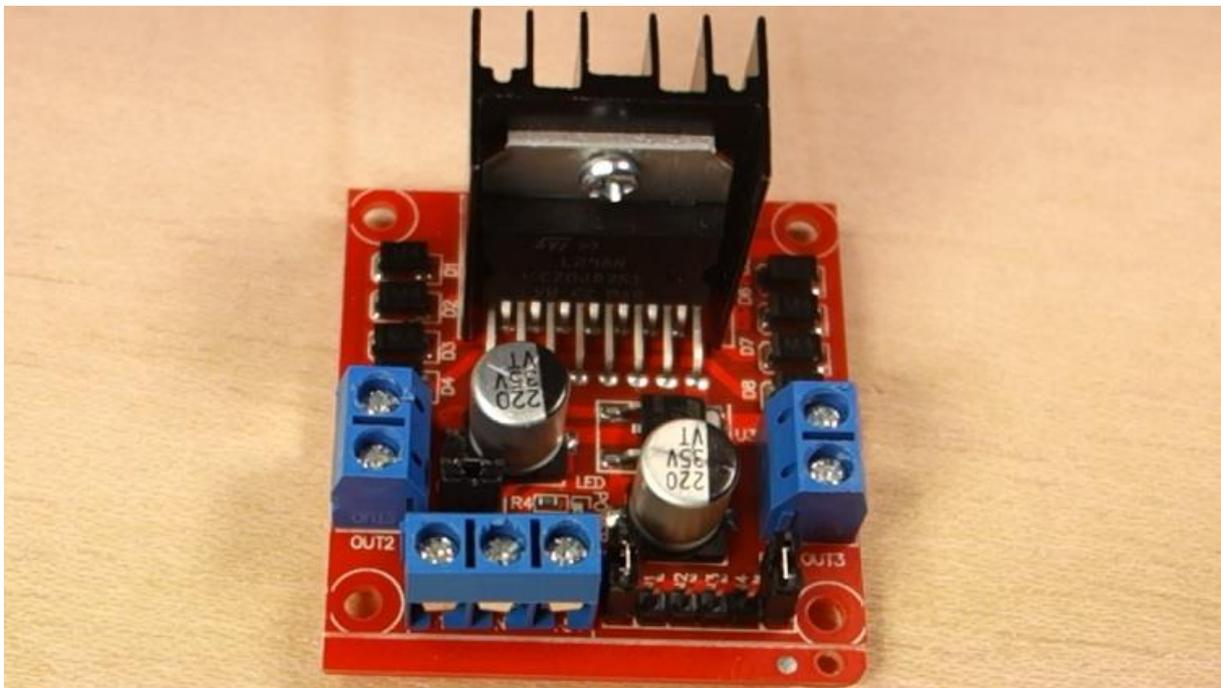
Smart Robot Chassis Kit

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

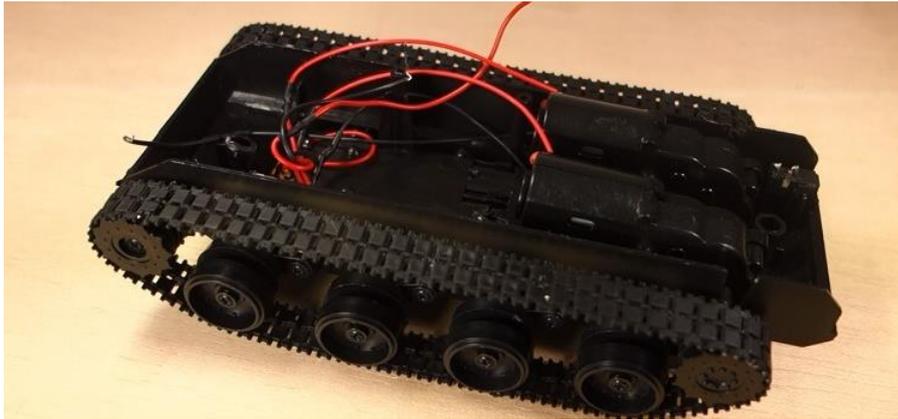
The motors draw a lot of current, so you need to use an external power supply. This means you need two different power sources. One will power the DC motors, and the other will power the ESP32. We'll power the ESP32 using a powerbank (like the ones used to charge your smartphone).



The motors will be powered using 4AA 1.5V batteries. You might consider using rechargeable batteries or any other suitable power supply.



Note: Our first idea was to build a robot using the tank kit in the following figure. However, we've faced some problems with this chassis due to its bad design.



We wrote a small review at the end of this Unit about this kit that shows you some reasons why we don't recommend the tank kit. So, to prevent future headaches we've decided to use the smart robot chassis kit.

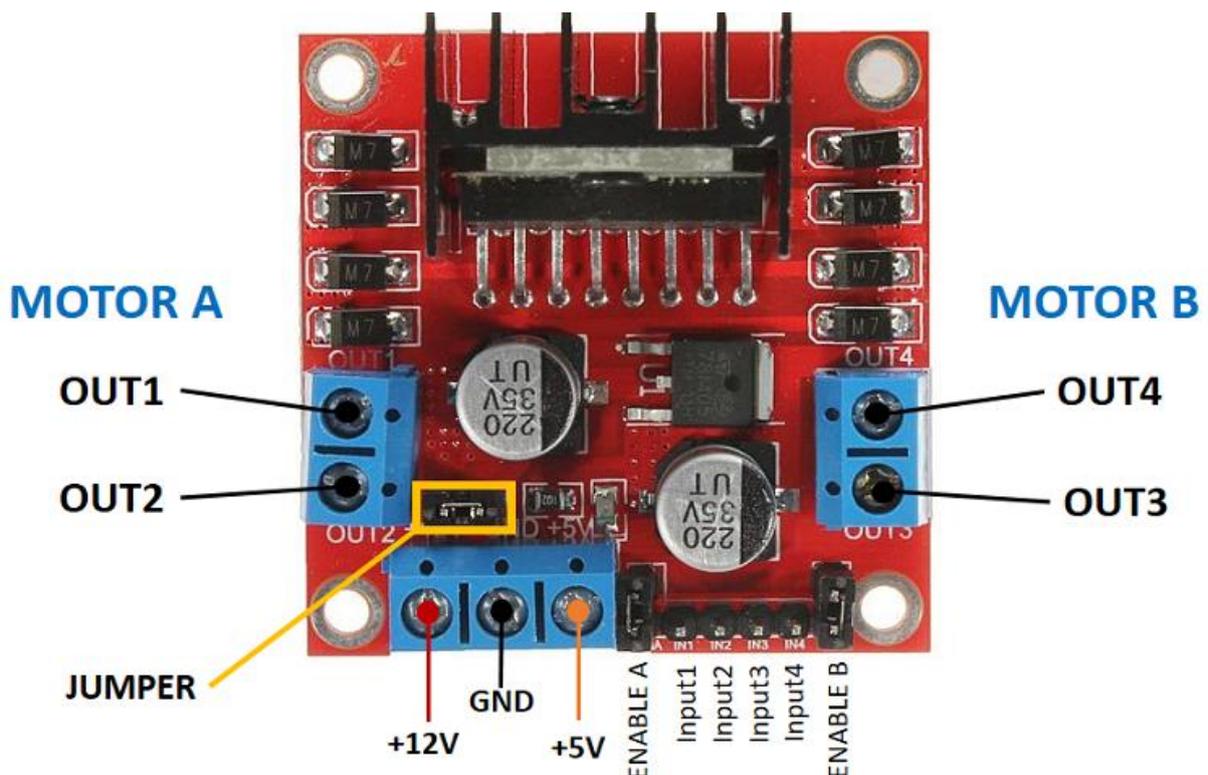
Introducing the L298N Motor Driver

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, which is perfect to build 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 in 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.

Note: If you supply more than 12V, you need to remove the jumper and supply 5V to the **+5V** terminal.

It's important to note that despite the **+12V** terminal name, with the setup we'll use here (with the jumper in place) you can supply any voltage between 6V and 12V. In our project will be using 4 AA 1.5V batteries that combined output approximately 6V.



In summary:

- **+12V:** The +12V terminal is where you should connect your 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

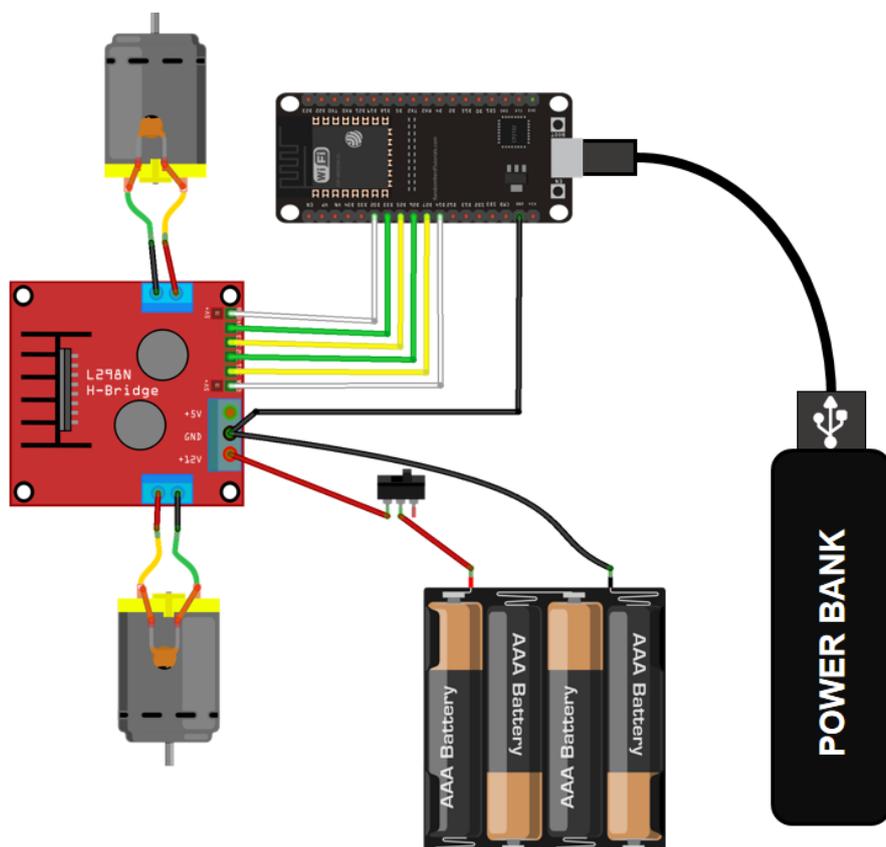
Now that you know how to control DC motors with the L298N motor driver, let's assemble the robot and build the circuit.

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.



(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.)

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

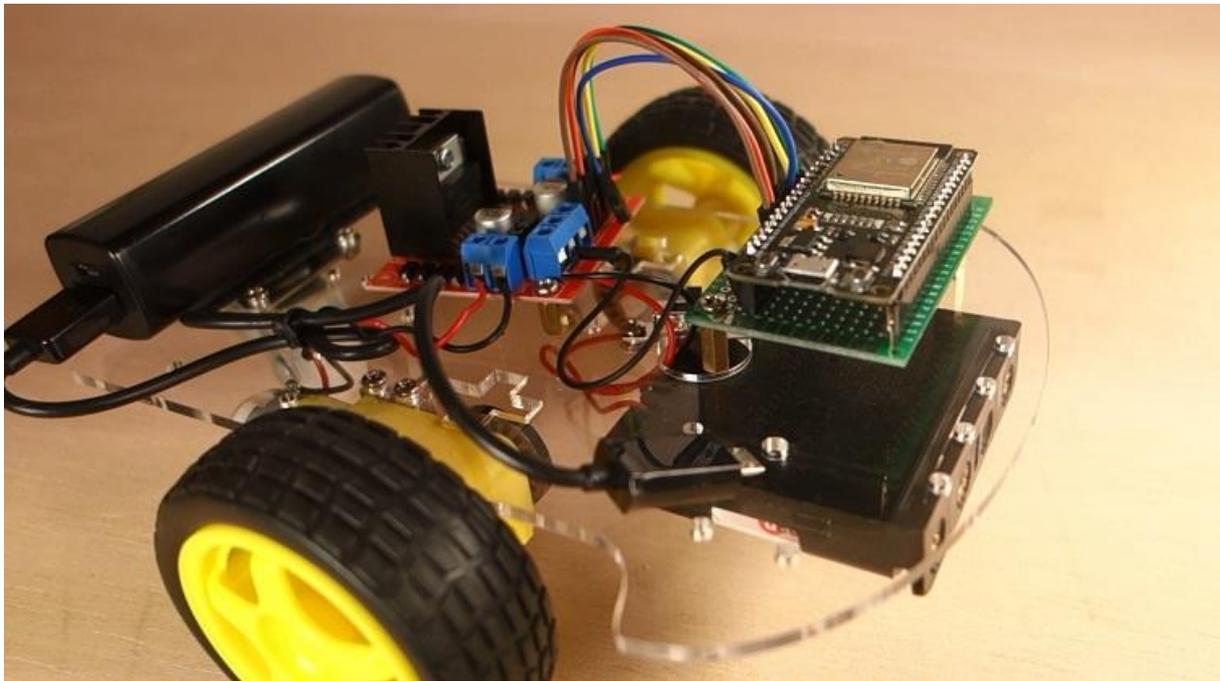
After that, wire each motor to their terminal blocks. We recommend soldering a 0.1 uF ceramic capacitor to the positive and negative terminals of each motor, as shown in the diagram to help smooth out any voltage spikes.

Additionally, you can solder a slider switch to the red wire that comes from the battery pack. This way, you can turn the power that goes to the motors and motor driver on and off.

Finally, power the motors using 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:

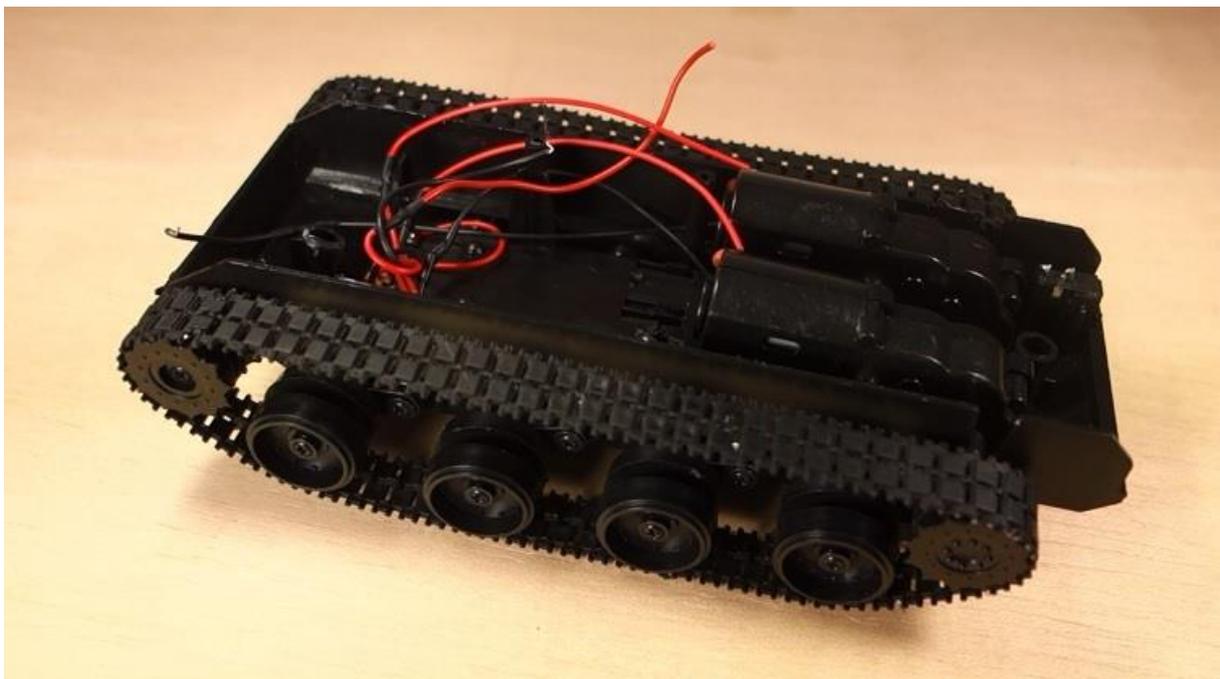


Continues...

Go to the next Unit to program the ESP32 with a web server to control the robot.

Robot Tank Chassis Kit – Not Recommended

As we've mentioned previously, our initial idea was to build a Robot using the Tank Kit shown in the figure below:

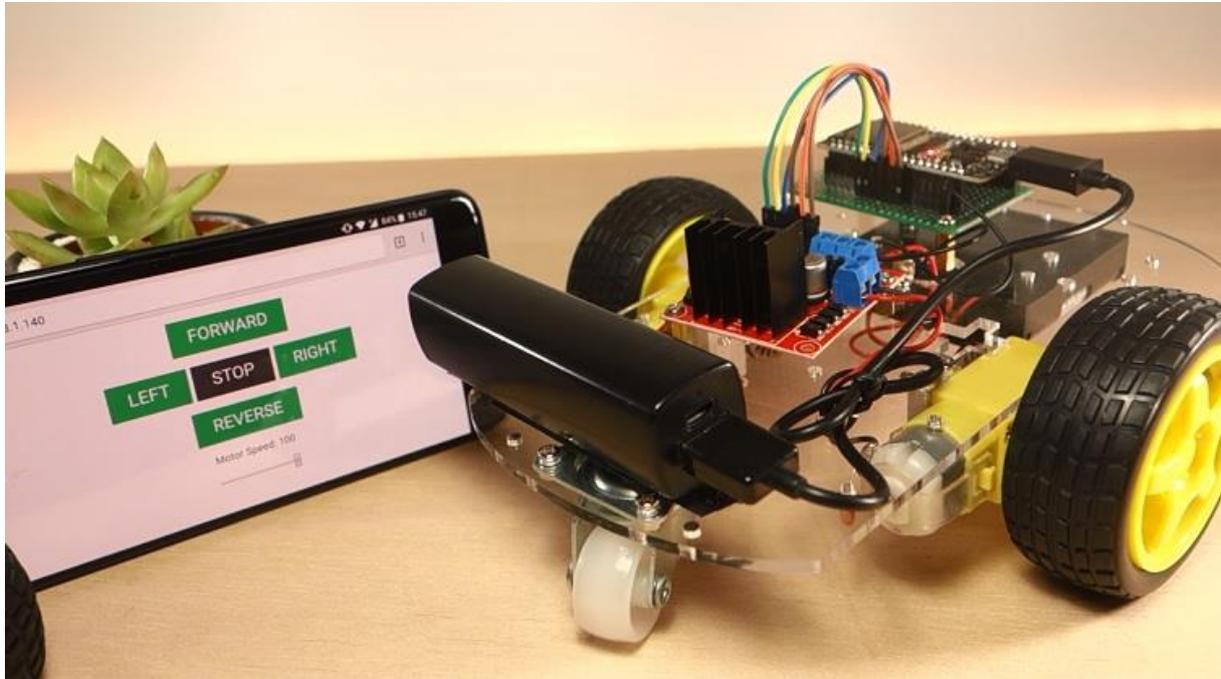


However, we don't recommend this kit for the following reasons:

- There isn't much space to accommodate the circuit. We had to break some plastics to put the circuit inside the chassis kit.
- The kit comes with space for 4 AA batteries, which is a great feature. However, 4 AA batteries were not enough to drive the motors.
- We added another 2 AA batteries. With 6 batteries we were able to spin the motors. However, the chassis become too heavy. When we put it on the floor, the motors didn't have enough strength to make the tank move.

For these reasons, we don't recommend this kit, especially if this is your first time building a robot.

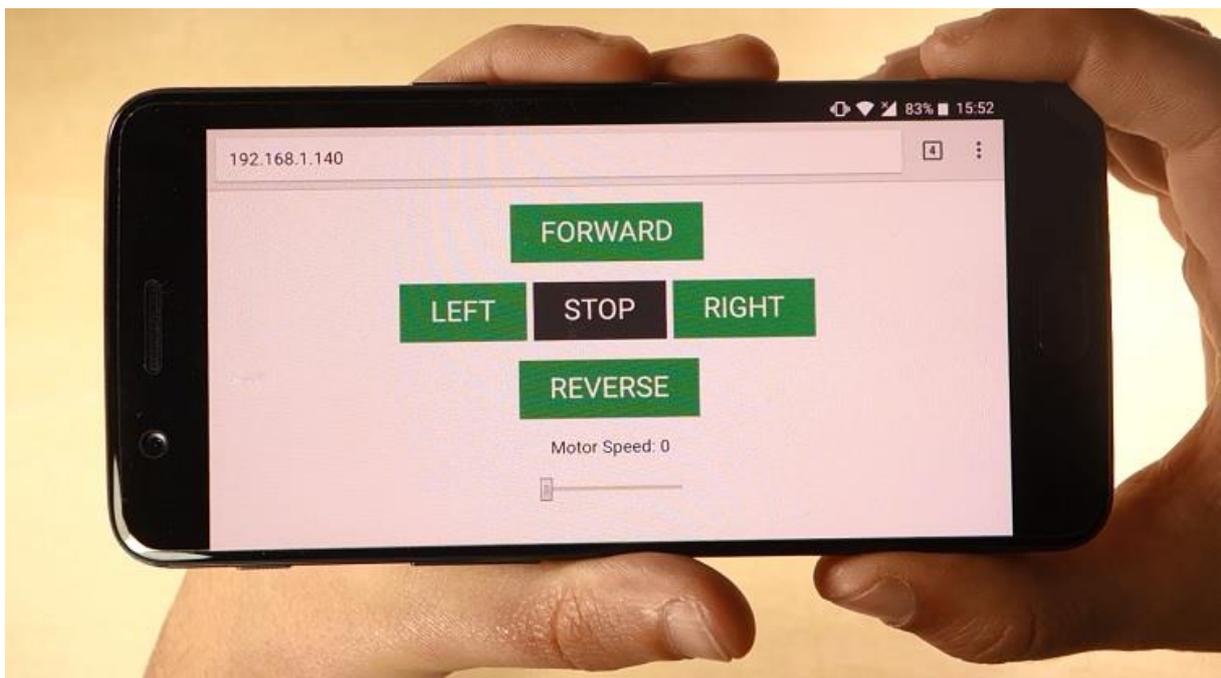
Unit 2 - Remote Controlled Wi-Fi Car Robot (Part 2/2)



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.

Note: We've already covered how to create buttons and sliders and how to control them with the ESP32. For a refresher, take a look at the "**Web Servers**" Module.

Code

Copy the following code to your Arduino IDE. Don't upload it yet. First, we'll take a quick look on how it works.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Project_Robot/Project_Robot.ino

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

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

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

// Variable to store the HTTP request
String header;

// 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 pwmChannel = 0;
const int resolution = 8;
int dutyCycle = 0;

// Decode HTTP GET value
String valueString = String(5);
int pos1 = 0;
int pos2 = 0;

// 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() {
  Serial.begin(115200);

  // Set the Motor pins as outputs
  pinMode(motor1Pin1, OUTPUT);
  pinMode(motor1Pin2, OUTPUT);
  pinMode(motor2Pin1, OUTPUT);
  pinMode(motor2Pin2, OUTPUT);

  // Configure PWM channel functionalities
  ledcSetup(pwmChannel, freq, resolution);

  // Attach the PWM channel 0 to the enable pins which are the GPIOs to be
  controlled
  ledcAttachPin(enable1Pin, pwmChannel);
  ledcAttachPin(enable2Pin, pwmChannel);

  // Produce a PWM signal to both enable pins with a duty cycle 0
  ledcWrite(pwmChannel, dutyCycle);

  // 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

```

```

    while (client.connected() && currentTime - previousTime <= timeoutTime)
{ // loop while the client's connected
    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();

                // Controls the motor pins according to the button pressed
                if (header.indexOf("GET /forward") >= 0) {
                    Serial.println("Forward");
                    digitalWrite(motor1Pin1, LOW);
                    digitalWrite(motor1Pin2, HIGH);
                    digitalWrite(motor2Pin1, LOW);
                    digitalWrite(motor2Pin2, HIGH);
                } else if (header.indexOf("GET /left") >= 0) {
                    Serial.println("Left");
                    digitalWrite(motor1Pin1, LOW);
                    digitalWrite(motor1Pin2, LOW);
                    digitalWrite(motor2Pin1, LOW);
                    digitalWrite(motor2Pin2, HIGH);
                } else if (header.indexOf("GET /stop") >= 0) {
                    Serial.println("Stop");
                    digitalWrite(motor1Pin1, LOW);
                    digitalWrite(motor1Pin2, LOW);
                    digitalWrite(motor2Pin1, LOW);
                    digitalWrite(motor2Pin2, LOW);
                } else if (header.indexOf("GET /right") >= 0) {
                    Serial.println("Right");
                    digitalWrite(motor1Pin1, LOW);
                    digitalWrite(motor1Pin2, HIGH);
                    digitalWrite(motor2Pin1, LOW);
                    digitalWrite(motor2Pin2, LOW);
                } else if (header.indexOf("GET /reverse") >= 0) {
                    Serial.println("Reverse");
                    digitalWrite(motor1Pin1, HIGH);
                    digitalWrite(motor1Pin2, LOW);
                    digitalWrite(motor2Pin1, HIGH);

```

```

        digitalWrite(motor2Pin2, LOW);
    }
    // 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 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 { -webkit-user-select: none; -moz-user-
select: none; -ms-user-select: none; user-select: none; background-color:
#4CAF50;}");
        client.println("border: none; color: white; padding: 12px 28px;
text-decoration: none; font-size: 26px; margin: 1px; cursor: pointer;}");
    client.println(".button2 {background-color: #555555;}</style>");
    client.println("<script
src=\"https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js\"</
script></head>");

    // Web Page
    client.println("<p><button                                class=\"button\"
onclick=\"moveForward()\">FORWARD</button></p>");
    client.println("<div                                style=\"clear:        both;\"><p><button
class=\"button\" onclick=\"moveLeft()\">LEFT </button>");
    client.println("<button                                class=\"button        button2\"
onclick=\"stopRobot()\">STOP</button>");
    client.println("<button                                class=\"button\"
onclick=\"moveRight()\">RIGHT</button></p></div>");
    client.println("<p><button                                class=\"button\"
onclick=\"moveReverse()\">REVERSE</button></p>");
    client.println("<p>Motor                                Speed:                                <span
id=\"motorSpeed\"></span></p>");
    client.println("<input type=\"range\" min=\"0\" max=\"100\"
step=\"25\" id=\"motorSlider\" onchange=\"motorSpeed(this.value)\" value=\"\"
+ valueString + \"\"/>");

    client.println("<script>$.ajaxSetup({timeout:1000});");
    client.println("function moveForward() { $.get(\"/forward\");
{Connection: close};}");
    client.println("function moveLeft() { $.get(\"/left\");
{Connection: close};}");
    client.println("function stopRobot() {$.get(\"/stop\");
{Connection: close};}");
    client.println("function moveRight() { $.get(\"/right\");
{Connection: close};}");
    client.println("function moveReverse() { $.get(\"/reverse\");
{Connection: close};}");
    client.println("var                                slider                                =
document.getElementById(\"motorSlider\");");
    client.println("var                                motorP                                =
document.getElementById(\"motorSpeed\"); motorP.innerHTML = slider.value;");
    client.println("slider.oninput = function() { slider.value =
this.value; motorP.innerHTML = this.value; }");
    client.println("function motorSpeed(pos) { $.get(\"/?value=\" +
pos + \"&\"); {Connection: close};}</script>");

```

```

        client.println("</html>");

        //Request example: GET /?value=100& HTTP/1.1 - sets PWM duty
        cycle to 100% = 255
        if(header.indexOf("GET /?value=")>=0) {
            pos1 = header.indexOf('=');
            pos2 = header.indexOf('&');
            valueString = header.substring(pos1+1, pos2);
            //Set motor speed value
            if (valueString == "0") {
                ledcWrite(pwmChannel, 0);
                digitalWrite(motor1Pin1, LOW);
                digitalWrite(motor1Pin2, LOW);
                digitalWrite(motor2Pin1, LOW);
                digitalWrite(motor2Pin2, LOW);
            }
            else {
                dutyCycle = map(valueString.toInt(), 25, 100, 200, 255);
                ledcWrite(pwmChannel, dutyCycle);
                Serial.println(valueString);
            }
        }
        // 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("");
}
}

```

How the Code Works

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 to the following variables, so that the ESP32 is able to 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.

```
// 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

You'll need to send PWM signals to the enable pins to control the motor speed. So, you need to create variables to set the PWM signal properties like frequency, the PWM channel, resolution, and duty cycle.

```
// Setting PWM properties  
const int freq = 30000;  
const int pwmChannel = 0;  
const int resolution = 8;  
int dutyCycle = 0;
```

Note: for more information about how to use PWM with the ESP32, check Module 2, Unit 3: **ESP32 Pulse-Width Modulation (PWM)**.

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);
```

Then, you configure the PWM channel with the properties you've defined earlier.

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

In the next lines, you attach both enable pins to the same PWM channel. So, you'll get the same signal on the enable1 and enable 2 pins.

```
ledcAttachPin(enable1Pin, pwmChannel);  
ledcAttachPin(enable2Pin, pwmChannel);
```

Finally, you generate the PWM signal with a defined duty cycle using the `ledcWrite()` function.

```
ledcWrite(pwmChannel, dutyCycle);
```

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();
```

loop()

In the `loop()`, the ESP32 is always listening for incoming clients, and when a request is received we save the incoming data.

```
WiFiClient client = server.available();
if (client) {
    Serial.println("New Client.");
    String currentLine = "";
    while (client.connected()) {
        if (client.available()) {
            char c = client.read();
            Serial.write(c);
            header += c;
            if (c == '\n') {
                if (currentLine.length() == 0) {
                    client.println("HTTP/1.1 200 OK");
                    client.println("Content-type:text/html");
                    client.println("Connection: close");
                    client.println();
                }
            }
        }
    }
}
```

Controlling the Robot

The following section of `if` and `else` statements checks which command was sent to the robot, whether it's a forward, left, stop, right, or reverse command. Then, it controls the motors accordingly.

For example for the robot to turn left, all the pins are set to LOW with the exception of `motor2Pin2` that is HIGH.

```
if (header.indexOf("GET /forward") >= 0) {
    Serial.println("Forward");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, HIGH);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, HIGH);
} else if (header.indexOf("GET /left") >= 0) {
```

```

    Serial.println("Left");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, HIGH);
} else if (header.indexOf("GET /stop") >= 0) {
    Serial.println("Stop");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, LOW);
} else if (header.indexOf("GET /right") >= 0) {
    Serial.println("Right");
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, HIGH);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, LOW);
} else if (header.indexOf("GET /reverse") >= 0) {
    Serial.println("Reverse");
    digitalWrite(motor1Pin1, HIGH);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, HIGH);
    digitalWrite(motor2Pin2, LOW);
}
}

```

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**.

Displaying the web page

The following part of the code displays the web page on your browser.

In simple terms, 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

```

// 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 buttons
client.println("<style>html { font-family: Helvetica; display: inline-
block; margin: 0px auto; text-align: center;}");
client.println(".button { -webkit-user-select: none; -moz-user-select:
none; -ms-user-select: none; user-select: none; background-color:
#4CAF50;}");
client.println("border: none; color: white; padding: 12px 28px; text-
decoration: none; font-size: 26px; margin: 1px; cursor: pointer;}");
client.println(".button2 {background-color: #555555;}</style>");
client.println("<script
src=\"https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js\"></
script></head>");

// Web Page
client.println("<p><button class=\"button\"
onclick=\"moveForward()\">FORWARD</button></p>");
client.println("<div style=\"clear: both;\"><p><button class=\"button\"
onclick=\"moveLeft()\">LEFT </button>");
client.println("<button class=\"button button2\"
onclick=\"stopRobot()\">STOP</button>");
client.println("<button class=\"button\"
onclick=\"moveRight()\">RIGHT</button></p></div>");

```

```

client.println("<p><button class=\"button\"
onclick=\"moveReverse()\">REVERSE</button></p>");
client.println("<p>Motor Speed: <span
id=\"motorSpeed\"></span></p>");
client.println("<input type=\"range\" min=\"0\" max=\"100\" step=\"25\"
id=\"motorSlider\" onchange=\"motorSpeed(this.value)\" value=\"\" +
valueString + \"\"/>");
client.println("<script>$.ajaxSetup({timeout:1000});");
client.println("function moveForward() { $.get(\"/forward\"); {Connection:
close};});");
client.println("function moveLeft() { $.get(\"/left\"); {Connection:
close};});");
client.println("function stopRobot() {$.get(\"/stop\"); {Connection:
close};});");
client.println("function moveRight() { $.get(\"/right\"); {Connection:
close};});");
client.println("function moveReverse() { $.get(\"/reverse\"); {Connection:
close};});");
client.println("var slider = document.getElementById(\"motorSlider\");");
client.println("var motorP = document.getElementById(\"motorSpeed\");
motorP.innerHTML = slider.value;");
client.println("slider.oninput = function() { slider.value = this.value;
motorP.innerHTML = this.value; });");
client.println("function motorSpeed(pos) { $.get(\"/?value=\" + pos +
\"&\"); {Connection: close};}</script>");
client.println("</html>");

```

Note: We've covered in great detail how to build the web page for the web server in a previous Module. Check Module 4, Unit 3: **ESP32 Web Server – HTML and CSS Basics (Part 1/2)**.

Controlling the Speed

In the following section is where we get the slider value and change the motor speed. We use the same method we've used earlier to control a servo motor – see Module 4, Unit 9: **ESP32 Control Servo Motor Remotely (Web Server)**.

```

if(header.indexOf("GET /?value=")>=0) {
  pos1 = header.indexOf('=');
  pos2 = header.indexOf('&');
  valueString = header.substring(pos1+1, pos2);
  //Set motor speed value
  if (valueString == "0") {
    ledcWrite(pwmChannel, 0);
    digitalWrite(motor1Pin1, LOW);
    digitalWrite(motor1Pin2, LOW);
    digitalWrite(motor2Pin1, LOW);
    digitalWrite(motor2Pin2, LOW);
  }
  else {
    dutyCycle = map(valueString.toInt(), 25, 100, 200, 255);
    ledcWrite(pwmChannel, dutyCycle);
    Serial.println(valueString);
  }
}

```

The valueString variable is used to save the current slider value.

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 (valueString == "0") {
  ledcWrite(pwmChannel, 0);
  digitalWrite(motor1Pin1, LOW);
  digitalWrite(motor1Pin2, LOW);
  digitalWrite(motor2Pin1, LOW);
  digitalWrite(motor2Pin2, LOW);
}
```

If the slider value is different than zero, we're going to 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.

```
else {
  dutyCycle = map(valueString.toInt(), 25, 100, 200, 255);
  ledcWrite(pwmChannel, dutyCycle);
  Serial.println(valueString);
}
```

We've found that a minimum of 200 for the duty cycle works just fine for this example. Then, we set the set the duty cycle to the PWM channel. This is what actually controls the speed of your motors.

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 see the ESP32 IP address.

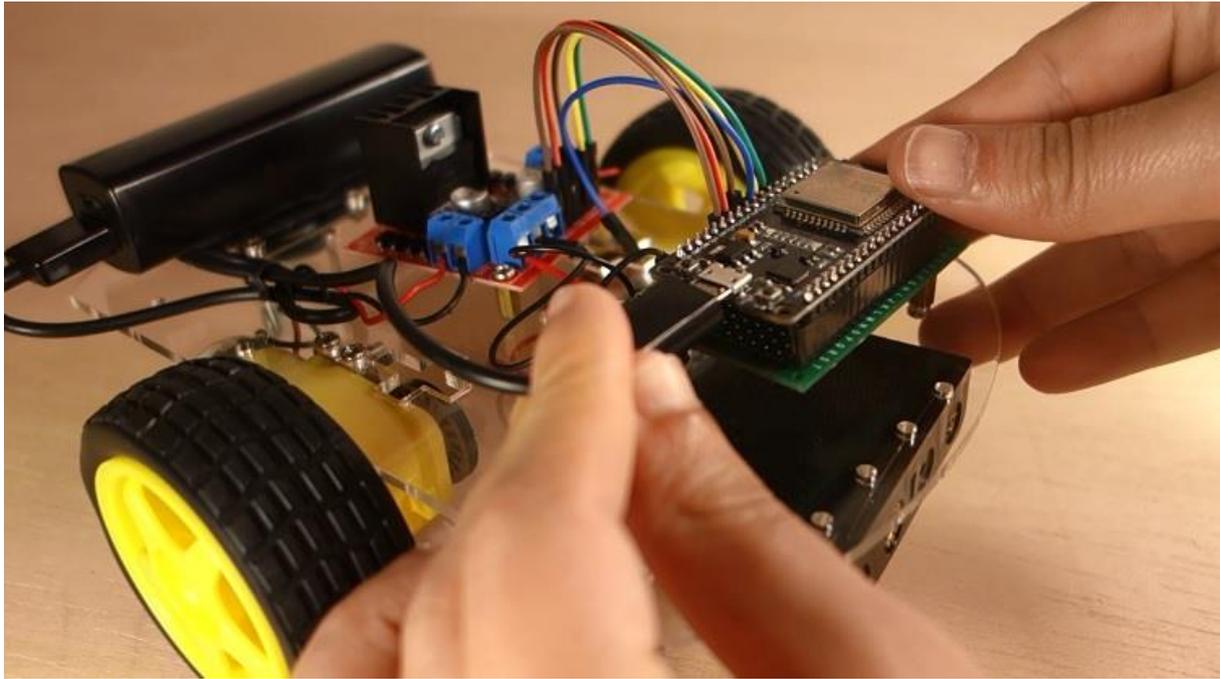
A screenshot of the Serial Monitor window. The window title is "COM4". The output text is as follows:

```
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
config: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
Connecting to MEO-620B4B
..
WiFi connected.
IP address:
192.168.1.140
```

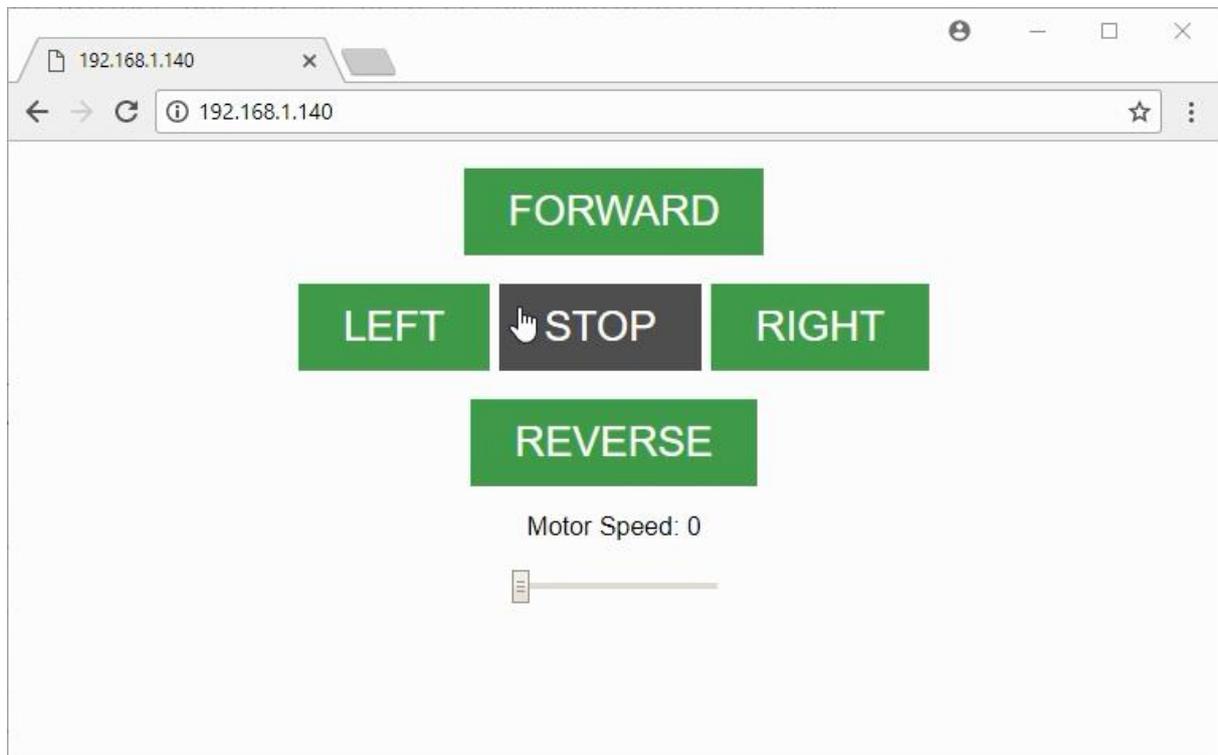
The IP address "192.168.1.140" is highlighted with a red rectangular box. At the bottom of the window, there are controls for "Autoscroll" (checked), "No line ending", "115200 baud", and "Clear output".

Unplug the ESP32 from your computer and power it with the powerbank.

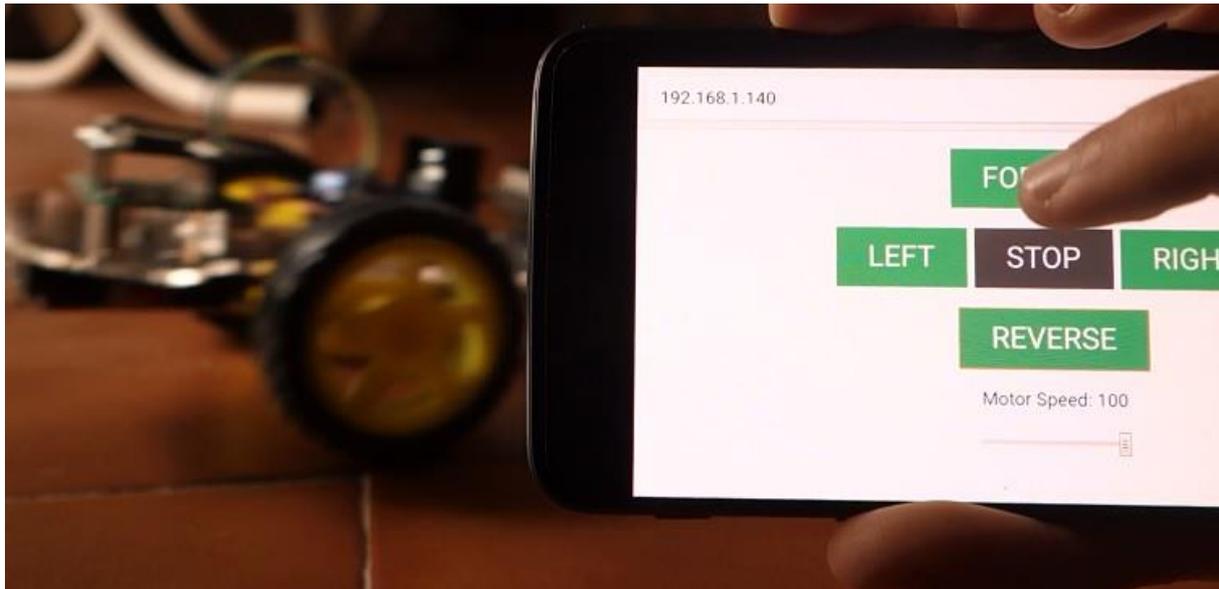


Make sure you also have the 4 AA batteries in place and the slider switch turned on.

Open your browser, and write 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 should be able to 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 completed! You can make your robot go forward, backwards, right, and left. You can stop it by tapping the STOP button. A really 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.



Note: my dog freaks out every time I build a new robot!

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 sees an obstacle;

In our opinion this is a great project to put into practice a lot of concepts learned throughout the course.

We hope you had fun building the robot.

Unit 3 - Assembling the Smart Robot

Car Chassis Kit

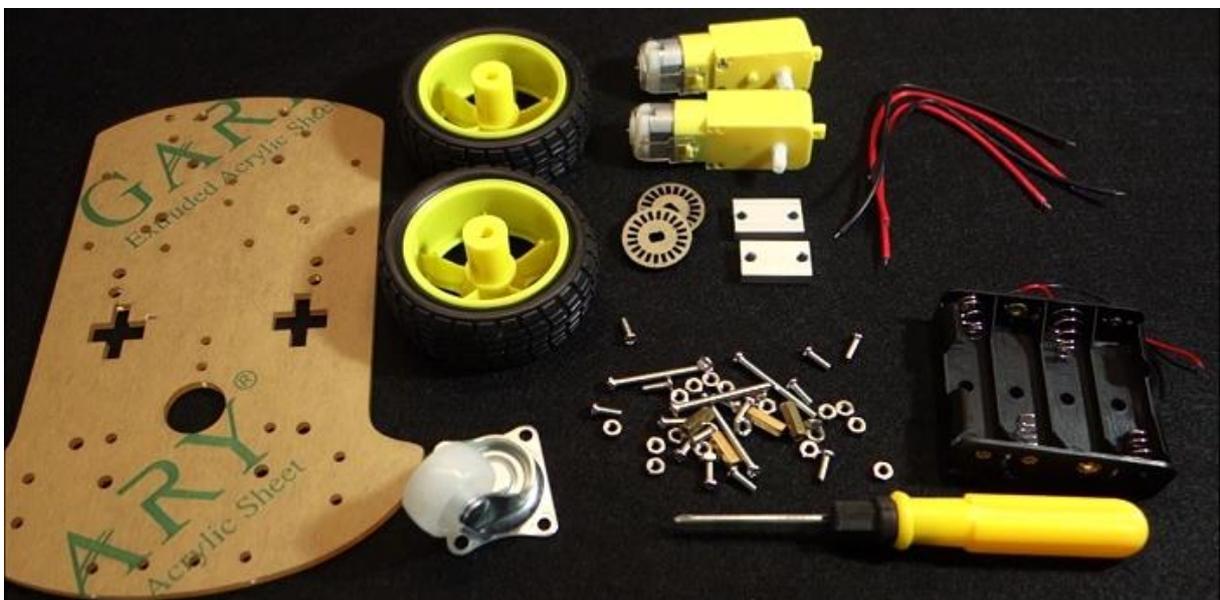
In this Unit you're going to learn how to build a smart robot car chassis kit that is commonly used to build a smart car robot with the ESP32, ESP8266, Arduino boards, etc.

How to Assemble the Kit

- 1) This is the package that you get with the [Smart Robot Car Chassis Kit](#).



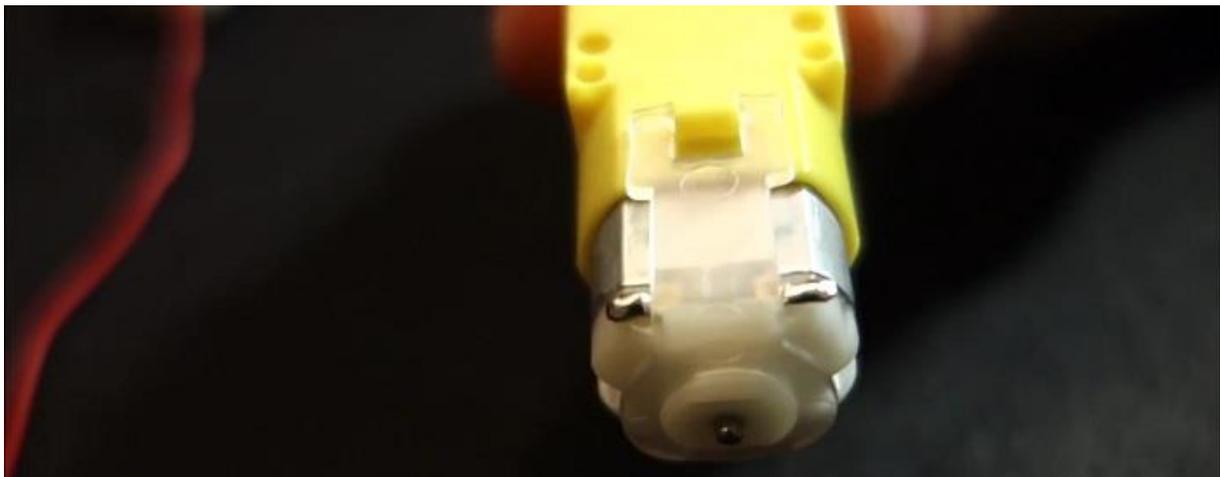
- 2) Open the package – it comes with one screwdriver, two DC motors, two wheels and one acrylic car chassis. There is also a small plastic bag that comes with a battery holder, a small wheel for the front, some bolts and screws, four wires and other required components to assemble the robot.



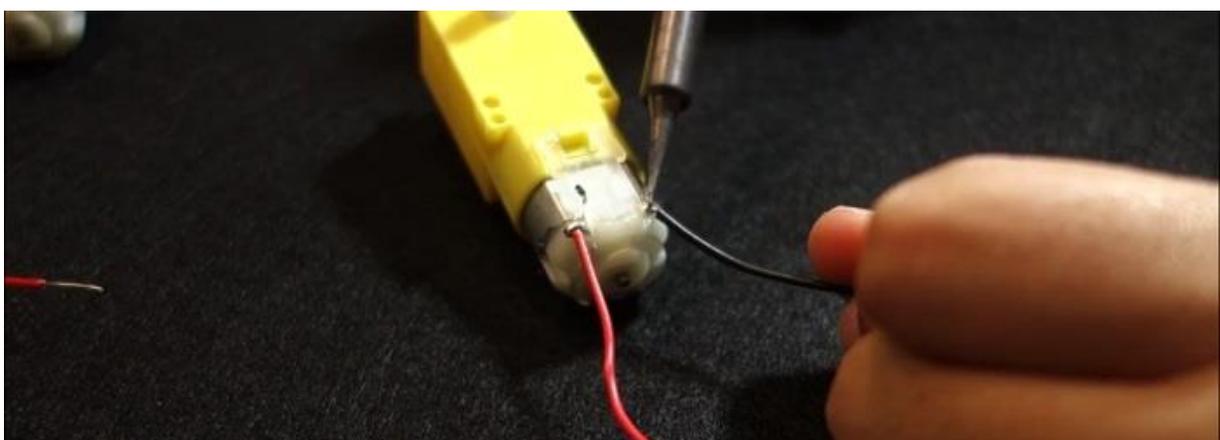
- 3) Although this kit comes with the needed wires, since they are a bit short and not very flexible, we've decided to replace them with other wires, because we can easily adjust their length. You'll need one red wire and one black wire for each motor. You can use a wire cutter to cut the wires to your desired length.



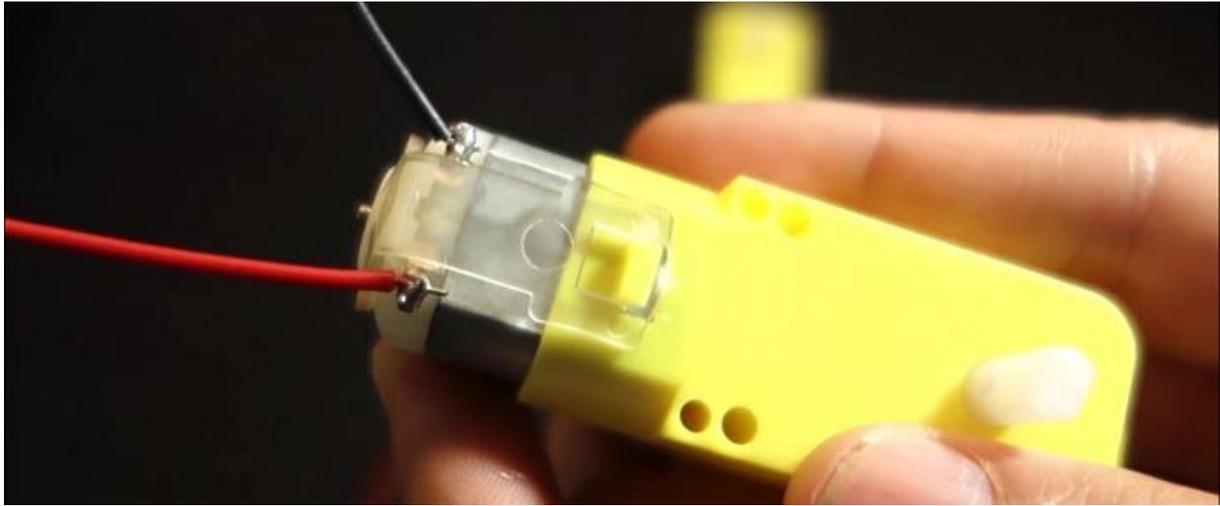
- 4) After preparing all four wires, you need to solder them to the DC motors. Tin the DC motor pins.



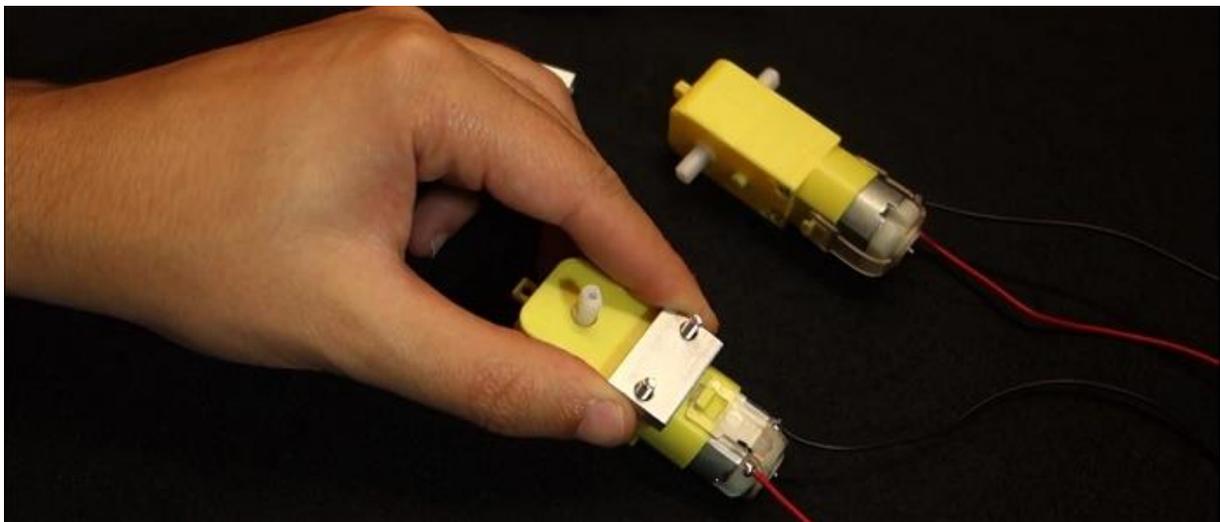
- 5) Then, grab the wire and solder it to the DC motor pin. Repeat that process to all the other wires.



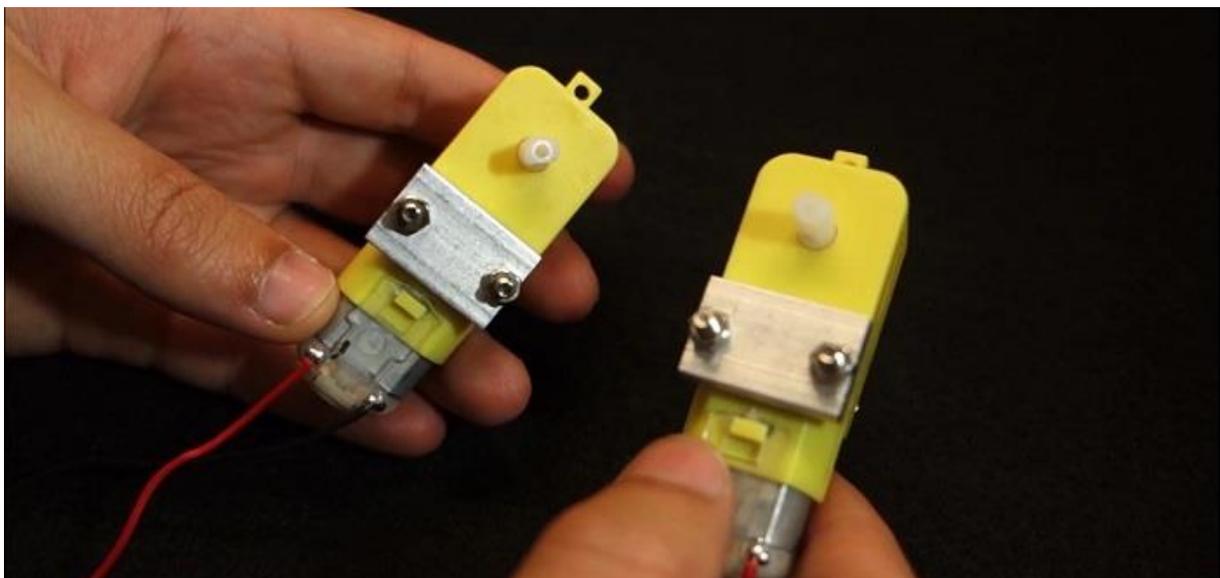
This is how the motors look like after soldering the wires.



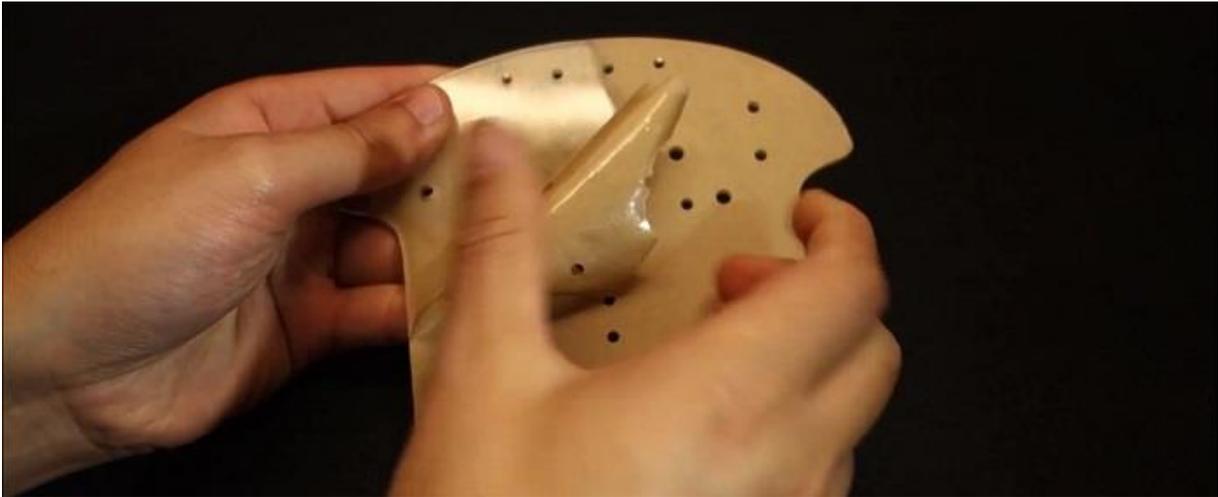
- 6) Now, you need the screwdriver, bolts and screws, and those metal pieces. Start by attaching the metal pieces to the DC motors.



- 7) Here's how they look like after this step:



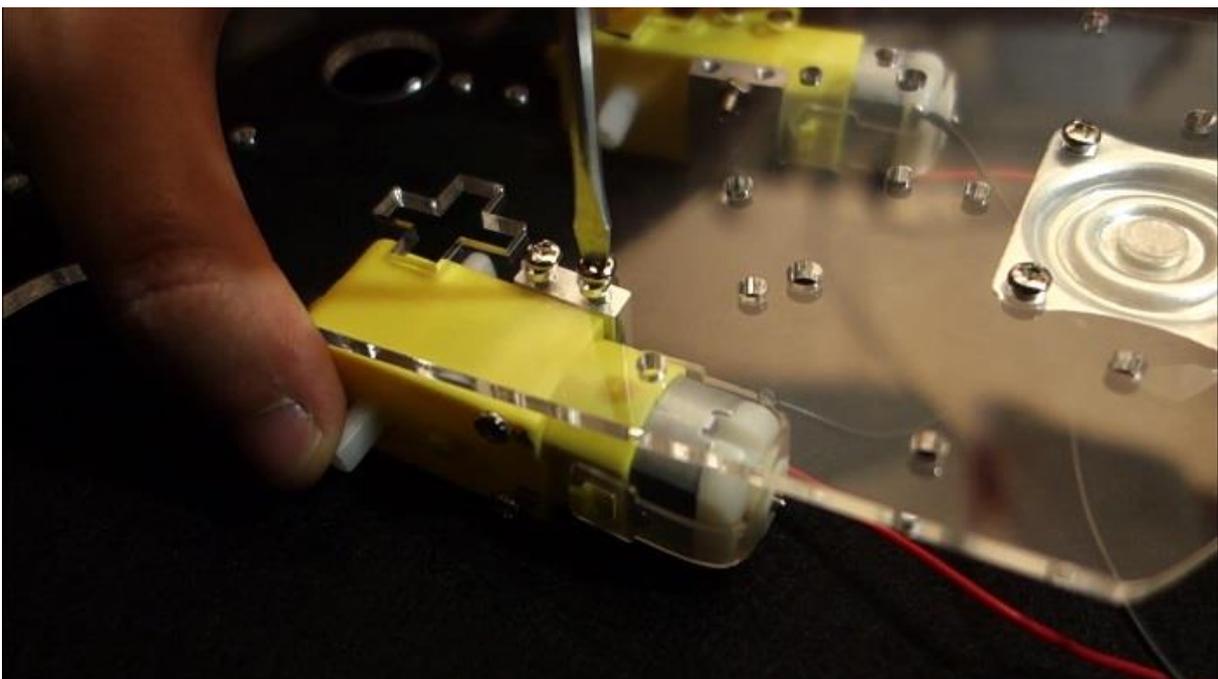
8) Remove the protective adhesive from the acrylic chassis in both sides.



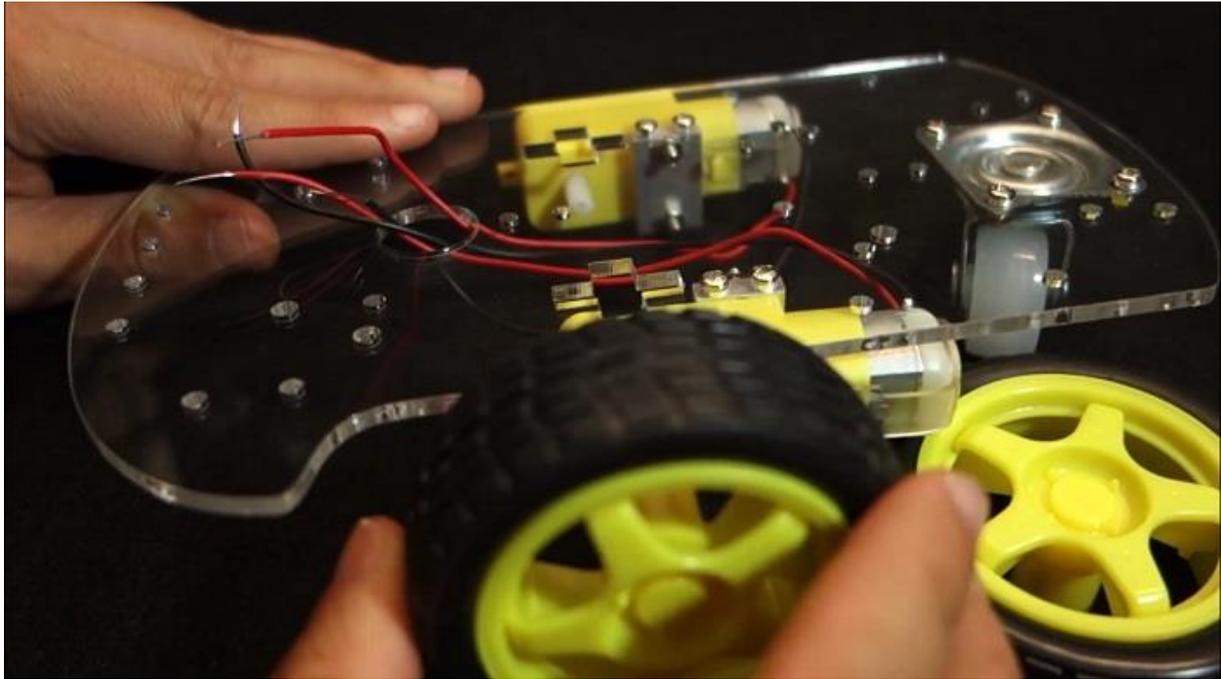
9) Grab the small wheel and attach it to the front part of the chassis.



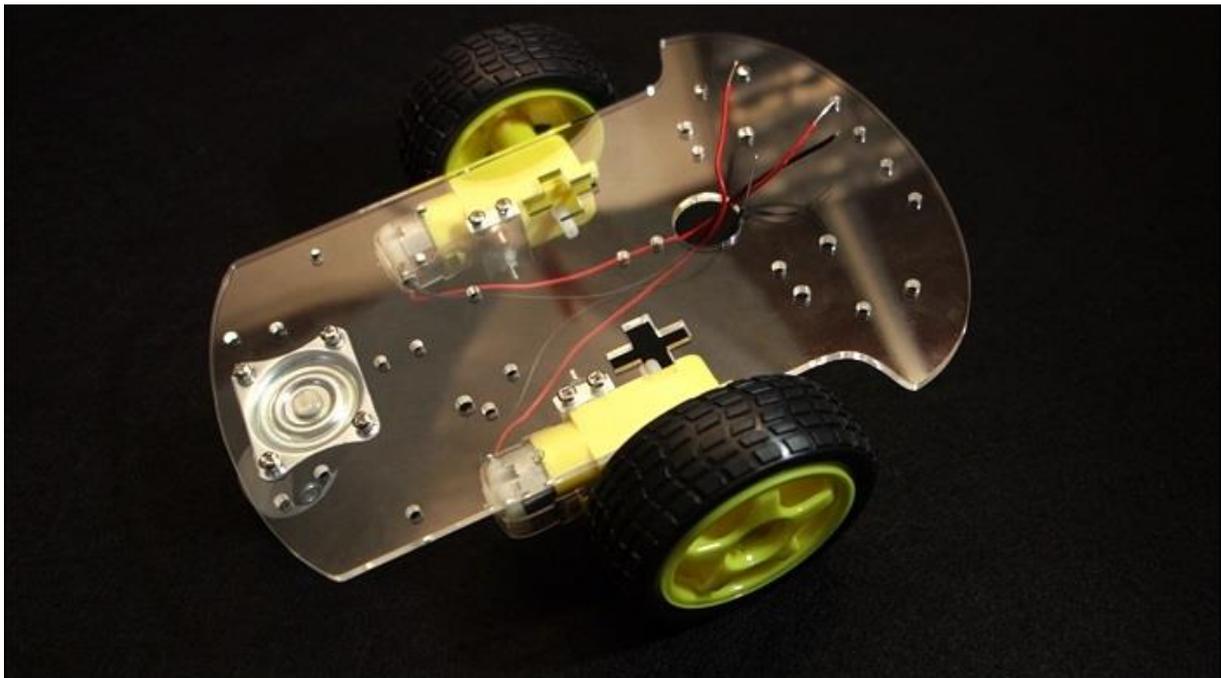
10) Finally, it is time to attach the DC motors to the chassis.



11) Lastly, connect the wheels to the DC motors.



Now, your robot car chassis is ready. Here's how it looks 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 if you want to take your Robot to a park and have some fun, you need to connect it to a wireless network (which is not very handy). 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.

Uploading the Sketch

To set the ESP32 as an Access Point, upload the next sketch to your board:

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Project_Robot_AP/Project_Robot_AP.ino

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

// You can customize the SSID name and change the password
const char* ssid      = "ESP32-Robot";
const char* password  = "123456789";

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

// Variable to store the HTTP request
String header;

// 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 pwmChannel = 0;
const int resolution = 8;
int dutyCycle = 0;

// Decode HTTP GET value
String valueString = "0";
int pos1 = 0;
int pos2 = 0;

// 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() {
    Serial.begin(115200);

    // Set the Motor pins as outputs
    pinMode(motor1Pin1, OUTPUT);
    pinMode(motor1Pin2, OUTPUT);
    pinMode(motor2Pin1, OUTPUT);
    pinMode(motor2Pin2, OUTPUT);

    // Configure PWM channel functionalities
    ledcSetup(pwmChannel, freq, resolution);

    // Attach the PWM channel 0 to the enable pins which are the GPIOs to be
    controlled
    ledcAttachPin(enable1Pin, pwmChannel);
    ledcAttachPin(enable2Pin, pwmChannel);

    // Produce a PWM signal to both enable pins with a duty cycle 0
    ledcWrite(pwmChannel, dutyCycle);

    // Connect to Wi-Fi network with SSID and password
    Serial.print("Setting AP (Access Point)...");
    // 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);
    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
    while (client.connected() && currentTime - previousTime <= timeoutTime)
{ // loop while the client's connected
    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();
                // turns the GPIOs on and off
                if (header.indexOf("GET /forward") >= 0) {
                    Serial.println("Forward");
                    digitalWrite(motor1Pin1, LOW);
                    digitalWrite(motor1Pin2, HIGH);
                    digitalWrite(motor2Pin1, LOW);
                    digitalWrite(motor2Pin2, HIGH);
                } else if (header.indexOf("GET /left") >= 0) {
                    Serial.println("Left");
                    digitalWrite(motor1Pin1, LOW);
                    digitalWrite(motor1Pin2, LOW);
                    digitalWrite(motor2Pin1, LOW);
                    digitalWrite(motor2Pin2, HIGH);
                } else if (header.indexOf("GET /stop") >= 0) {
                    Serial.println("Stop");
                    digitalWrite(motor1Pin1, LOW);
                    digitalWrite(motor1Pin2, LOW);
                    digitalWrite(motor2Pin1, LOW);
                    digitalWrite(motor2Pin2, LOW);
                } else if (header.indexOf("GET /right") >= 0) {
                    Serial.println("Right");
                    digitalWrite(motor1Pin1, LOW);
                    digitalWrite(motor1Pin2, HIGH);
                    digitalWrite(motor2Pin1, LOW);
                    digitalWrite(motor2Pin2, LOW);
                }
            }
        }
    }
}

```

```

    } else if (header.indexOf("GET /reverse") >= 0) {
        Serial.println("Reverse");
        digitalWrite(motor1Pin1, HIGH);
        digitalWrite(motor1Pin2, LOW);
        digitalWrite(motor2Pin1, HIGH);
        digitalWrite(motor2Pin2, LOW);
    }
    // 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 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 { -webkit-user-select: none; -moz-user-
select: none; -ms-user-select: none; user-select: none; background-color:
#4CAF50;}");
        client.println("border: none; color: white; padding: 12px 28px;
text-decoration: none; font-size: 26px; margin: 1px; cursor: pointer;");
    client.println(".button2                                {background-color:
#555555;}</style></head>");
    // Web Page
    client.println("<p><a href=\"/forward\"><button class=\"button\"
onclick=\"moveForward()\">FORWARD</button></a></p>");
    client.println("<div                                style=\"clear:
both;\"><p><a
href=\"/left\"><button                                class=\"button\"
onclick=\"moveLeft()\">LEFT
</button></a>");
        client.println("<a                                href=\"/stop\"><button
class=\"button
button2\" onclick=\"stopRobot()\">STOP</button></a>");
    client.println("<a                                href=\"/right\"><button
class=\"button\"
onclick=\"moveRight()\">RIGHT</button></a></p></div>");
    client.println("<p><a href=\"/reverse\"><button class=\"button\"
onclick=\"moveReverse()\">REVERSE</button></a></p>");
    client.println("<input type=\"range\" min=\"0\" max=\"100\"
step=\"25\" id=\"motorSlider\" onchange=\"motorSpeed(this.value)\" value=\""
+ valueString + "\"/>");
    client.println("<script> function motorSpeed(pos) { ");
    client.println("var xhr = new XMLHttpRequest();");
    client.println("xhr.open('GET', \"/?value=\"" + pos + "\"&\",
true);");
    client.println("xhr.send(); } </script>");

    client.println("</html>");

    //Request example: GET /?value=100& HTTP/1.1 - sets PWM duty
cycle to 100
    if(header.indexOf("GET /?value=")>=0) {
        pos1 = header.indexOf('=');
        pos2 = header.indexOf('&');
        valueString = header.substring(pos1+1, pos2);
        //Set motor speed value
        if (valueString == "0") {
            ledcWrite(pwmChannel, 0);

```

```

        digitalWrite(motor1Pin1, LOW);
        digitalWrite(motor1Pin2, LOW);
        digitalWrite(motor2Pin1, LOW);
        digitalWrite(motor2Pin2, LOW);
    }
    else {
        dutyCycle = map(valueString.toInt(), 25, 100, 200, 255);
        ledcWrite(pwmChannel, dutyCycle);
        Serial.println(valueString);
    }
}
// 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("");
}
}

```

Customize the 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 actually set the ESP32 as an Access Point:

```

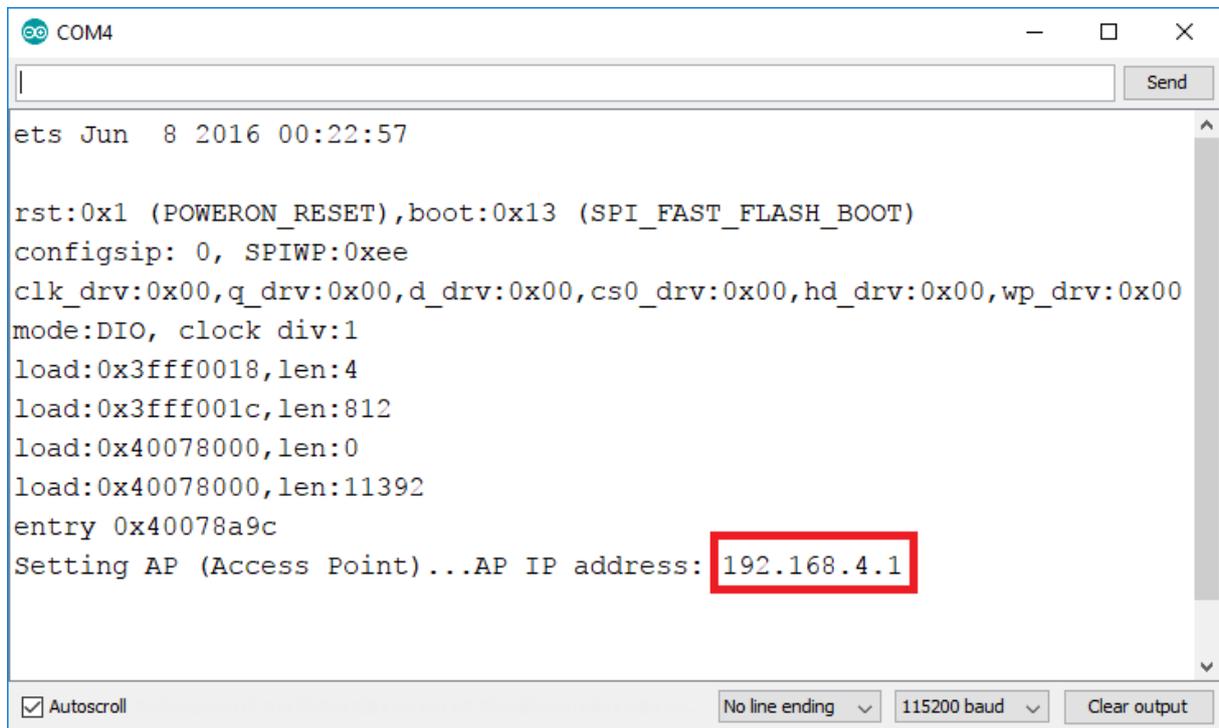
// Connect to Wi-Fi network with SSID and password
Serial.print("Setting AP (Access Point)...");
// Remove the password parameter, if you want the AP (Access Point) to be
open
WiFi.softAP(ssid, password);

IPAddress IP = WiFi.softAPIP();
Serial.print("AP IP address: ");
Serial.println(IP);
server.begin();

```

The rest of the code and the web page features are very 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:



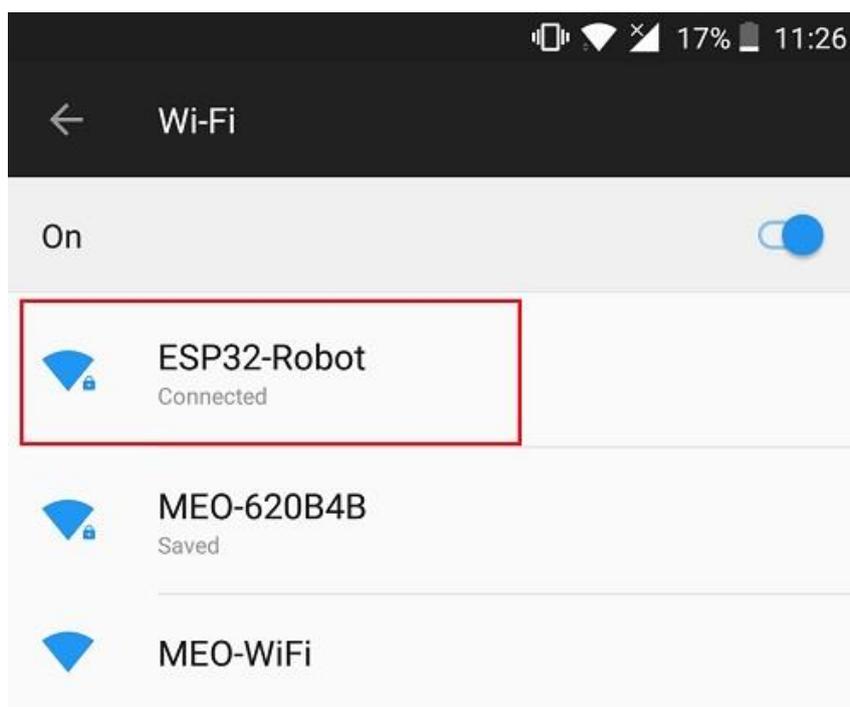
```
ets Jun  8 2016 00:22:57

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:0x3fff0018,len:4
load:0x3fff001c,len:812
load:0x40078000,len:0
load:0x40078000,len:11392
entry 0x40078a9c
Setting AP (Access Point)...AP IP address: 192.168.4.1
```

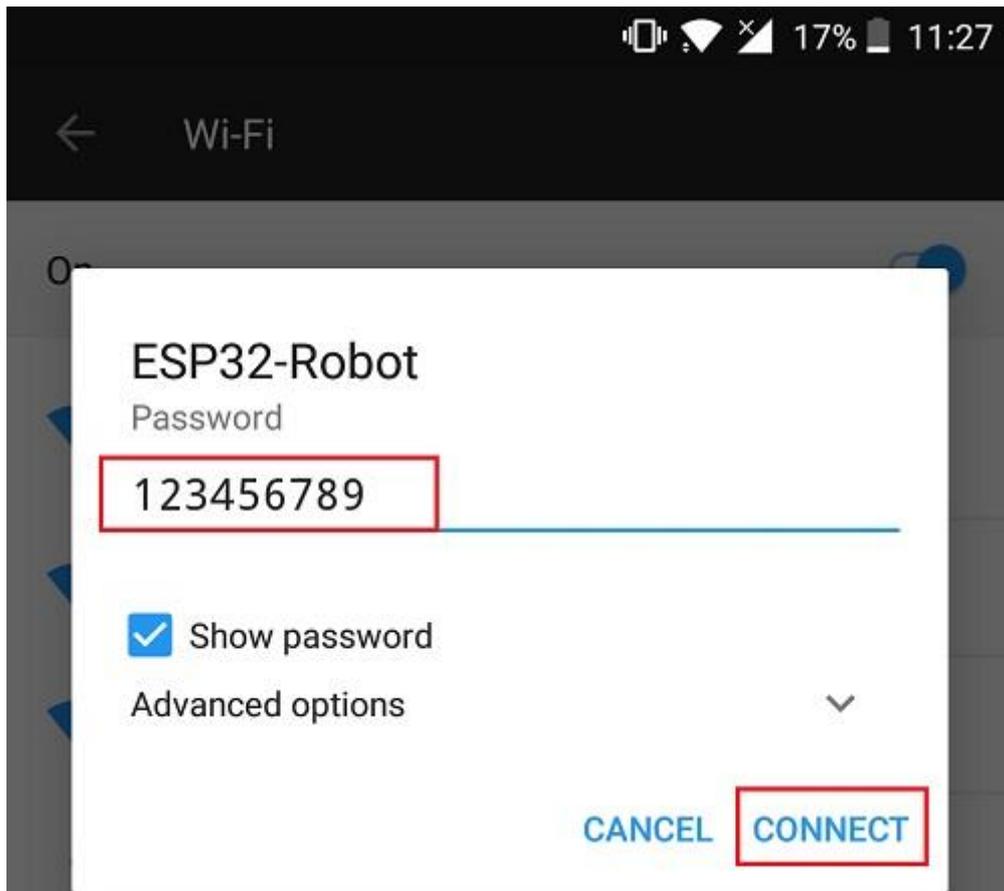
In my case it's **192.168.4.1**, save that IP, because you'll need it later.

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.

The only differences in this web page:

- The slider doesn't show the values on the web page;
- The web page is refreshed every time you press the buttons and make an HTTP request.



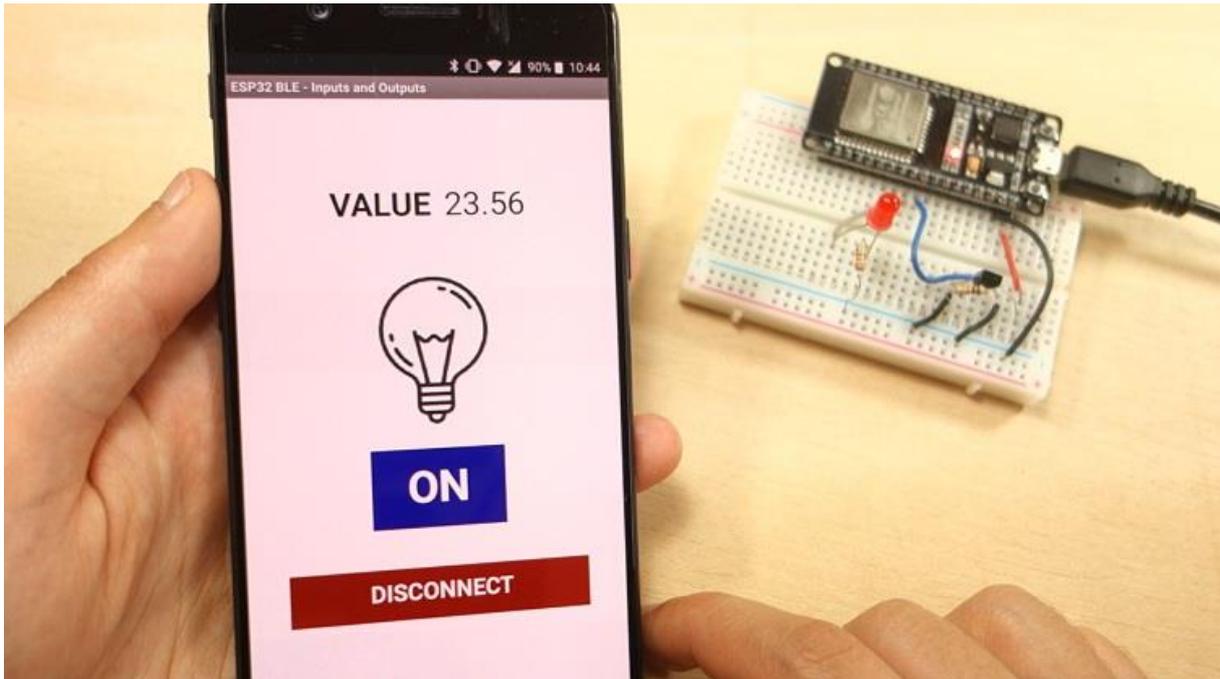
PROJECT 3

ESP32 BLE Android Application

Unit 1 - ESP32 BLE Android

Application: Control Outputs and Display Sensor Readings

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



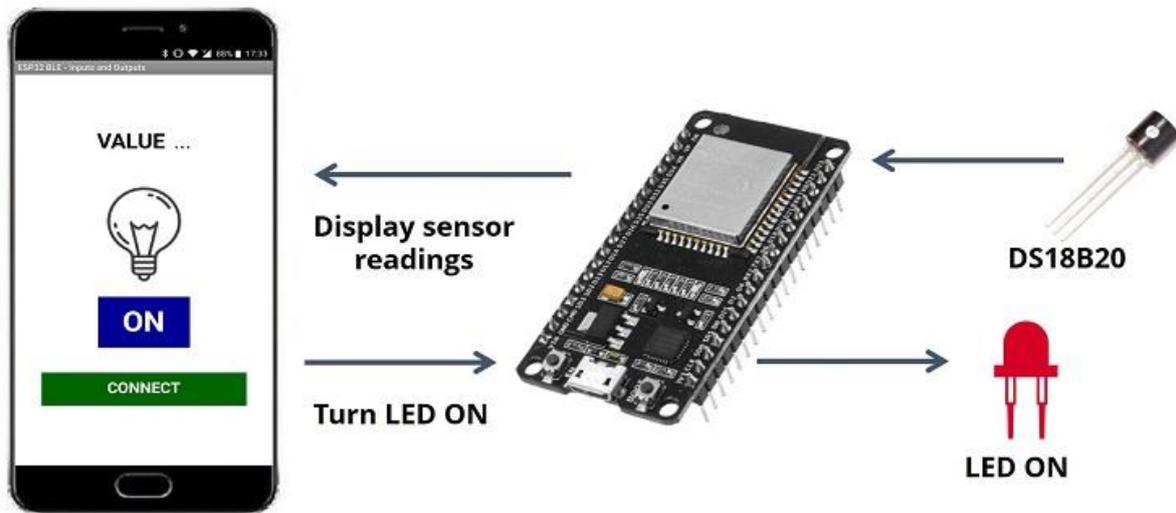
There are three steps in this project:

- 1) First, building the circuit;
- 2) Then, uploading the code to the ESP32;
- 3) And finally, installing the Android app provided.

Project Overview

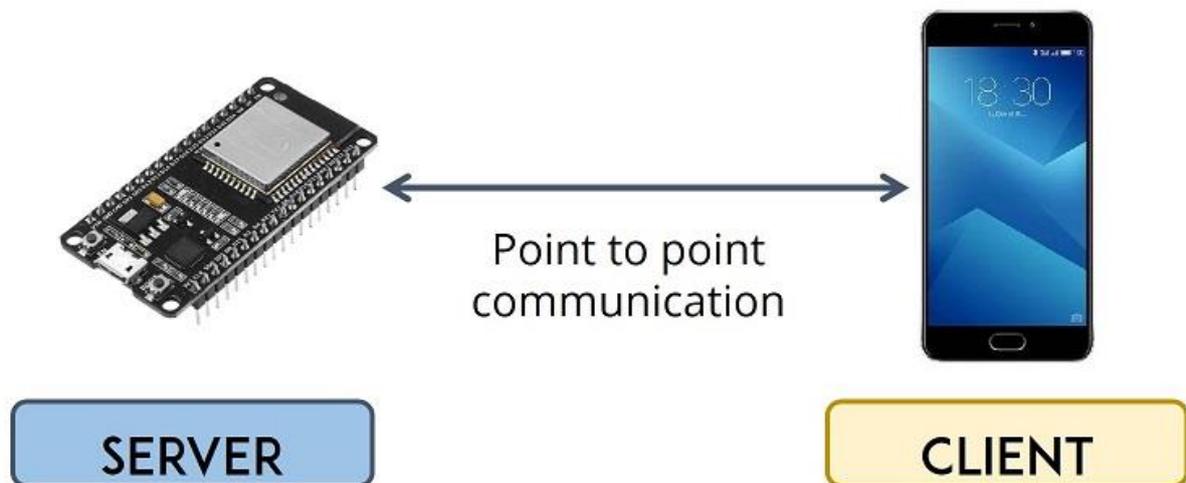
Let's take a look at the project's main features:

- 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 and they will be establishing a point-to-point communication;



The ESP32, as the server, advertises its existence, so that it can be found by other devices, 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. In this video we won't cover how to build the Android app, but we provide an additional Unit that explains in more detail how the app was created.

Installing Libraries

Before uploading the code, you need to install two libraries in your Arduino IDE. The [OneWire library by Paul Stoffregen](#) and the [Dallas Temperature library](#), so that you can use the DS18B20 sensor. Follow the next steps to install those libraries.

OneWire library

- 1) [Click here to download the OneWire library](#). You should have a .zip folder in your Downloads
- 2) Unzip the .zip folder and you should get OneWire-master folder
- 3) Rename your folder from ~~OneWire-master~~ to OneWire
- 4) Move the OneWire folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Dallas Temperature library

- 1) [Click here to download the DallasTemperature library](#). You should have a .zip folder in your Downloads
- 2) Unzip the .zip folder and you should get Arduino-Temperature-Control-Library-master folder
- 3) Rename your folder from ~~Arduino-Temperature-Control-Library-master~~ to DallasTemperature
- 4) Move the DallasTemperature folder to your Arduino IDE installation libraries folder
- 5) Finally, re-open your Arduino IDE

Uploading the code

After installing the required libraries, copy the following code to your Arduino IDE.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_BLE_Inputs_and_Outputs/ESP32_BLE_Inputs_and_Outputs.ino

```
// 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) {
        std::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.find("ON") != -1) {
                Serial.println(" - LED ON");
                digitalWrite(ledPin, HIGH);
            }
            else if(rxValue.find("OFF") != -1) {
                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 the Code Works

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 that will be sent to the client, in this case to the Android application.

```
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 temperature sensor.

```
Serial.begin(115200);  
pinMode(ledPin, OUTPUT);  
sensors.begin();
```

BLE device

Create a new BLE device with a name that allows you clearly identify your ESP32 board, for example **ESP32_Board**.

```
BLEDevice::init("ESP32_Board");
```

BLE server

Create a BLE server.

```
BLEServer *pServer = BLEDevice::createServer();
```

Set callbacks for the server and characteristic: the `MyServerCallbacks()` function and the `MyCallbacks()` function::

```
pServer->setCallbacks(new MyServerCallbacks());
```

```
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 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) {
    std::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.find("ON") != -1) {
            Serial.println(" - LED ON");
            digitalWrite(ledPin, HIGH);
        }
        else if(rxValue.find("OFF") != -1) {
            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 the `rxValue` variable. Then, accordingly to the value received, whether it's a ON command or OFF command, it turns the LED on or off.

```
if(rxValue.find("ON") != -1) {
    Serial.println(" - LED ON");
    digitalWrite(ledPin, HIGH);
}
else if(rxValue.find("OFF") != -1) {
    Serial.println(" - LED OFF");
    digitalWrite(ledPin, LOW);
}
```

Start BLE

After creating the server, 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

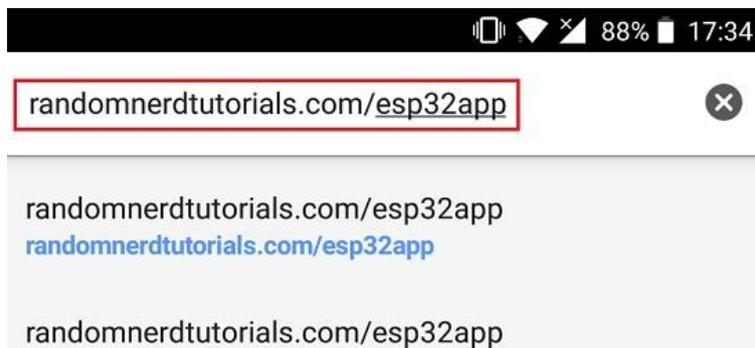
Let's move on to the Android application. As we've mentioned previously, this app was built using [MIT App Inventor 2](#). You can download the .apk and .aia files:

- [Android application .apk file](#)
- [Edit the app with .aia file on MIT App Inventor](#)

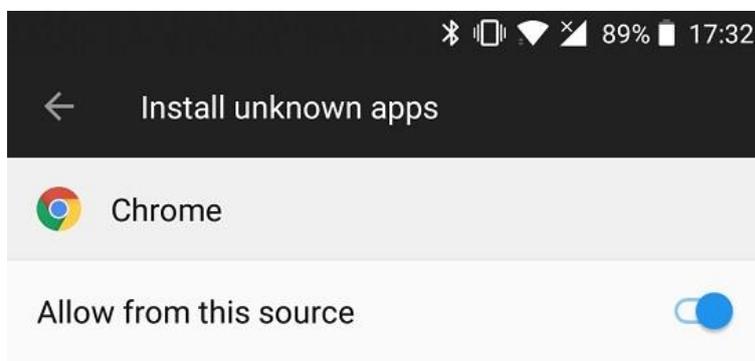
To install the app in your smartphone, copy the following link to your smartphone to download the .apk file:

- <https://RandomNerdTutorials.com/esp32app>

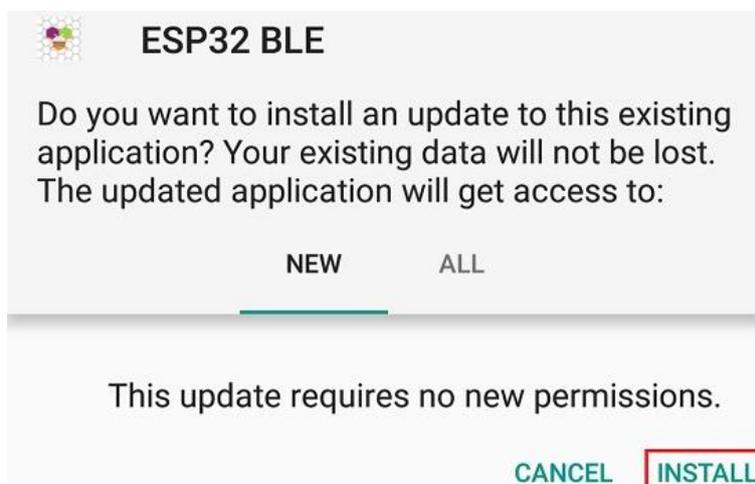
Alternatively, simply move the .apk file downloaded earlier to your smartphone.



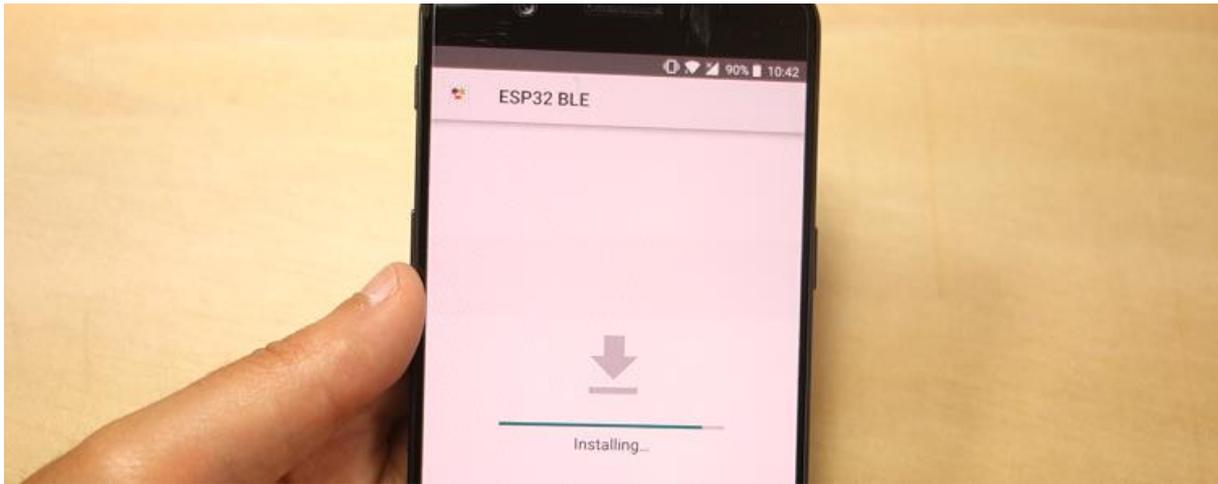
In your smartphone, you need to open the downloaded file and allow applications from this source to be installed:



Follow the installation wizard to to install the app.



It should take a few seconds to install...

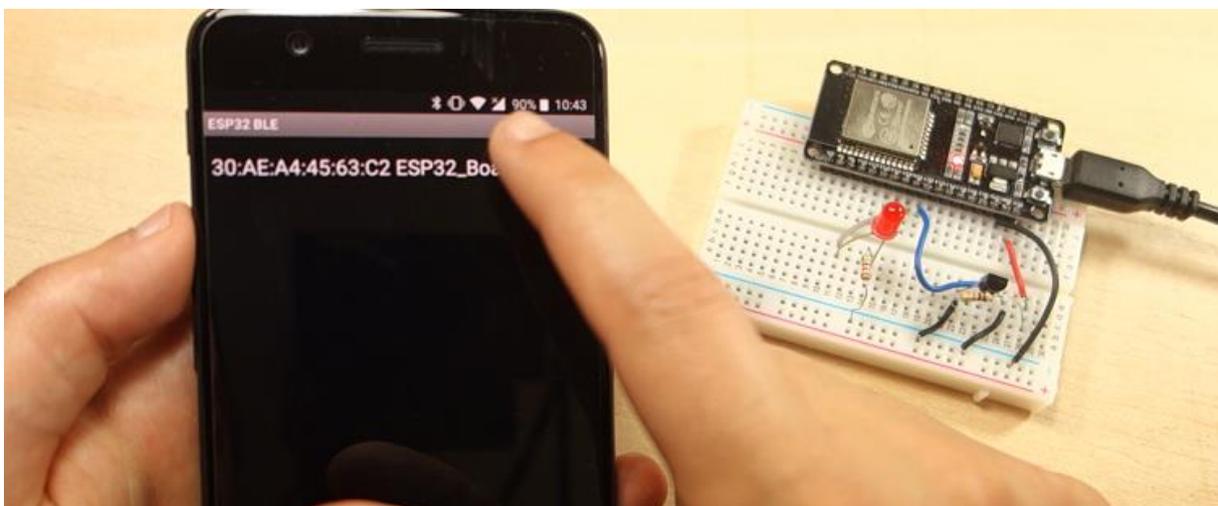


If everything went as expected the app icon should be in your home screen (or you might have to search for the app "ESP32 BLE").

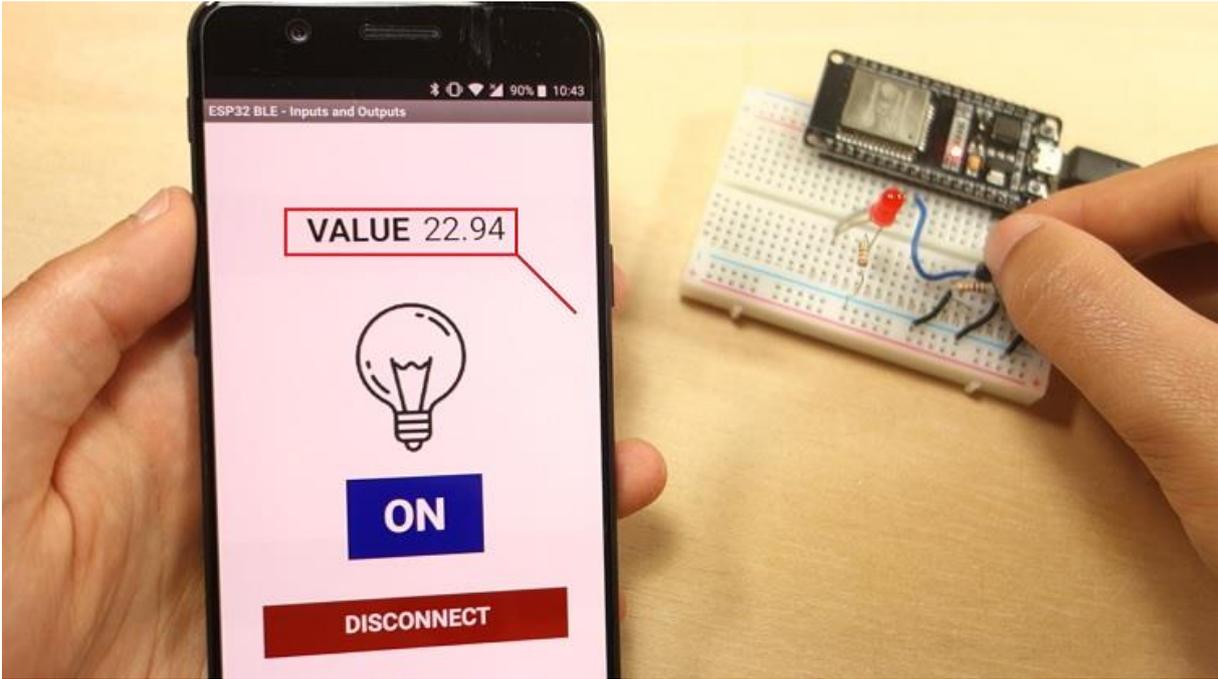


Demonstration

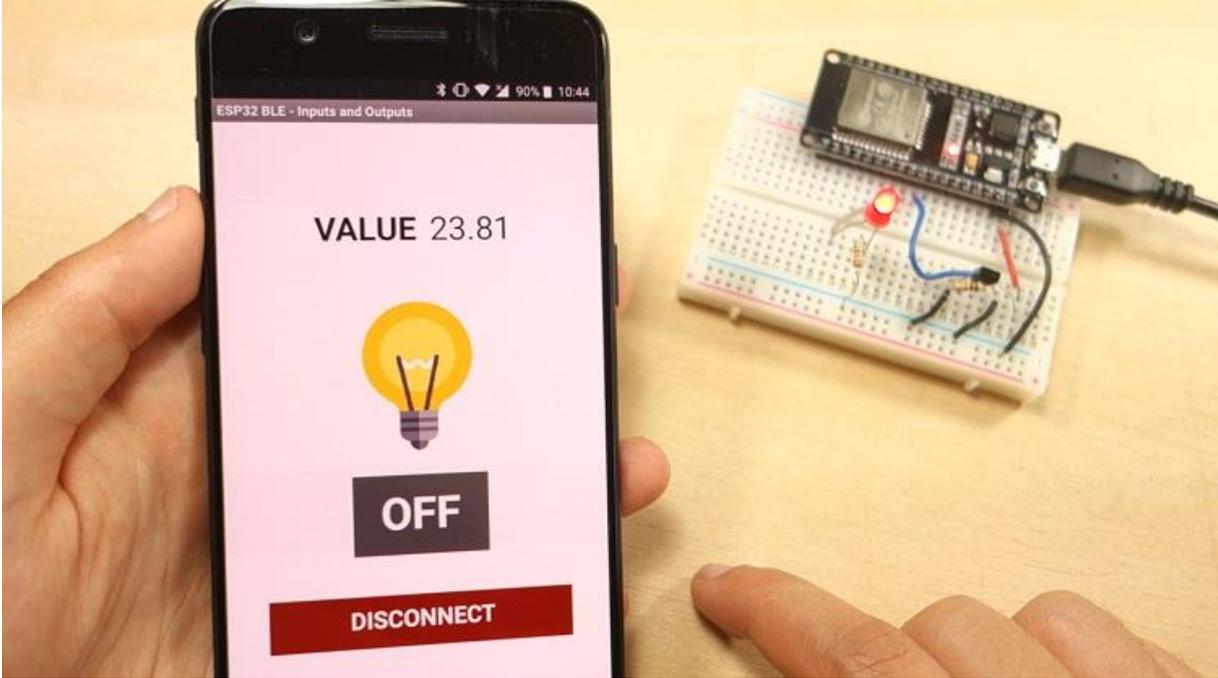
At this point everything should be ready. Let's test the project. On your smartphone enable Bluetooth, open the Android app and tap the "CONNECT" button to search for nearby devices. Select the ESP32_Board device, and it should connect within a few seconds.



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 to 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 in 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.

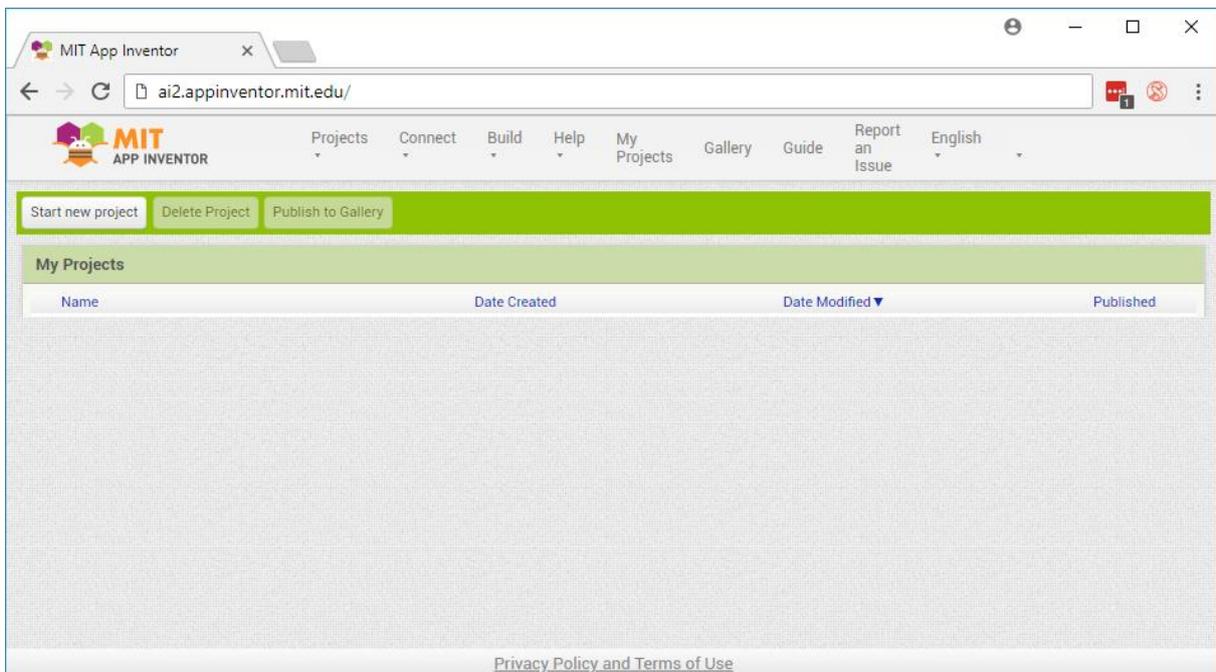
Creating the design is as easy as selecting and placing components in the smartphone screen. The coding is done with drag and drop puzzle blocks. Anyone can learn how to build their own apps with MIT App Inventor 2 with a few hours of practice.

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.

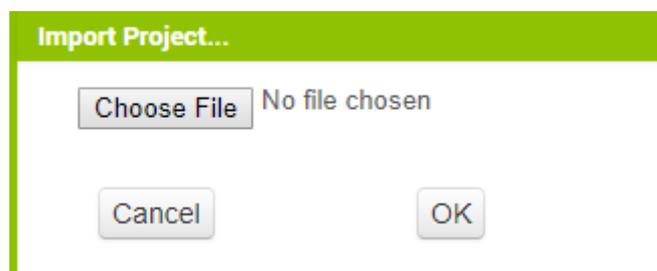


To access the app builder, you need a Google account. Follow the on-screen steps to login into MIT App Inventor 2. After that, you'll be presented with the following dashboard:



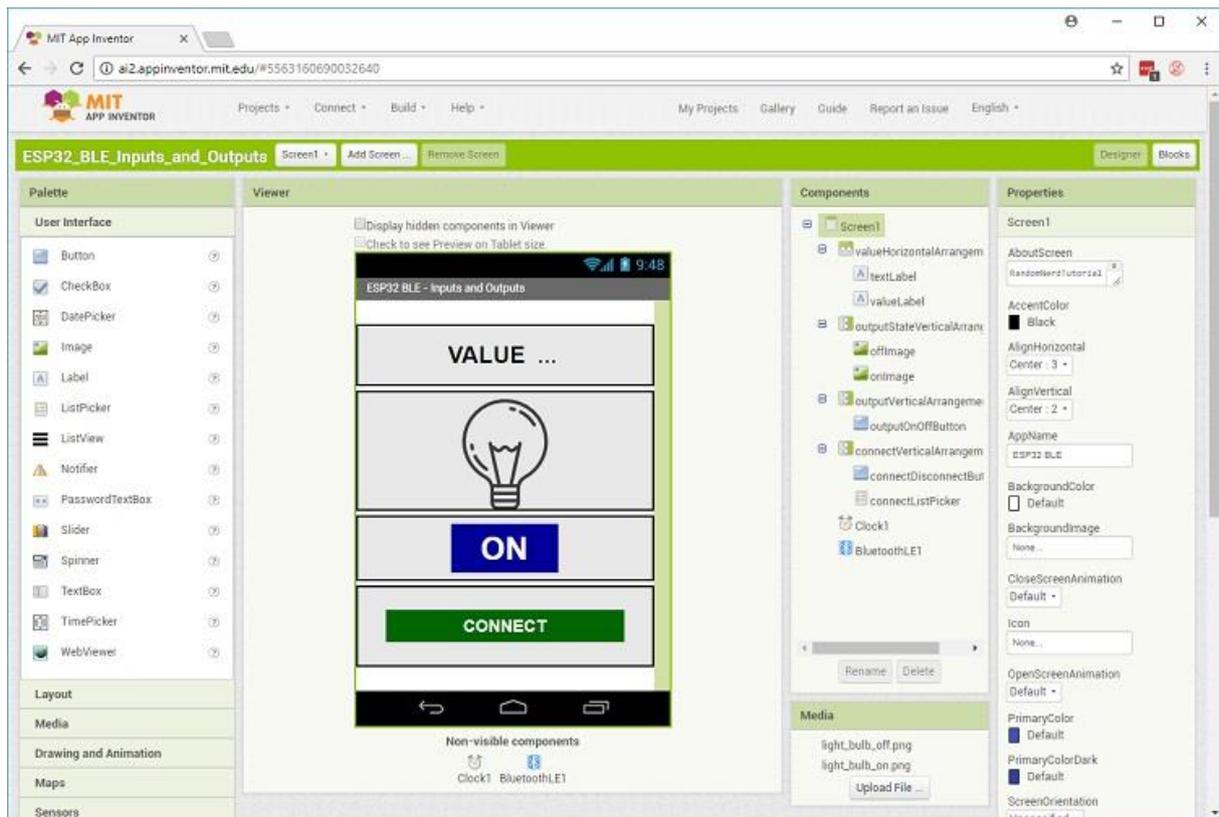
Importing the Android App File

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.



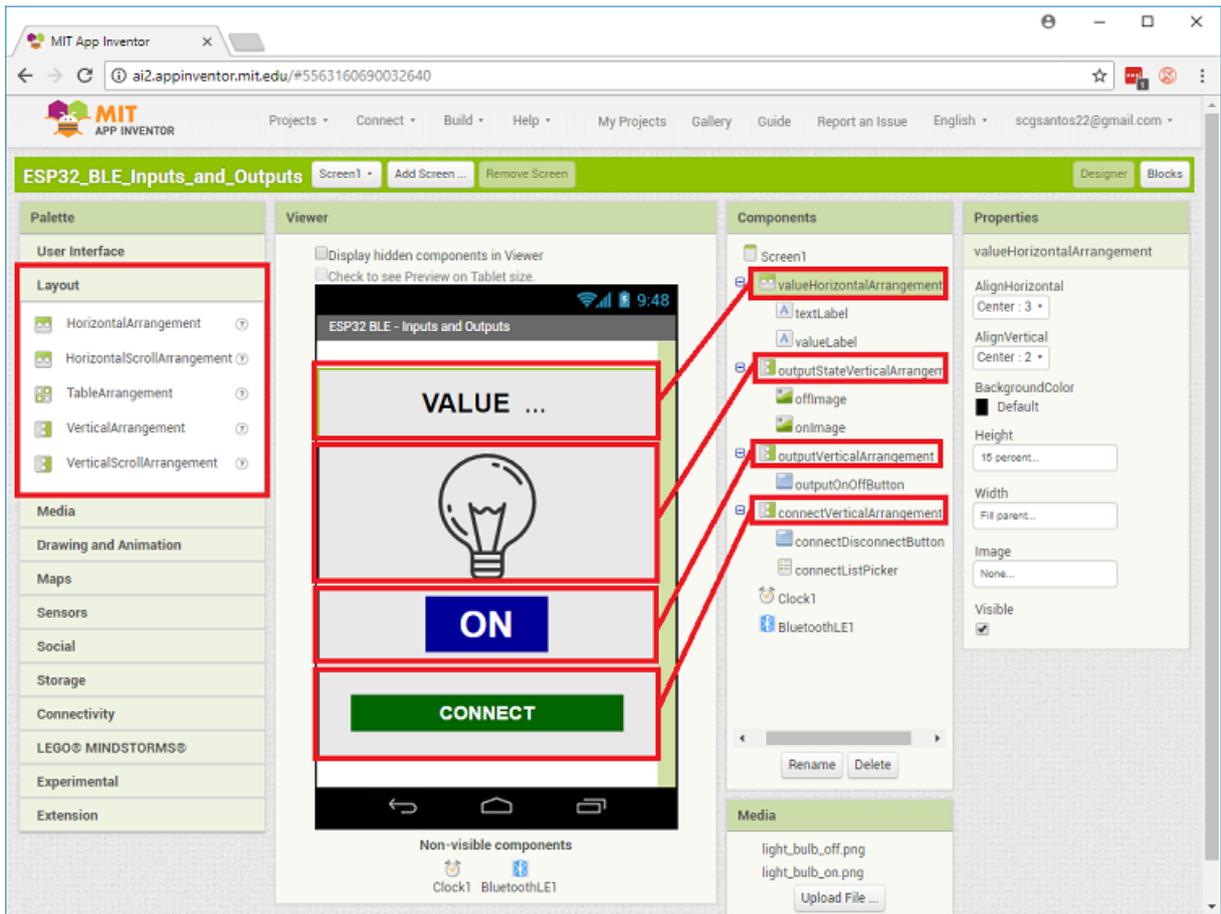
This is the **Designer** tab. The Designer tab gives you the ability to add buttons, text, images, sliders, etc... It allows you to change the overall app look. In the Designer tab, you have several sections:

- **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.

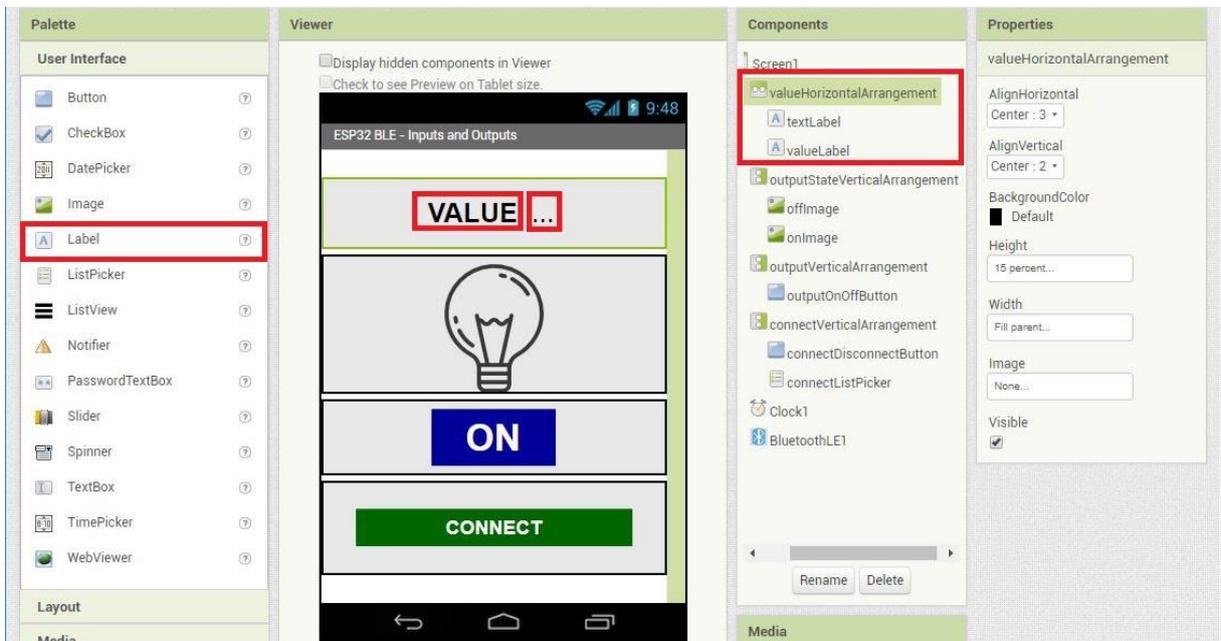
There are several arrangements to place the components. The image below shows the arrangements for each app component.



Now let's take a look at each individual component.

Text Labels

The first arrangement contains two text labels.



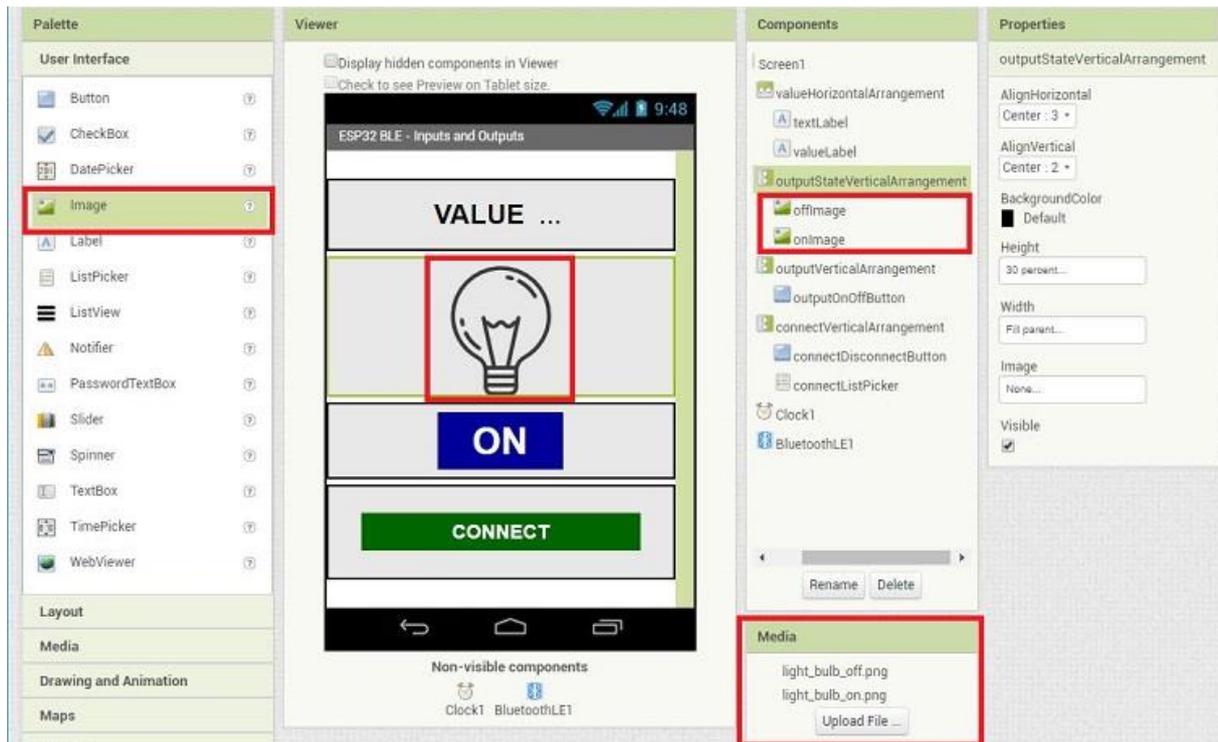
One text label contains the text "VALUE". We've defined the text as "VALUE" because instead of temperature you may want to display any other sensor readings. Modify that text to match the measurement you're displaying.

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 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

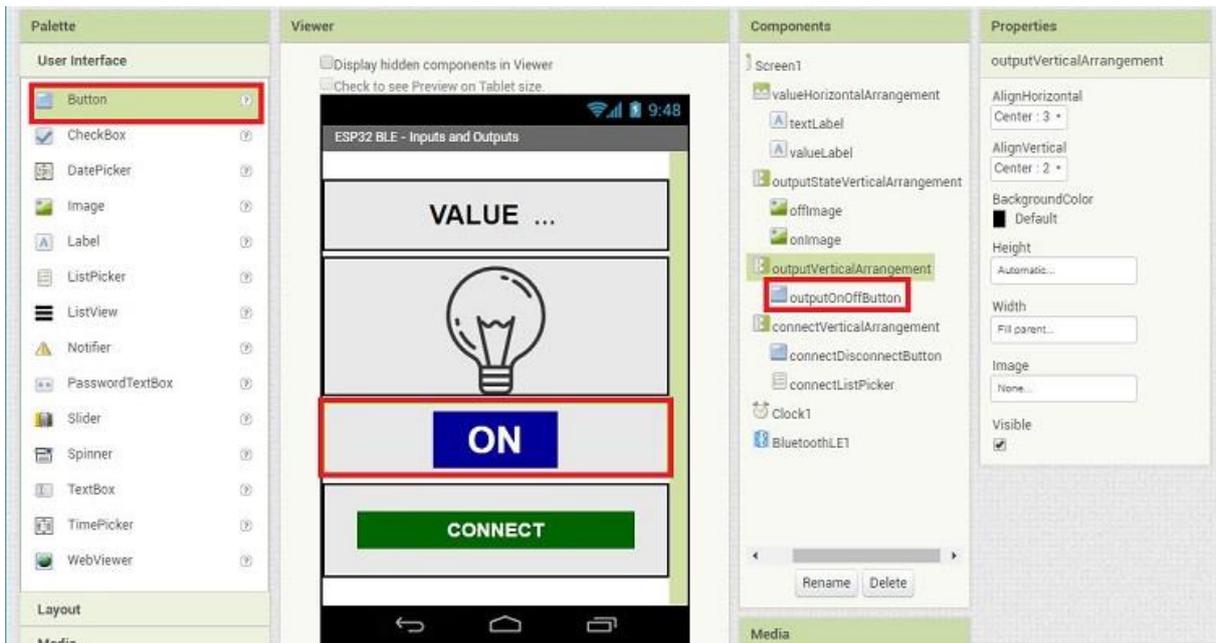
The next arrangement contains the lamp image.



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.

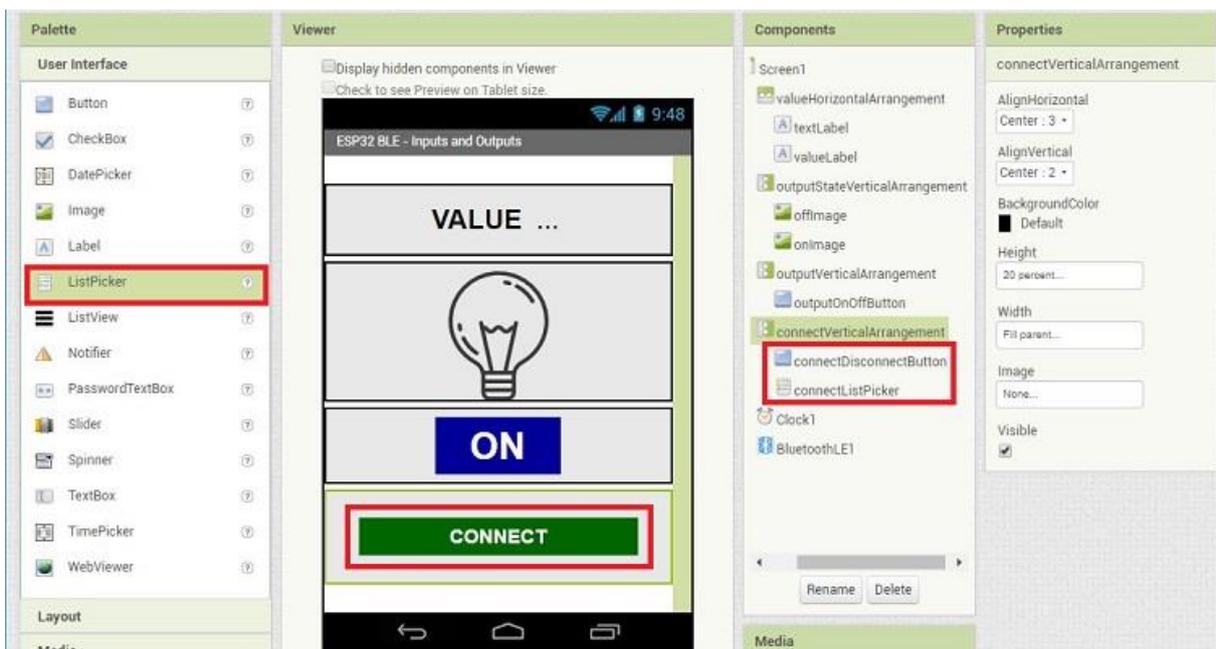
Button

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.

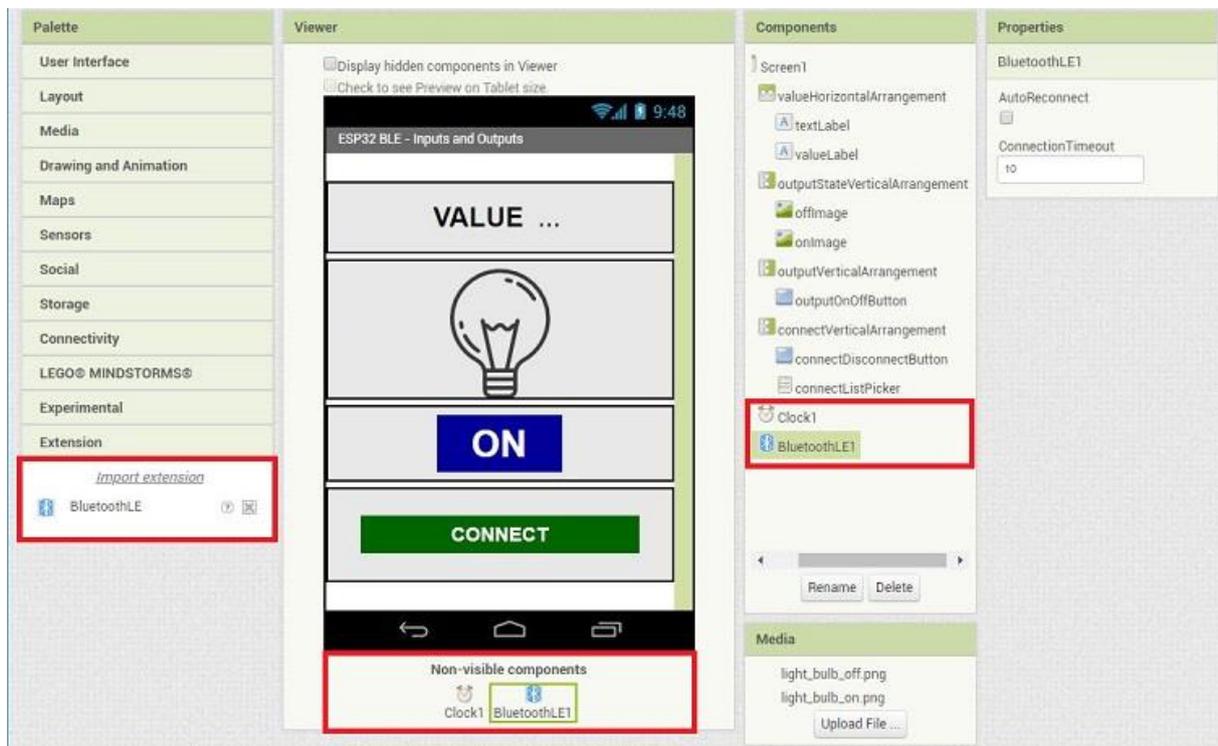
ListPicker



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.



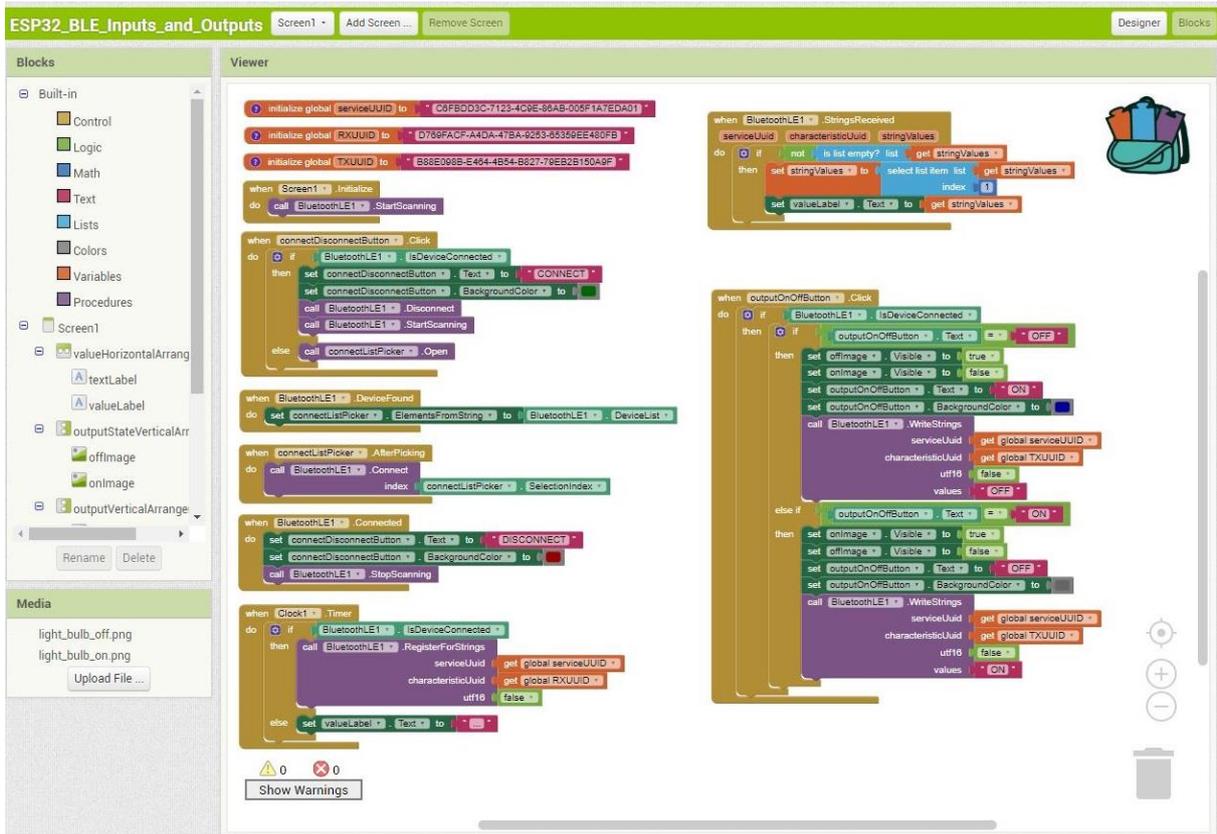
The MIT App Inventor 2 software doesn't support BLE by default. So, the BLE component shows up in the Extension section. Since the imported app contains a BluetoothLE component, it is automatically added to the extensions.

Exploring the App Logic

Now that we've explored the app design, let's take a 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

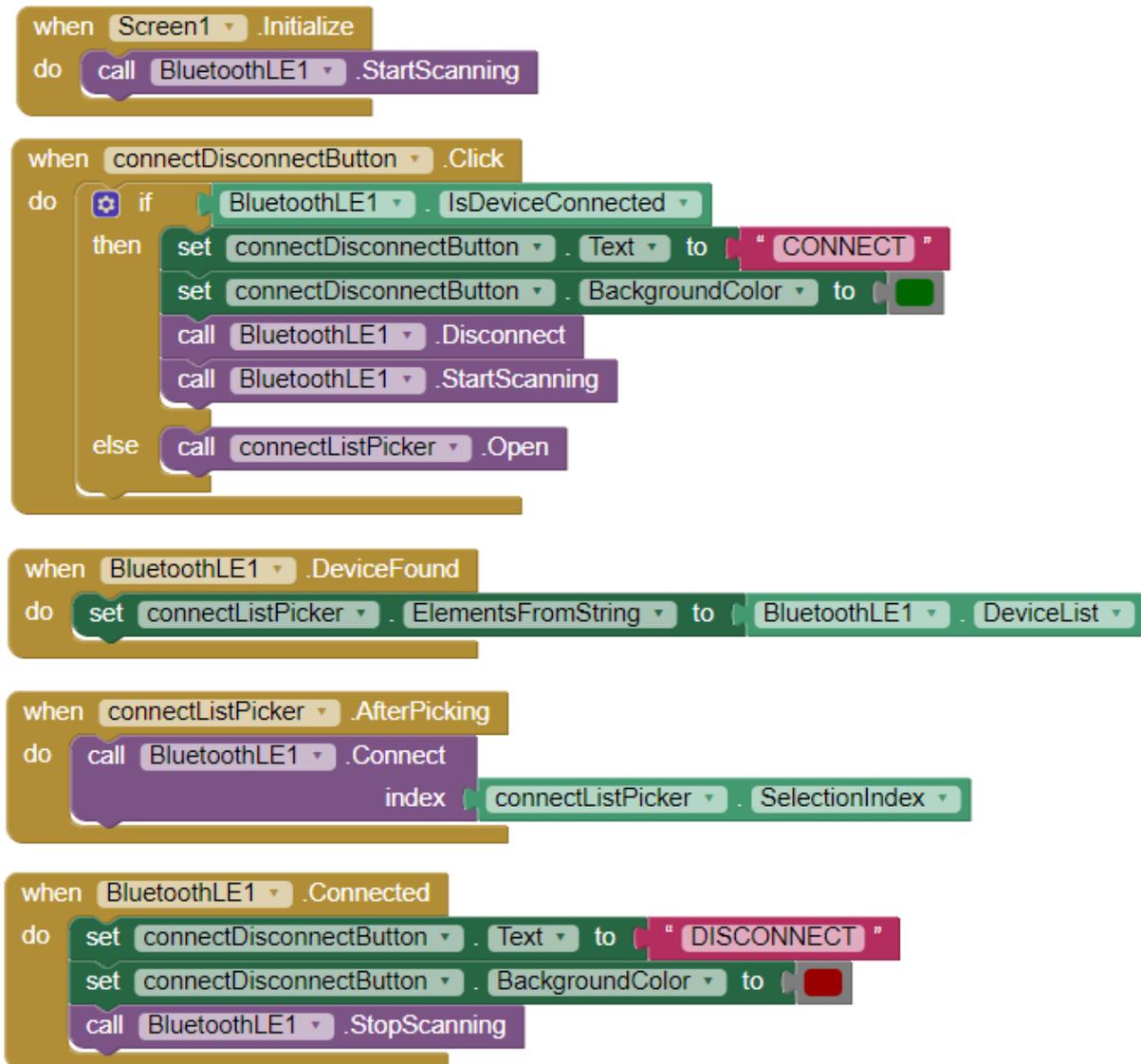
The following blocks define the UUIDs for the service, RX characteristic and TX characteristic.



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

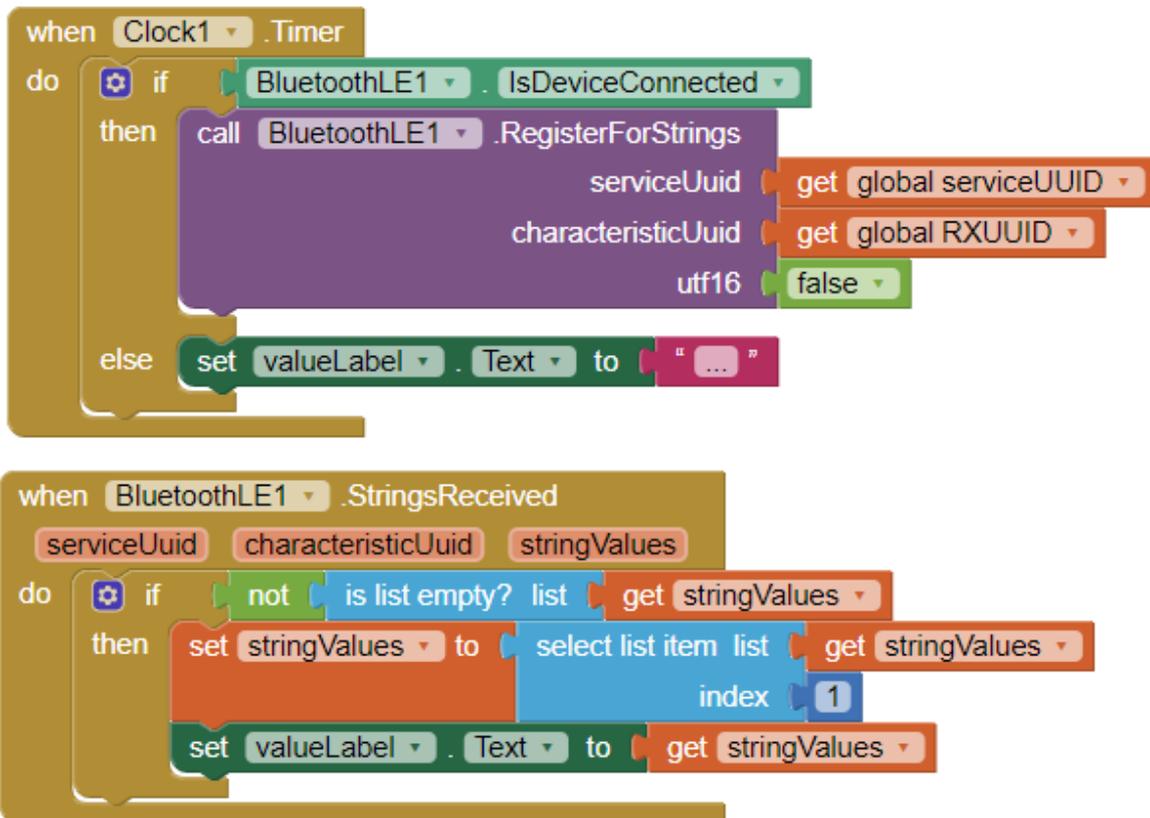
The next set of blocks is responsible for listing, picking and connecting to a BLE device.



This set also changes the button appearance depending 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.

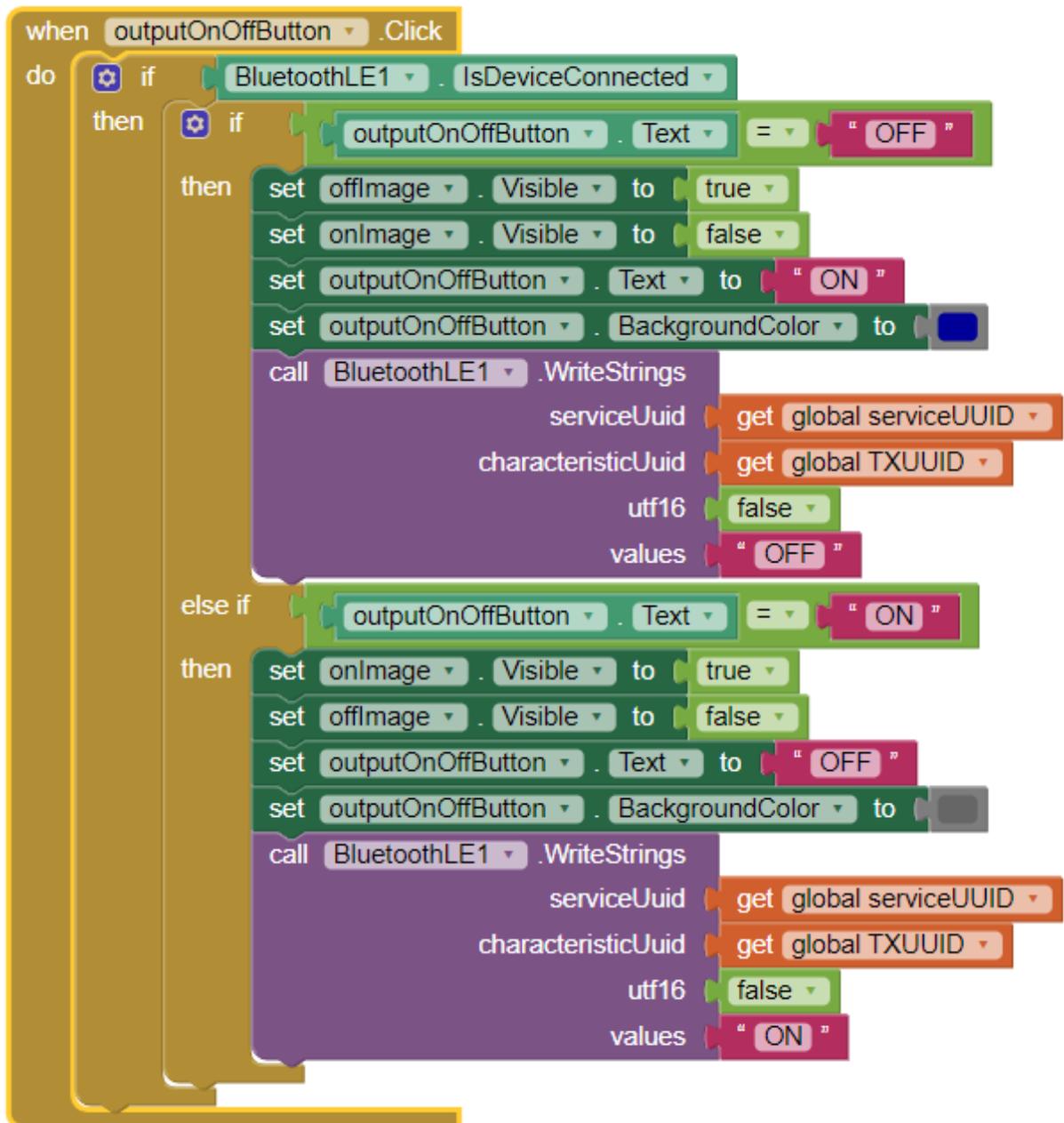


The first block starts the RX characteristic to receive new values or sets the temperature value to "...", if the client gets disconnected. The second block receives the message sent from the ESP32, saves it in the stringValues variable and displays it in the valueLabel.

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 is what 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.



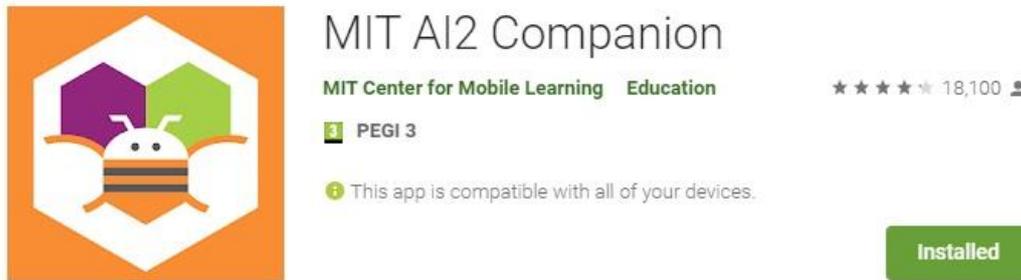
Basically, to send commands to the ESP32 you write “ON” or “OFF” in the TX characteristic. Here’s what happens when you click the “ON” button:

- 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 in your smartphone while editing in real time. You need to install the MIT AI2 Companion app in your smartphone. Go to Google Play Store and search for “MIT AI2 Companion” and install it.



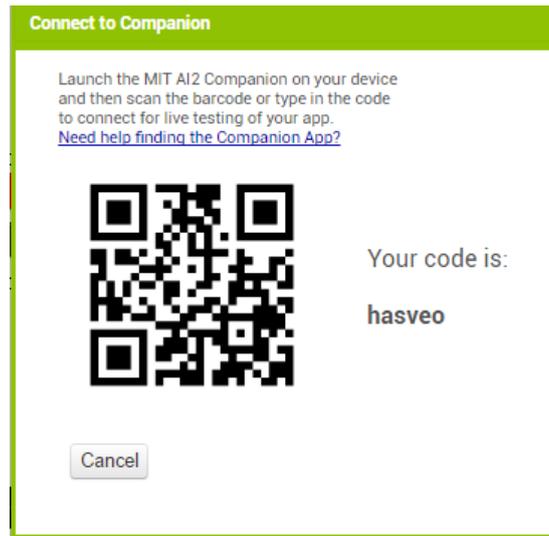
Open the app, you'll be presented with the following screen.



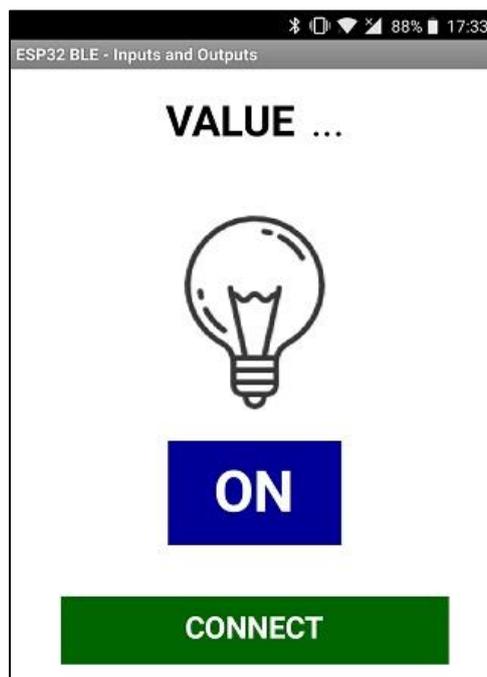
To test your app, in the MIT App Inventor 2 software, go to **Connect** ▶ **AI Companion**:



A QR code like the one in the following figure pops up:



In 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 on 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

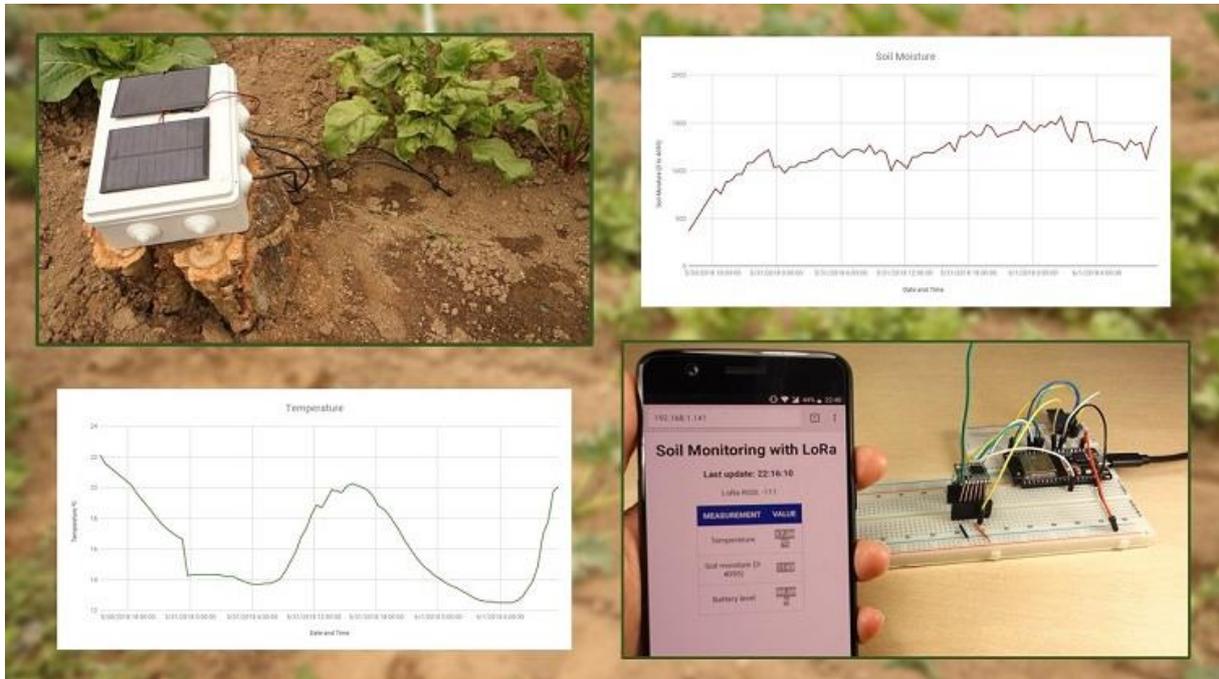
This is Part 1 of the project called LoRa Long Range Sensor Monitoring and Data Logging – Reporting Sensor Readings from Outside: Soil Moisture and Temperature.



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 will be using LoRa radio. The indoor LoRa receiver displays sensor readings on a web server and logs the data into a micro SD 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:** Module 6, Unit 1
- **ESP32 Web Server – Display Sensor Readings:** Module 4, Unit 8
- **ESP32 Deep Sleep – Timer Wake Up:** Module 3, Unit 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](#)

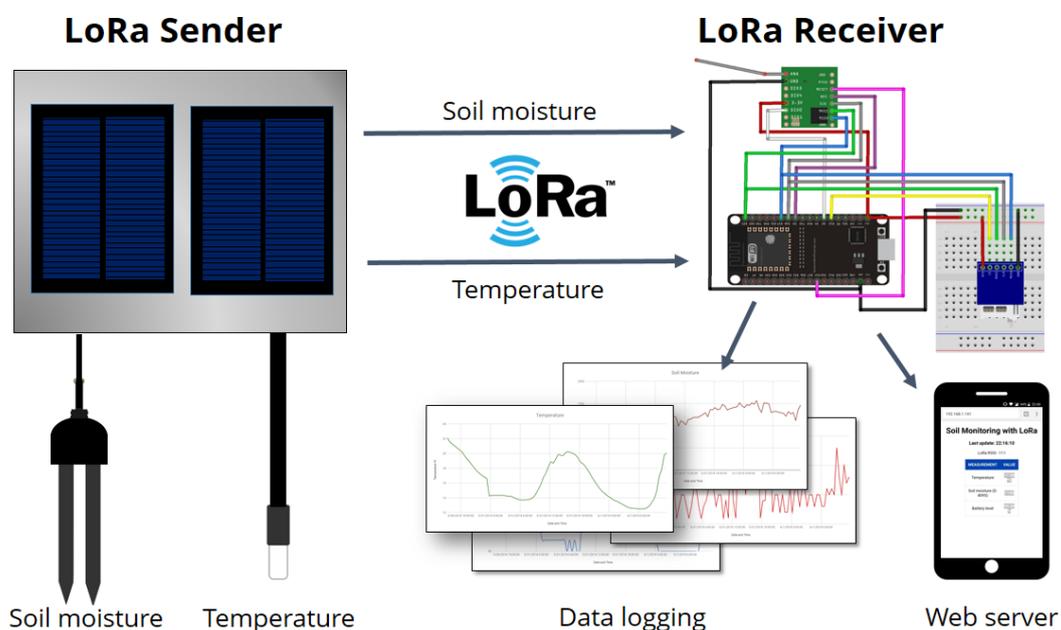
Useful tools for this project:

- [Soldering Iron](#)
- [Hot glue gun](#)
- [Multimeter](#)

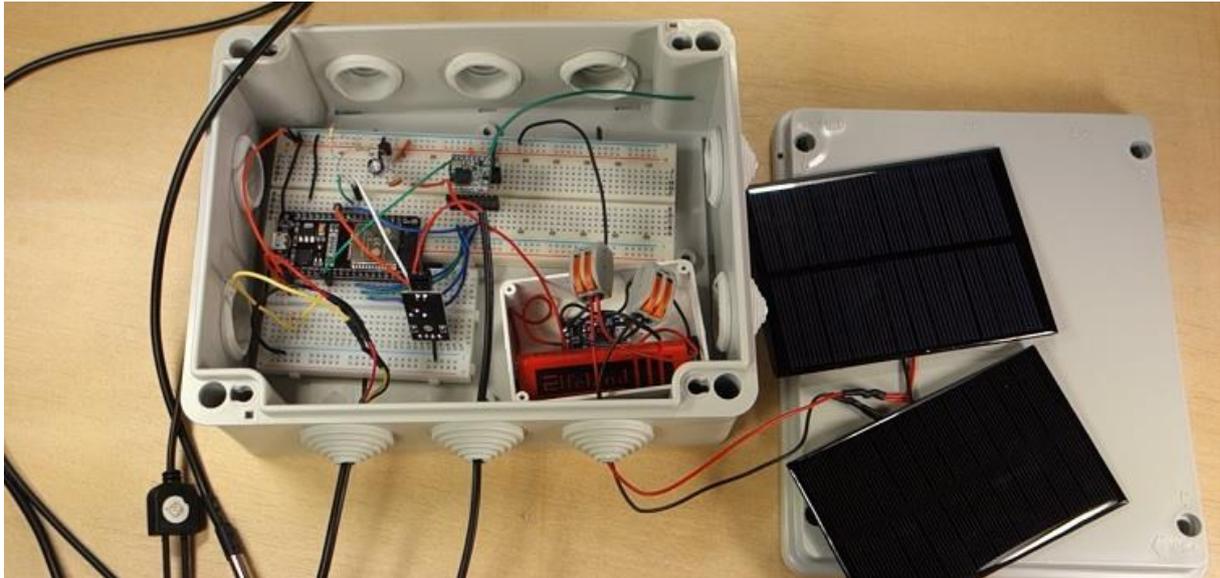
Project Overview

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 a 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.

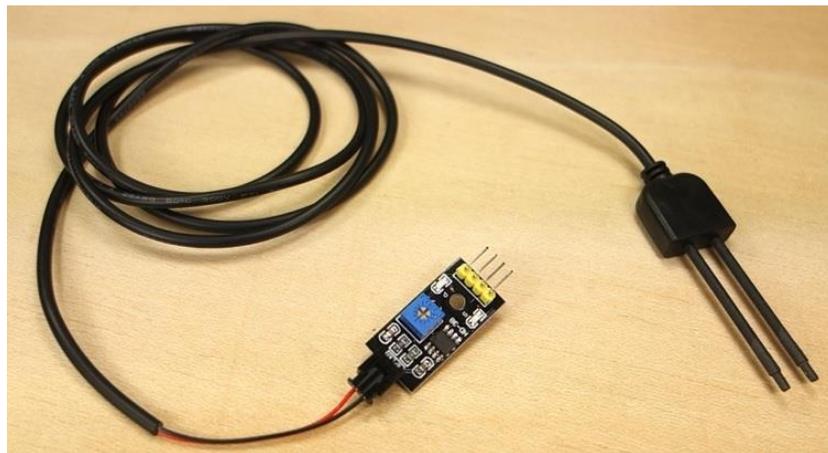


The LoRa Sender



Sensors

To read soil moisture we're using the Resistive Soil Moisture Sensor shown in the next figure.



To read temperature we're going to use the waterproof version of the DS18B20 temperature sensor.



Power

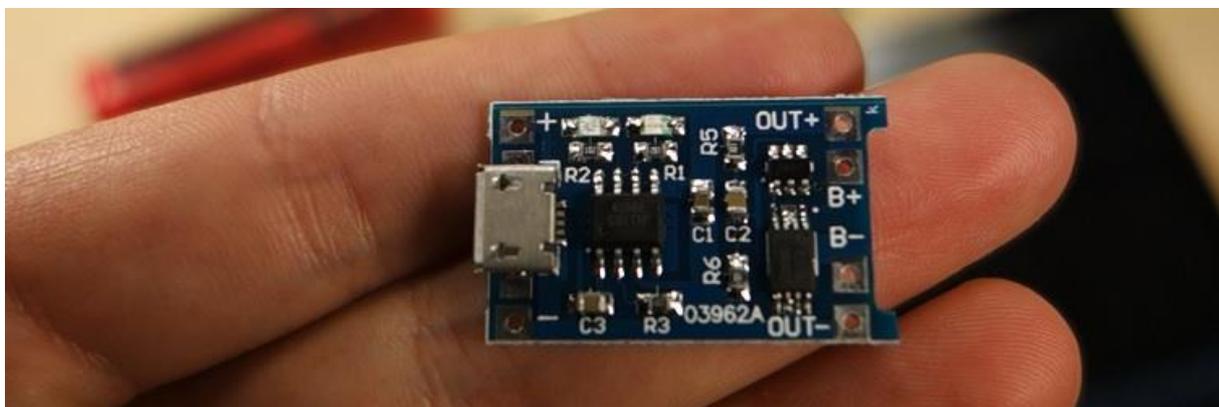
To power the circuit, we're going to use a rechargeable Lithium battery with 3800 mAh capacity.



The battery is charged using two mini 5V (1.2W) solar panels in combination with the TP4056 lithium battery charger module.



The TP4056 battery charger module is shown in the figure below.



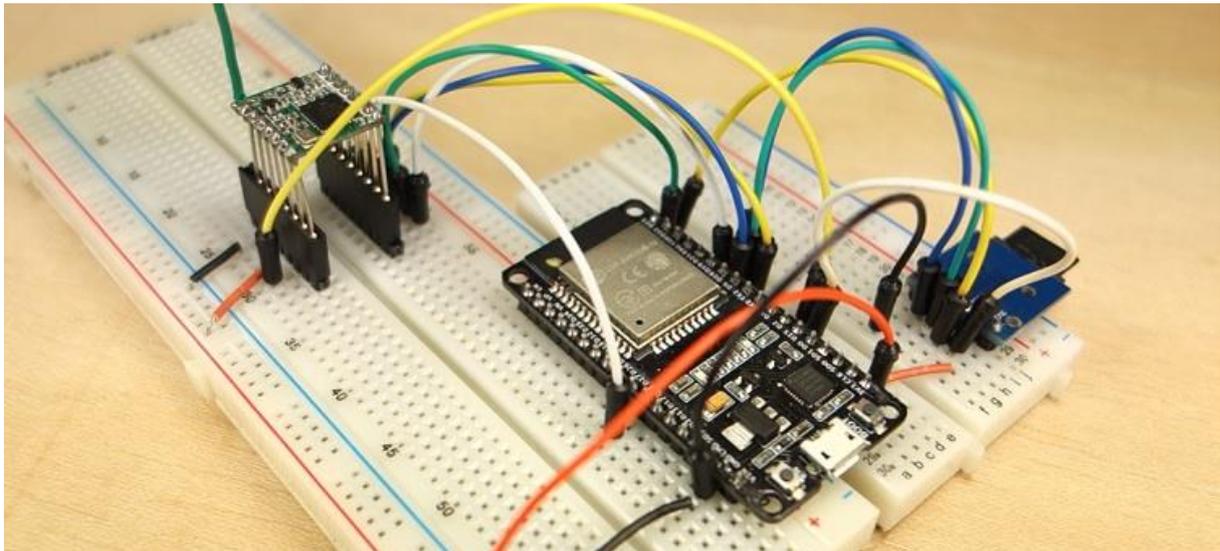
With this setup we have a self-sustainable LoRa sender node. We'll also be monitoring the battery level in each reading.

Deep Sleep

To save power, the ESP32 we'll 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.

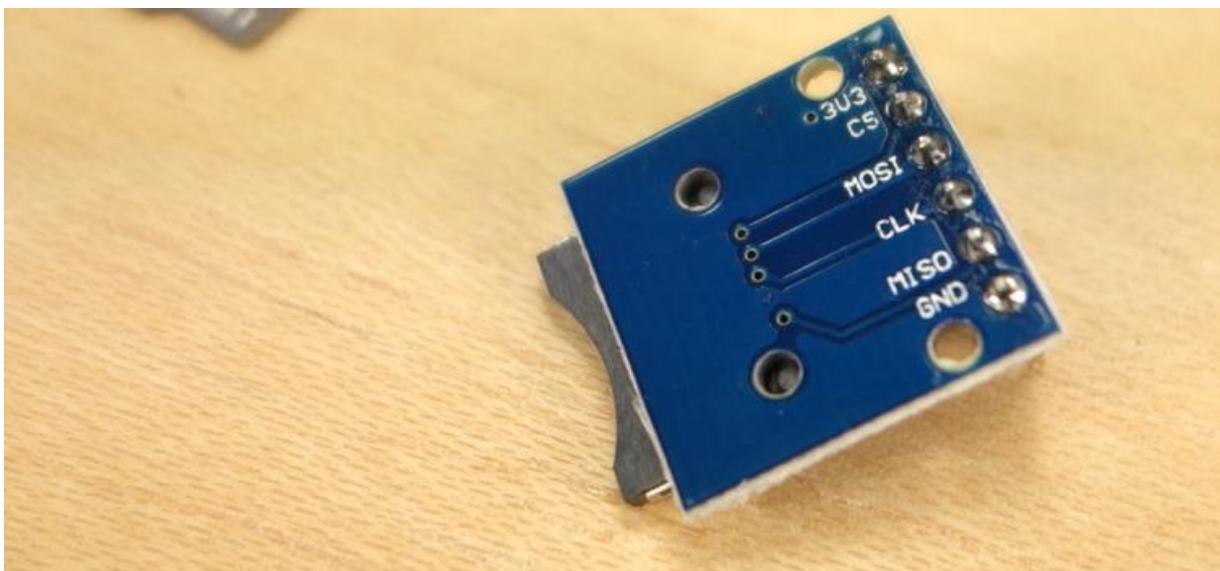


The LoRa Receiver



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.

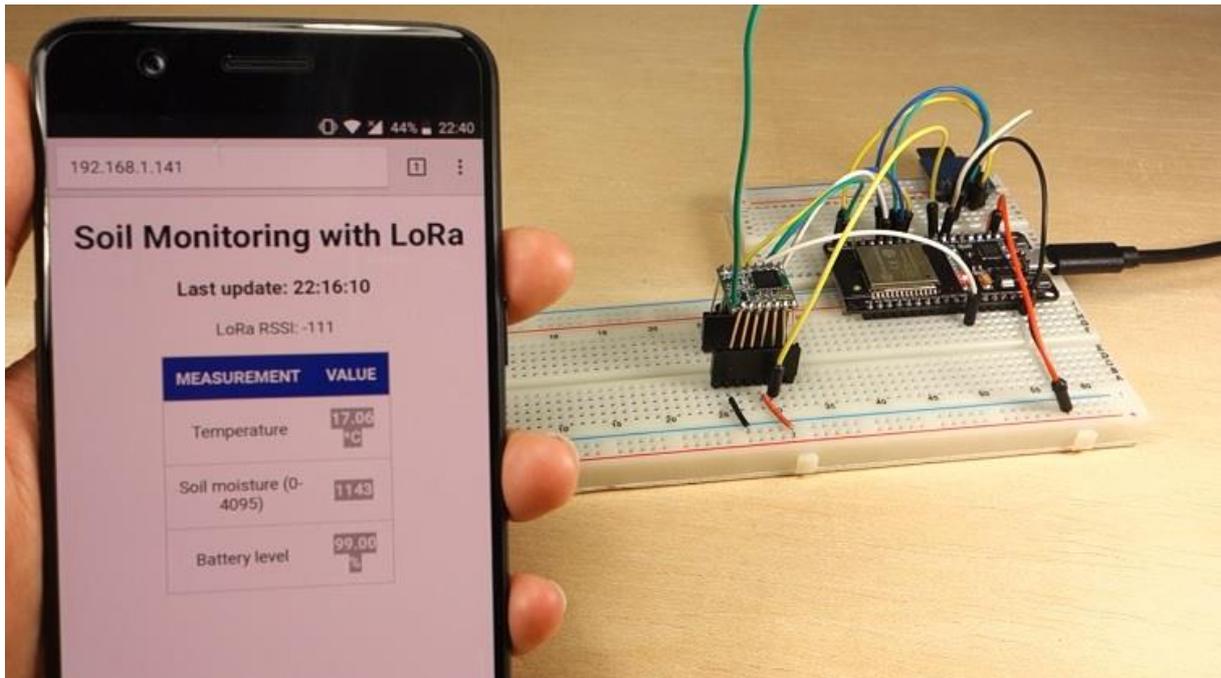


TimeStamps

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 gives us an idea of the LoRa signal's strength.



The web server should be only used occasionally to check the sensor readings.

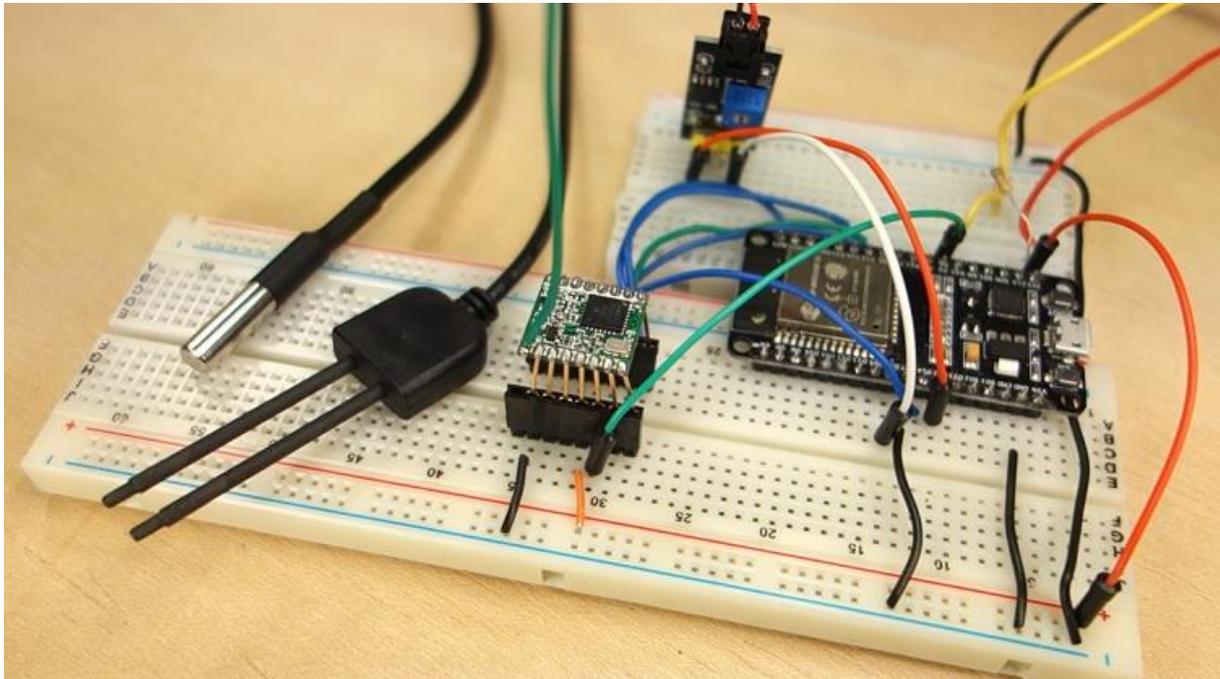
Important: after checking the readings on the web server, you must close the browser window, so that the receiver is able to get new LoRa packets.

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

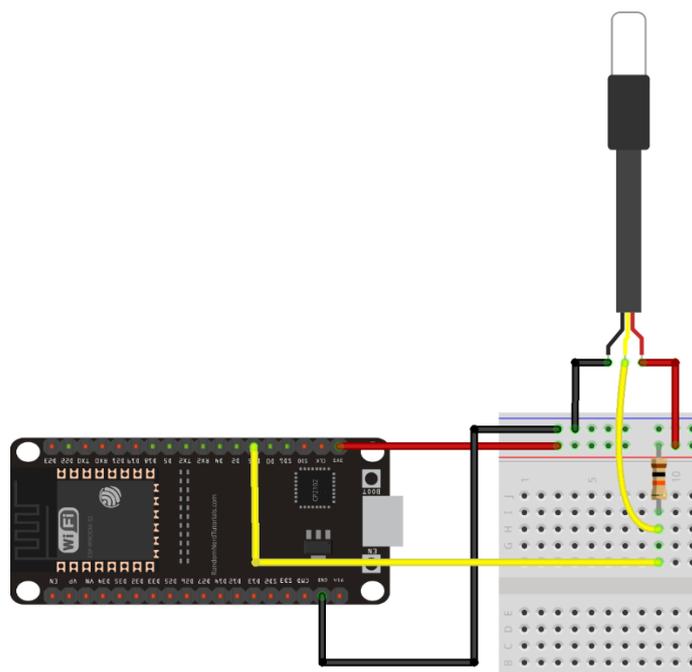
To read temperature we're going to use the waterproof version of the DS18B20 temperature sensor. The sensor is shown in the figure below.



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:



(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 follow the next 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)

Before uploading the code, you need to install two libraries in your Arduino IDE. The [OneWire library by Paul Stoffregen](#) and the [Dallas Temperature library](#), so that you can use the DS18B20 sensor. Follow the next steps to install those libraries.

OneWire library

1. [Click here to download the OneWire library](#). You should have a .zip folder in your Downloads
2. Unzip the .zip folder and you should get OneWire-master folder
3. Rename your folder from ~~OneWire-master~~ to OneWire
4. Move the OneWire folder to your Arduino IDE installation libraries folder
5. Finally, re-open your Arduino IDE

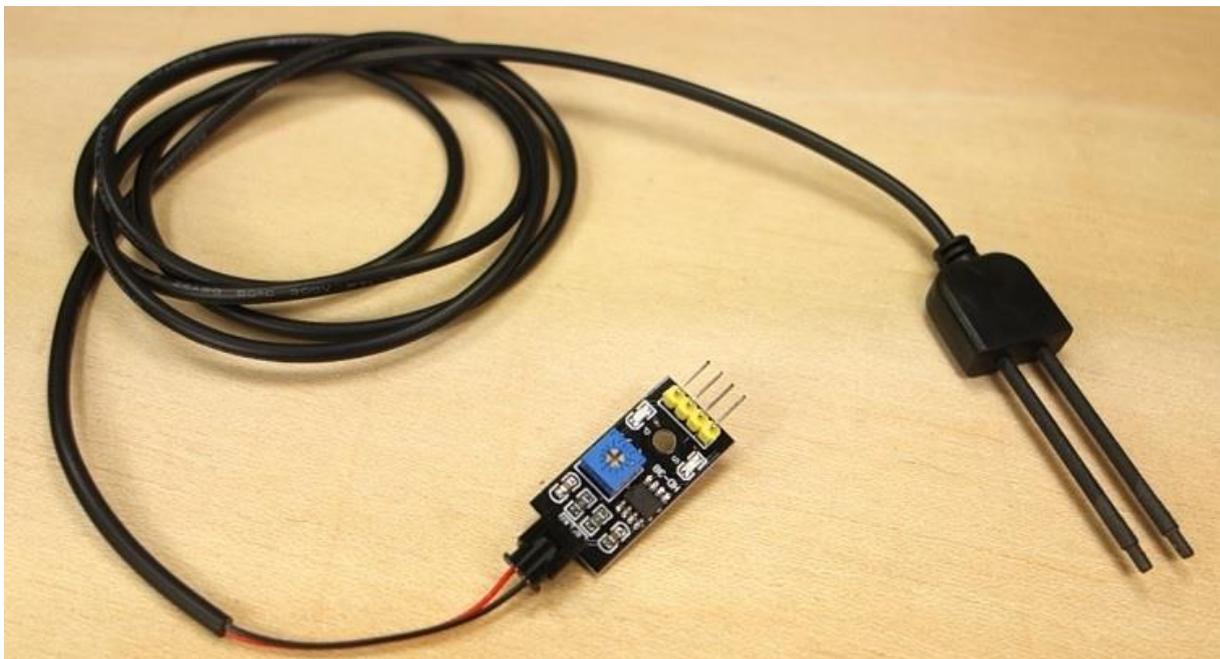
Dallas Temperature library

[Click here to download the DallasTemperature library](#). You should have a .zip folder in your Downloads

1. Unzip the .zip folder and you should get Arduino-Temperature-Control-Library-master folder
2. Rename your folder from ~~Arduino-Temperature-Control-Library-master~~ to DallasTemperature
3. Move the DallasTemperature folder to your Arduino IDE installation libraries folder
4. Finally, re-open your Arduino IDE

Preparing the Soil Moisture Sensor

To read soil moisture, we'll use a resistive soil moisture sensor as shown in the figure below.



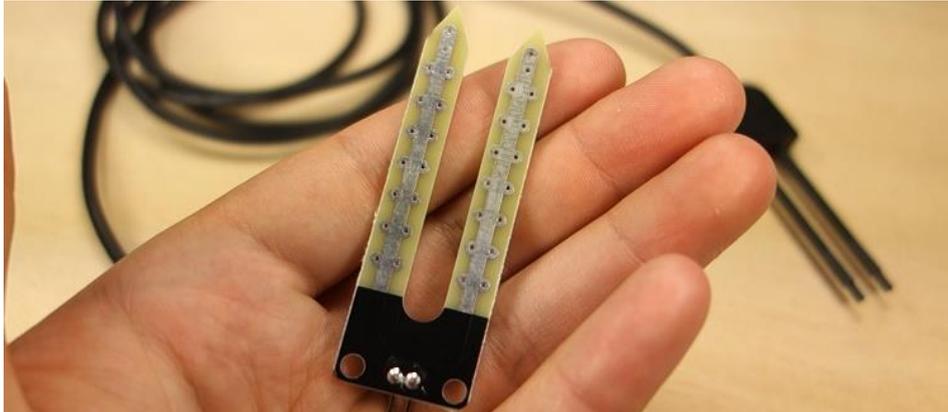
The sensor's output voltage changes accordingly to the soil moisture. When the soil is:

- **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

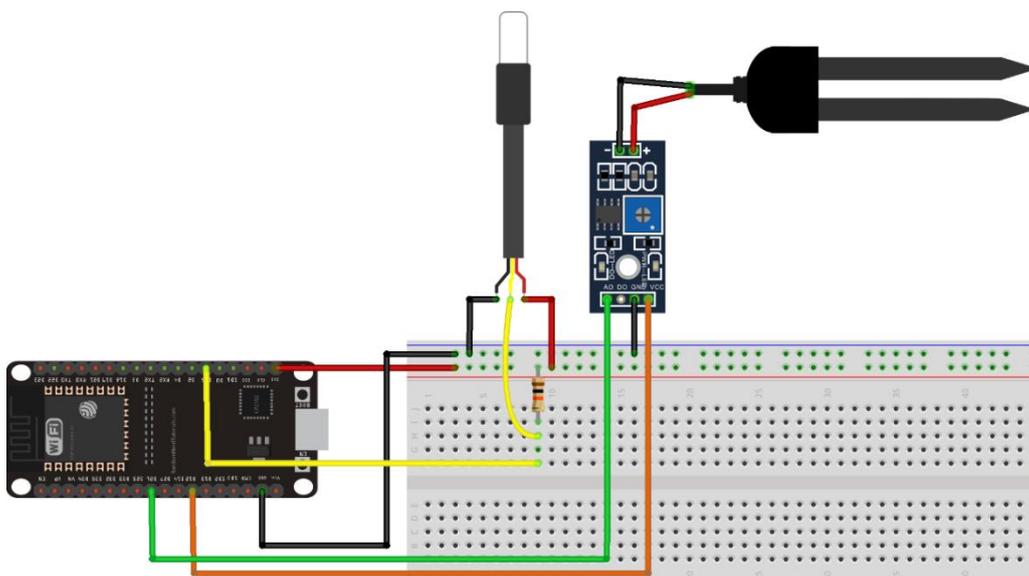
Cheap soil moisture sensors can oxidize very quickly and get damaged.



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.



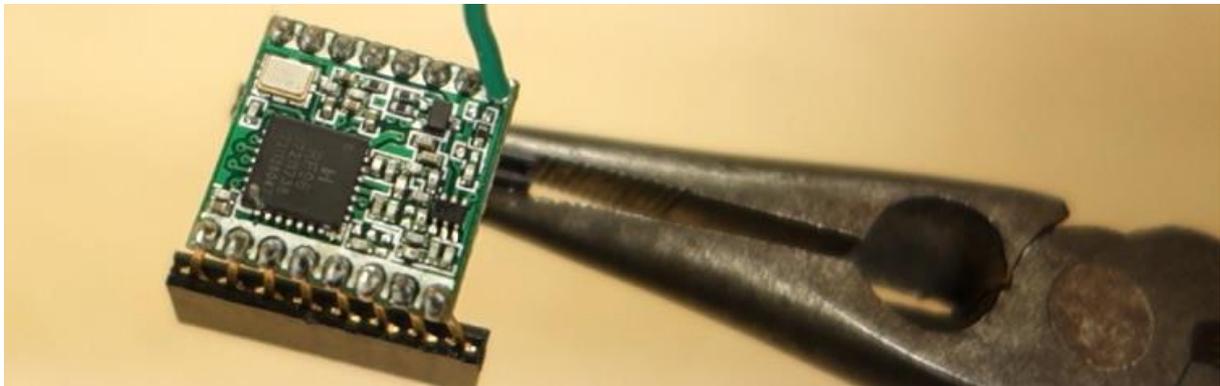
(This schematic uses the ESP32 DEVKIT V1 module version with 36 GPIOs – if you're using another model, please check the pinout for the board you're using.)

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

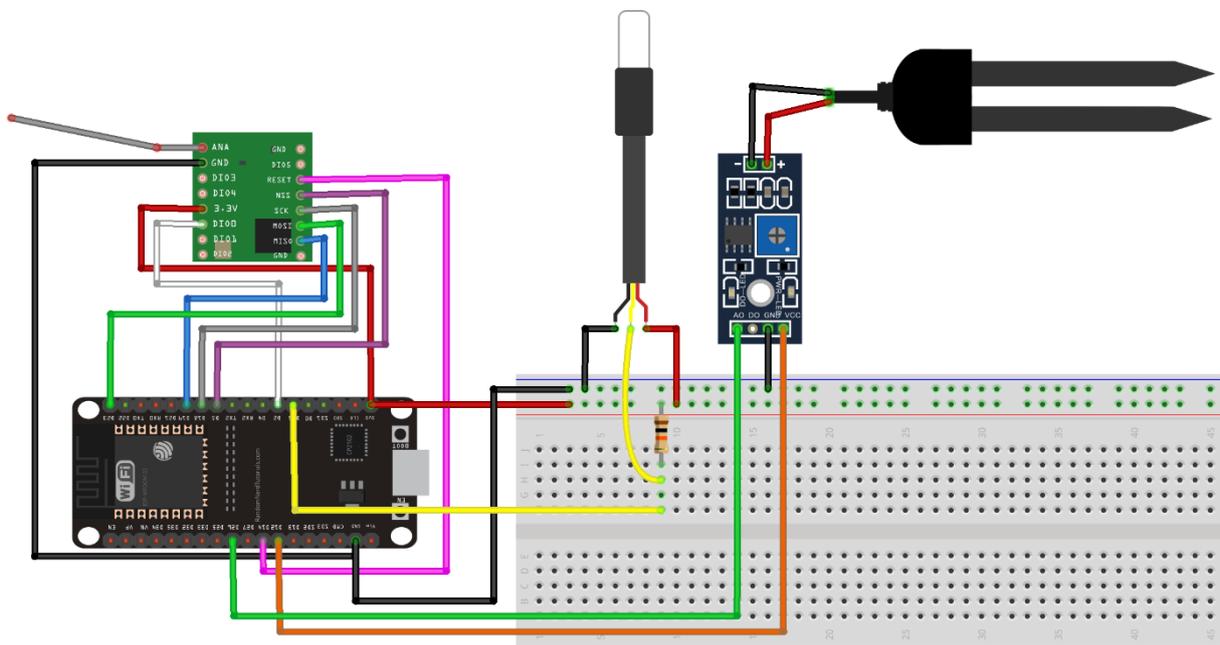
Preparing the LoRa Transceiver Module

The sensor readings are sent to the indoor receiver using LoRa. In this project we're going to use the LoRa transceiver module we've used in the **"LoRa Module ESP32 - LoRa Sender and Receiver"** Unit. So, we recommend reading the LoRa Module first, to prepare your LoRa transceiver.



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.



(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.)

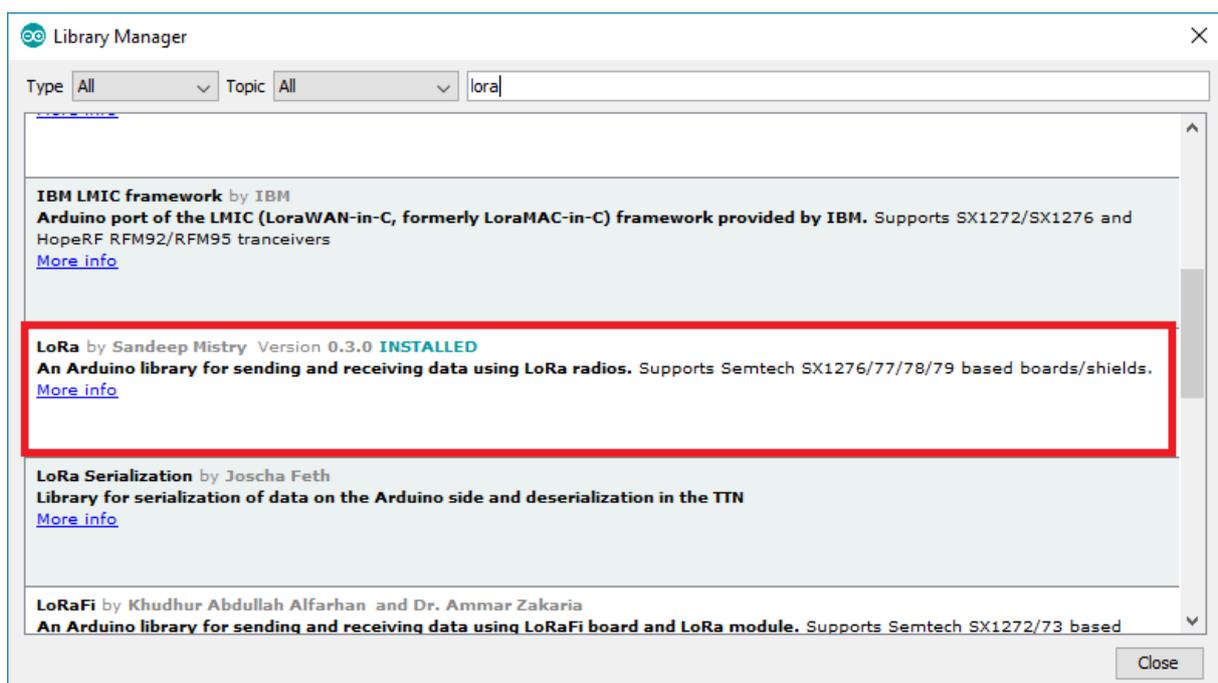
You can also follow the next wiring table.

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	-

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". Select the LoRa library highlighted in the figure below, and install it.



The circuit is ready for this Unit. We need to make sure that both the LoRa sender and receiver are working properly before making it 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

The LoRa Sender Code

Copy the following code to your Arduino IDE. Don't upload it yet. First, we'll take a quick look on how it works:

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/LoRa_Project/LoRa_Sender/LoRa_Sender.ino

```
// 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(moisturePin);
  digitalWrite(moisturePowerPin, 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 the Code Works

To better understand the code, we recommend taking a look at the following Units:

- **Deep Sleep -Timer Wake Up:** Module 3, Unit 2
- **ESP32 Web Server – Display Sensor Readings:** Module 4, Unit 8
- **ESP32 – LoRa Sender and Receiver:** Module 6, Unit 2

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.

```
    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() function

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.

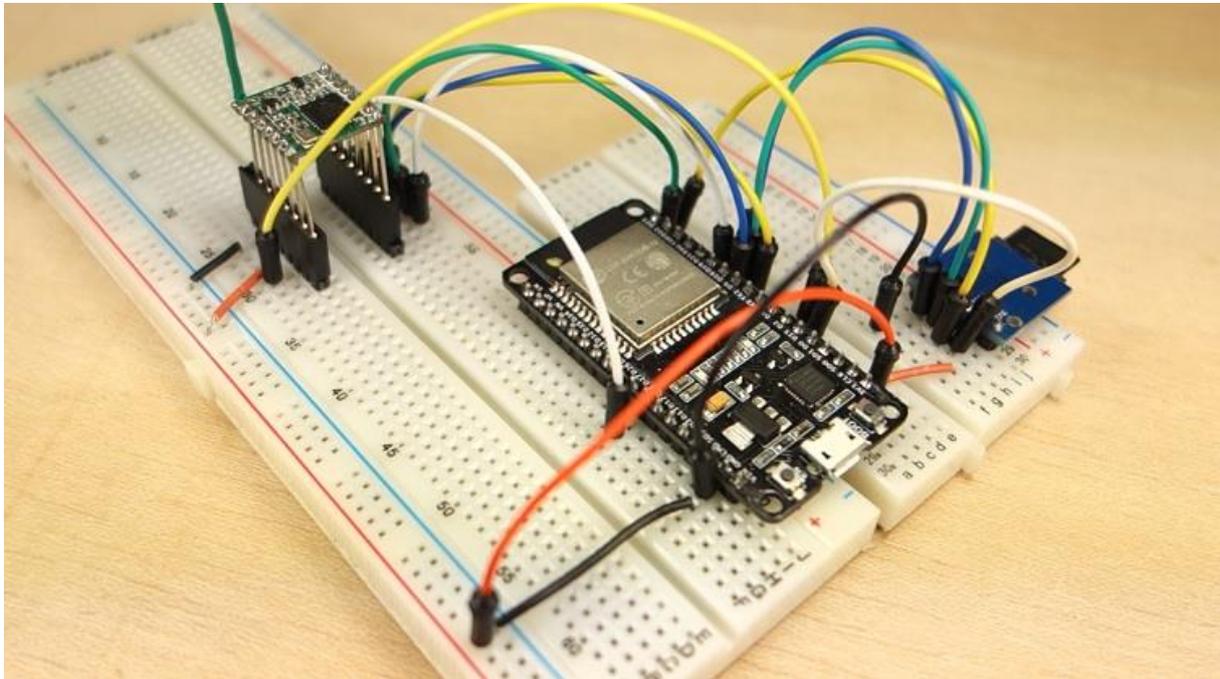
```
rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:812
load:0x40078000,len:0
load:0x40078000,len:11392
entry 0x40078a9c
initializing 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.
```

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.

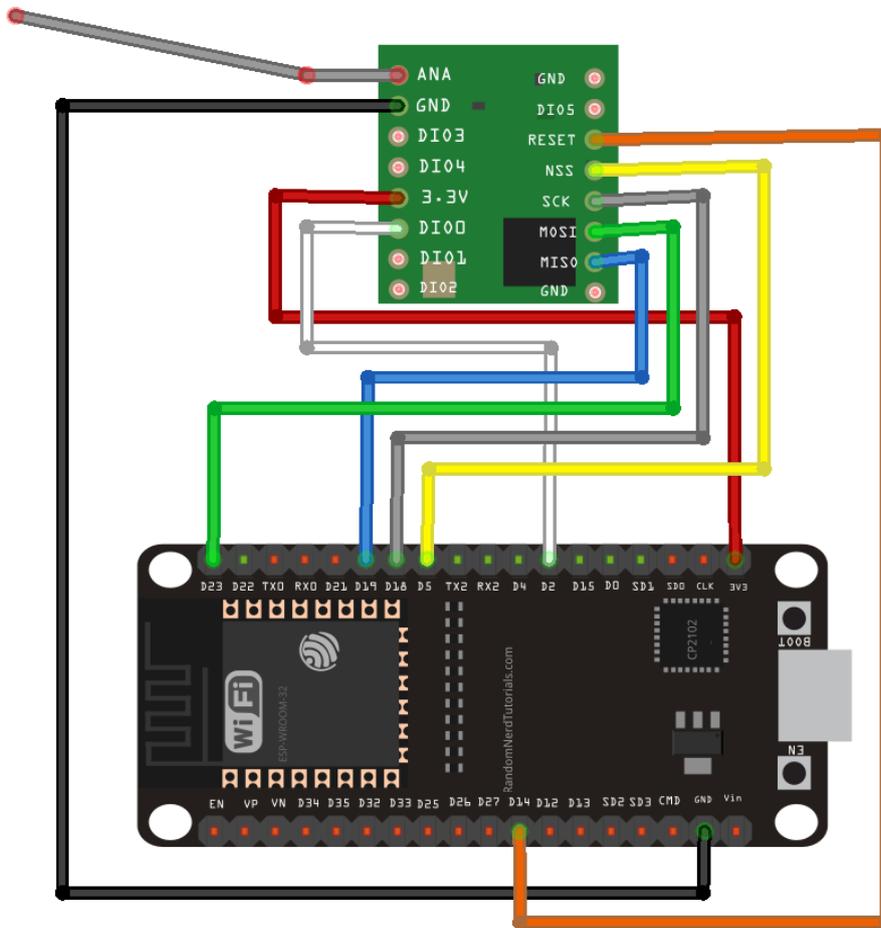
Parts Required:

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.



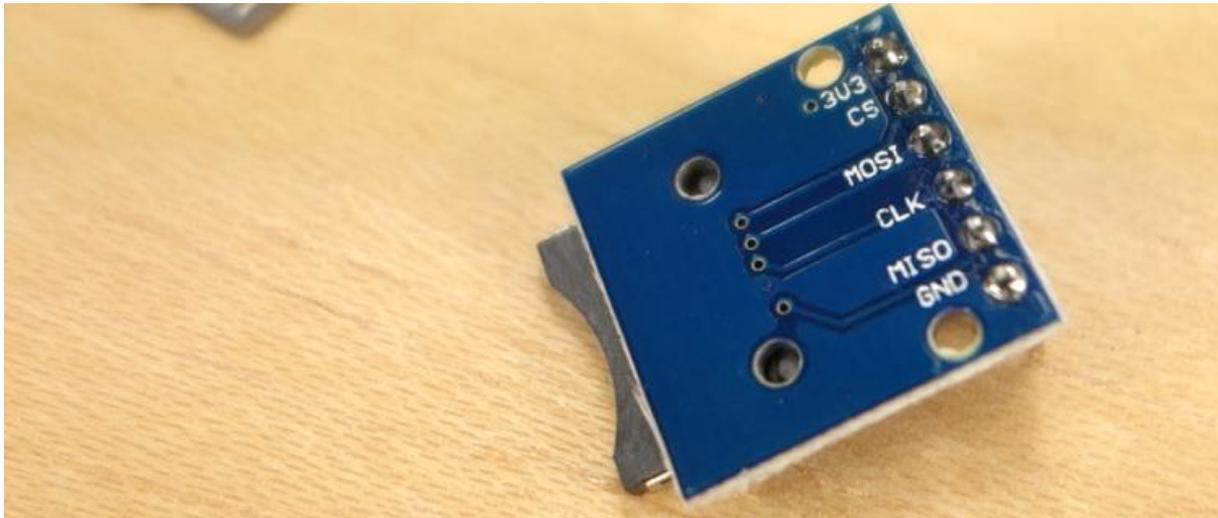
(This schematic uses the ESP32 DEVKIT V1 module version with 36 GPIOs – if you're using another model, please check the pinout for the board you're using.)

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.

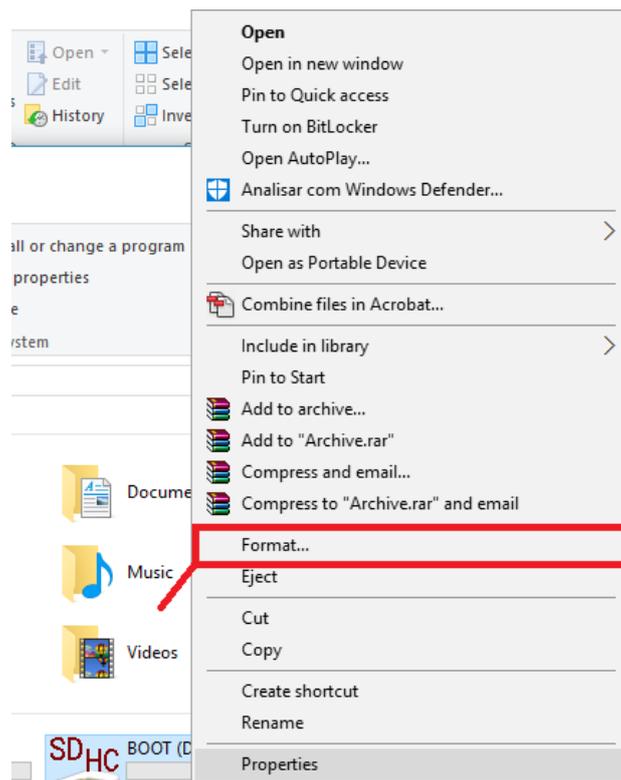


There are different models from different suppliers, but they all work in a similar way. They use SPI communication protocol. To communicate with the microSD card we're going to use the SD.h library that comes installed in the Arduino IDE by default.

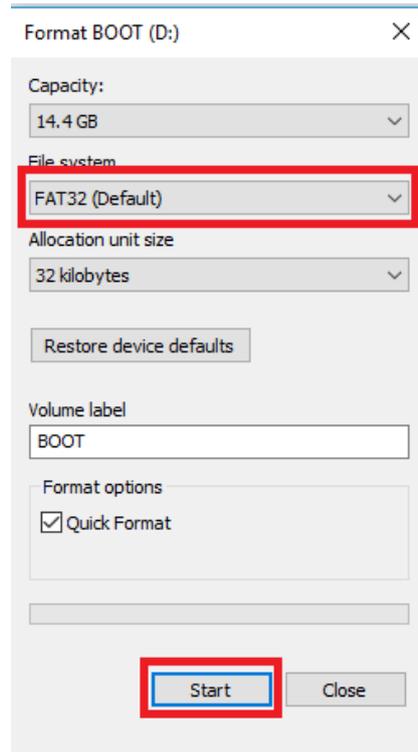
Formatting the microSD card

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.

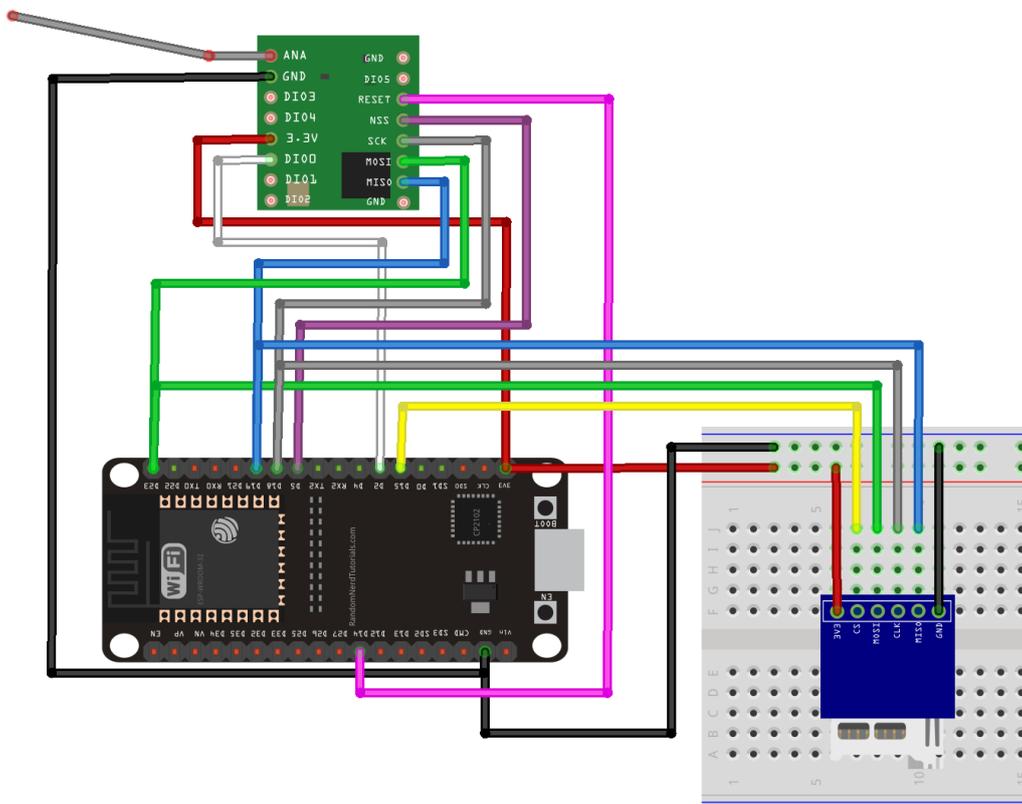


2. A new window pops up. Select FAT32, press Start to initialize the formatting process and follow the onscreen instructions.



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.

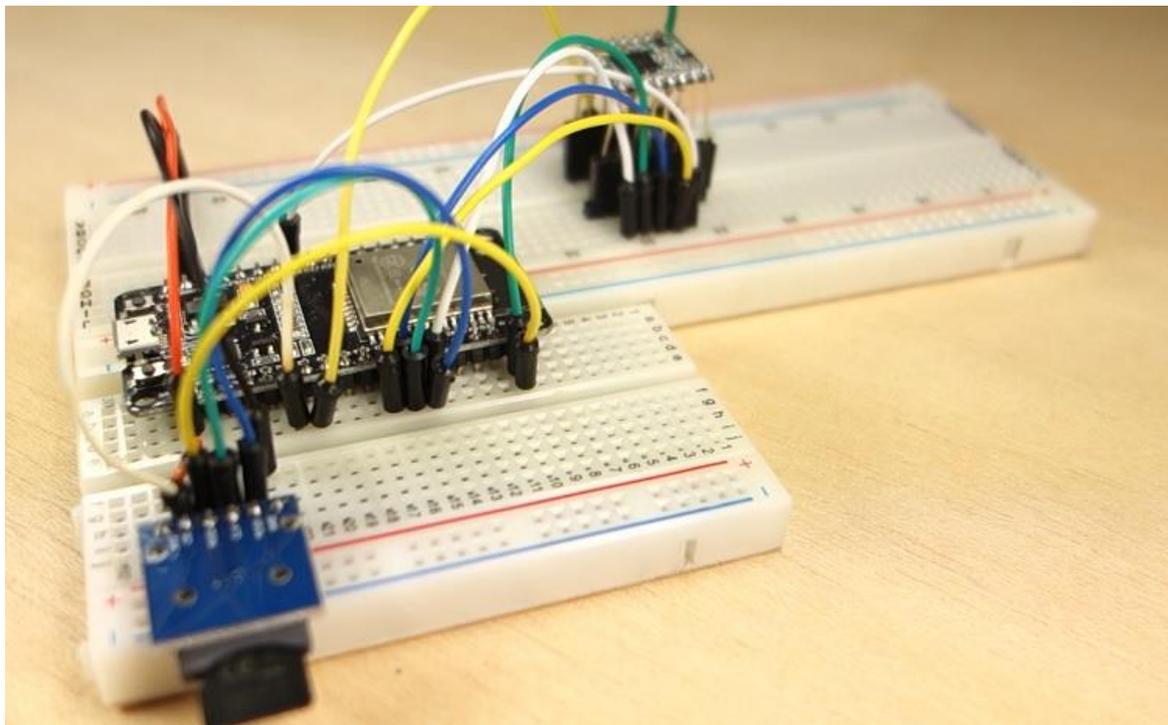


(This schematic uses the ESP32 DEVKIT V1 module version with 36 GPIOs – if you're using another model, please check the pinout for the board you're using.)

You can also use the following table as a reference.

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

Everytime the LoRa receiver picks up a new a LoRa message, it will request the date and time to from an NTP server to add timestamps to the data logger.

For that we'll be using the [NTPClient library forked by Taranais](#). 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
2. Unzip the .zip folder and you should get NTPClient-master folder

3. Rename your folder from ~~NTPClient-master~~ to NTPClient
4. Move the NTPClient folder to your Arduino IDE installation libraries folder
5. Finally, re-open your Arduino IDE

The LoRa Receiver Code

Copy the following code to your Arduino IDE and let's see how it works.

SOURCE CODE

[https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/LoRa Project/LoRa Receiver/LoRa Receiver.ino](https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/LoRa%20Project/LoRa%20Receiver/LoRa%20Receiver.ino)

```
// 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();
    if (client) {
        Serial.println("New Client.");
        String currentLine = "";
        while (client.connected()) {
            if (client.available()) {
                char c = client.read();
                Serial.write(c);
                header += c;
                if (c == '\n') {
                    if (currentLine.length() == 0) {
                        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>");
        client.println();
        break;
    } else {
        currentLine = "";
    }
} else if (c != '\r') {
    currentLine += c;
}
}
}
header = "";
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
    rssi = LoRa.packetRssi();
    Serial.print(" with RSSI ");
    Serial.println(rssi);
}

// 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();
}

```

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 is able to 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 rssi;  
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.

```
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.

```
rss = 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()

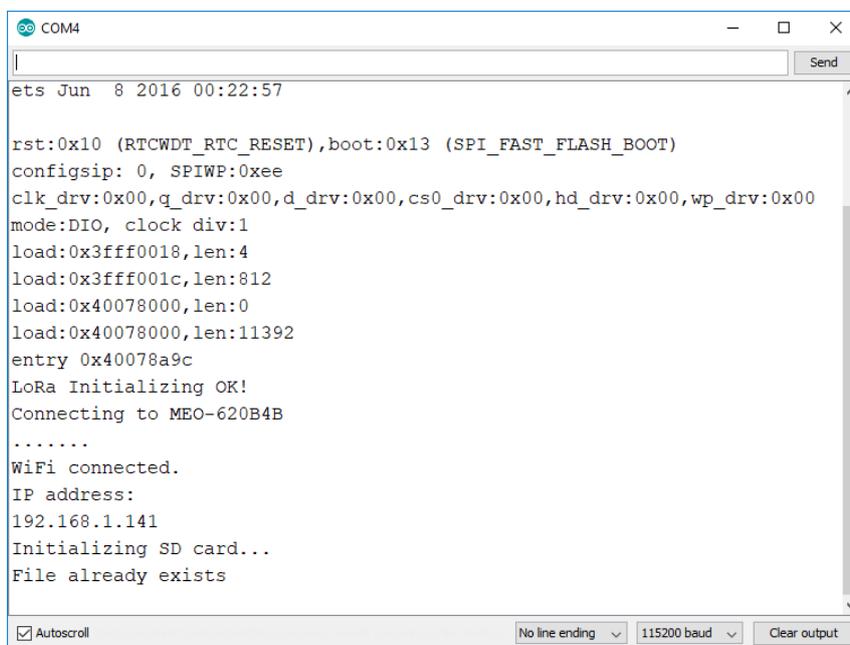
The last two functions: `writeFile()` and `appendFile()` are used to write and append data to the microSD card. They come with the SD card library examples and you shouldn't modify them.

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.

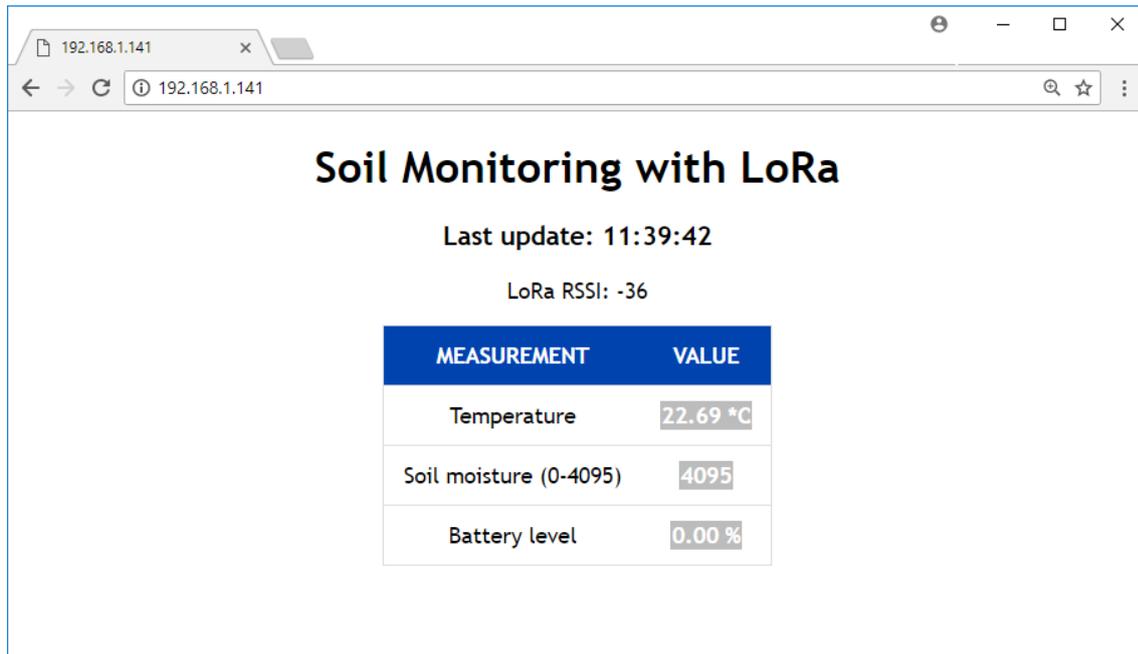


```
ets Jun 8 2016 00:22:57

rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
confsip: 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
LoRa Initializing OK!
Connecting to MEO-620B4B
.....
WiFi connected.
IP address:
192.168.1.141
Initializing SD card...
File already exists
```

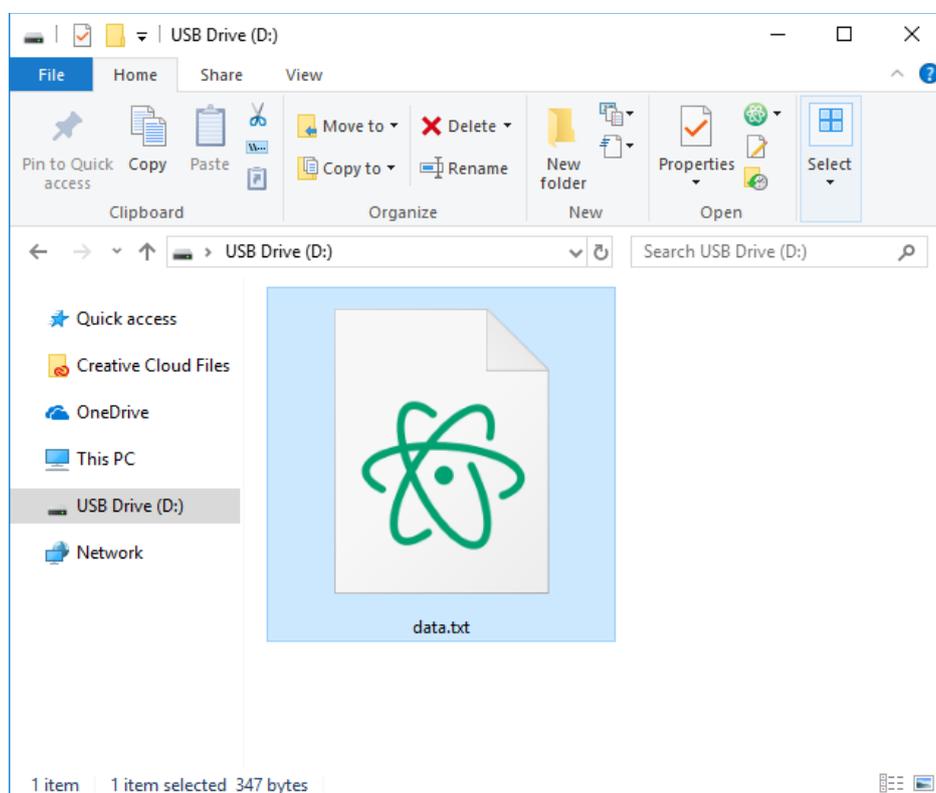
Also, check if the LoRa transceiver and the microSD card module were successfully initialized. If everything is working properly, let's see if it can receive the LoRa sender messages.

You can view the latest readings by accessing the web server. Type ESP32 IP address in your browser, and check the readings.

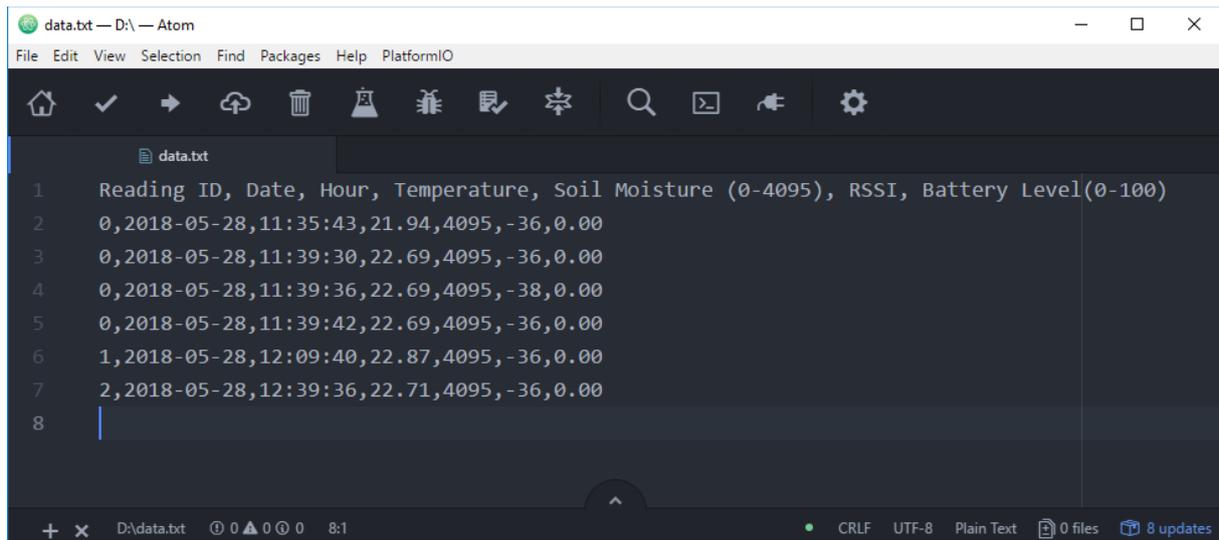


Close your browser and let the sender and receiver circuits on for a few hours to test if everything is working properly.

After the testing period, insert the microSD card in your computer and open the *data.txt* file.



You should have new readings on the file.



The screenshot shows a text editor window titled 'data.txt' with the following content:

```
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
```

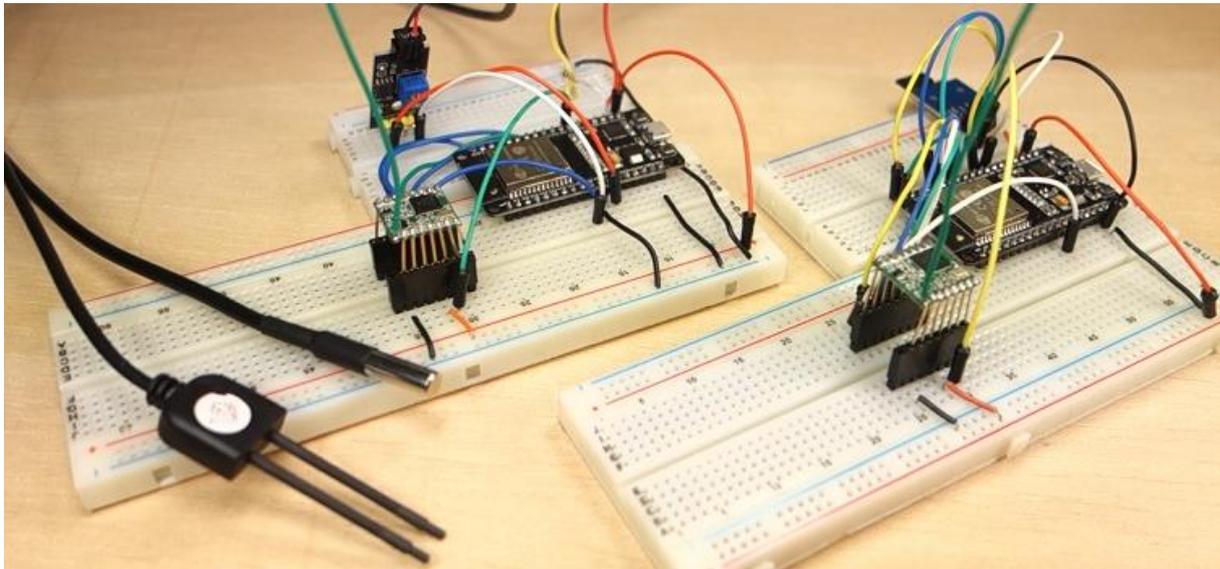
The status bar at the bottom indicates the file is in CRLF, UTF-8, Plain Text format, with 0 files and 8 updates.

Continue To The Next Unit ...

If everything is working, go to the next Unit to make your LoRa Sender solar powered.

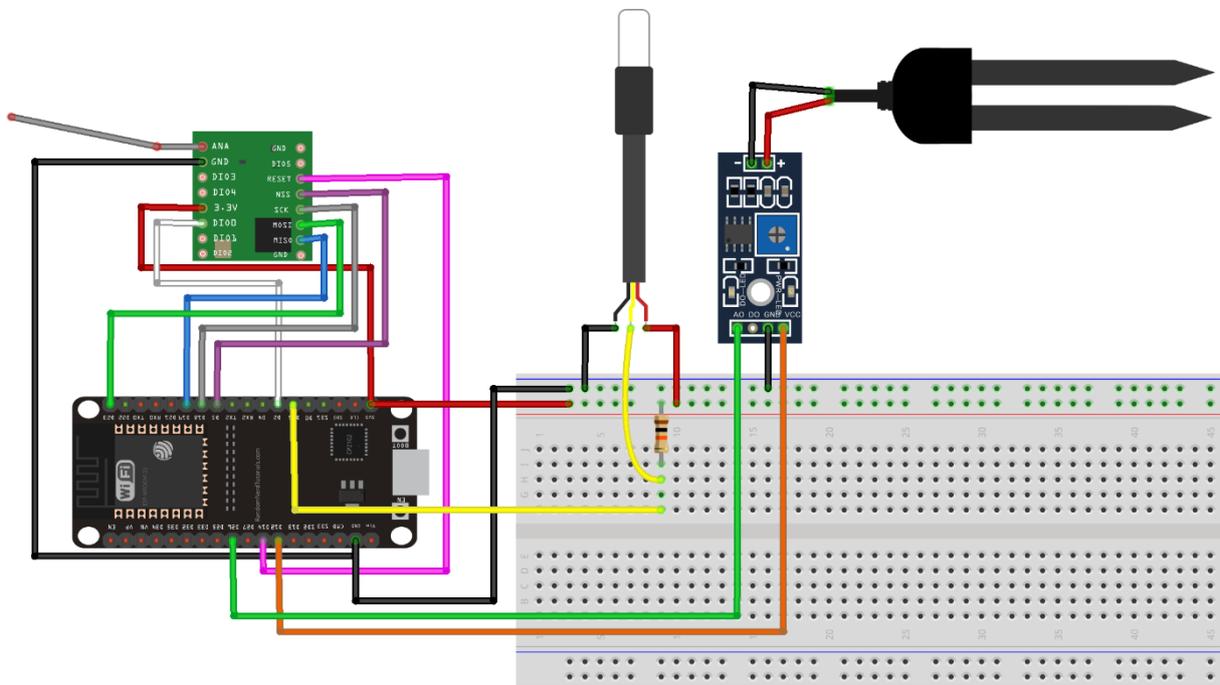
Unit 4 - LoRa Sender Solar Powered

In Part 2 you've built the LoRa sender circuit and in Part 3 the LoRa receiver.



In this Unit you're going to 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).

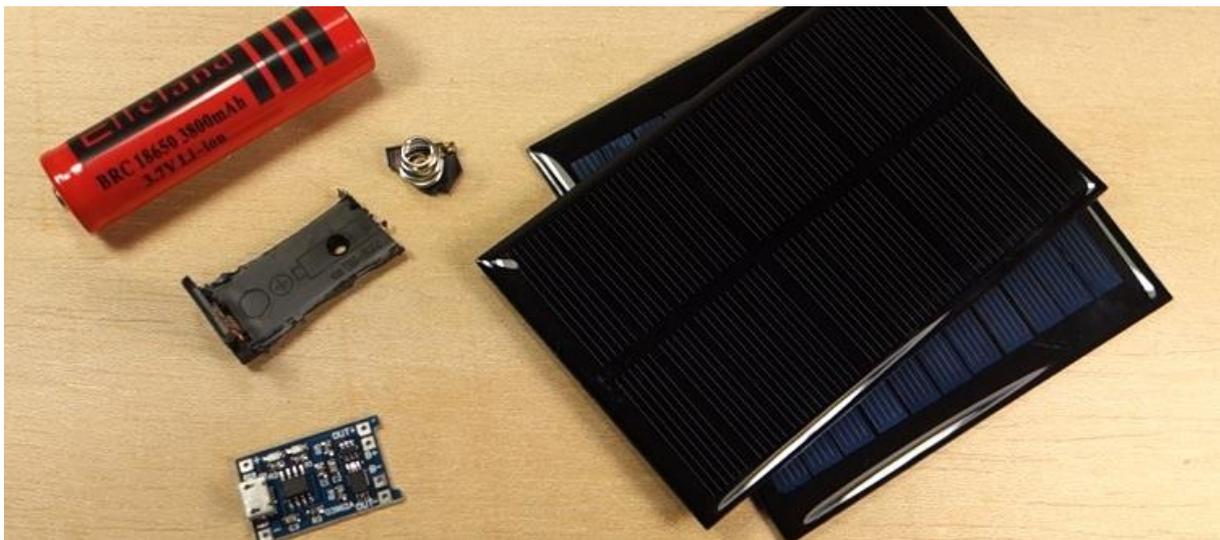
Parts Required:

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, 2 mini solar panels (5V and at least 1.2W), and the TP4056 lithium battery charger module.

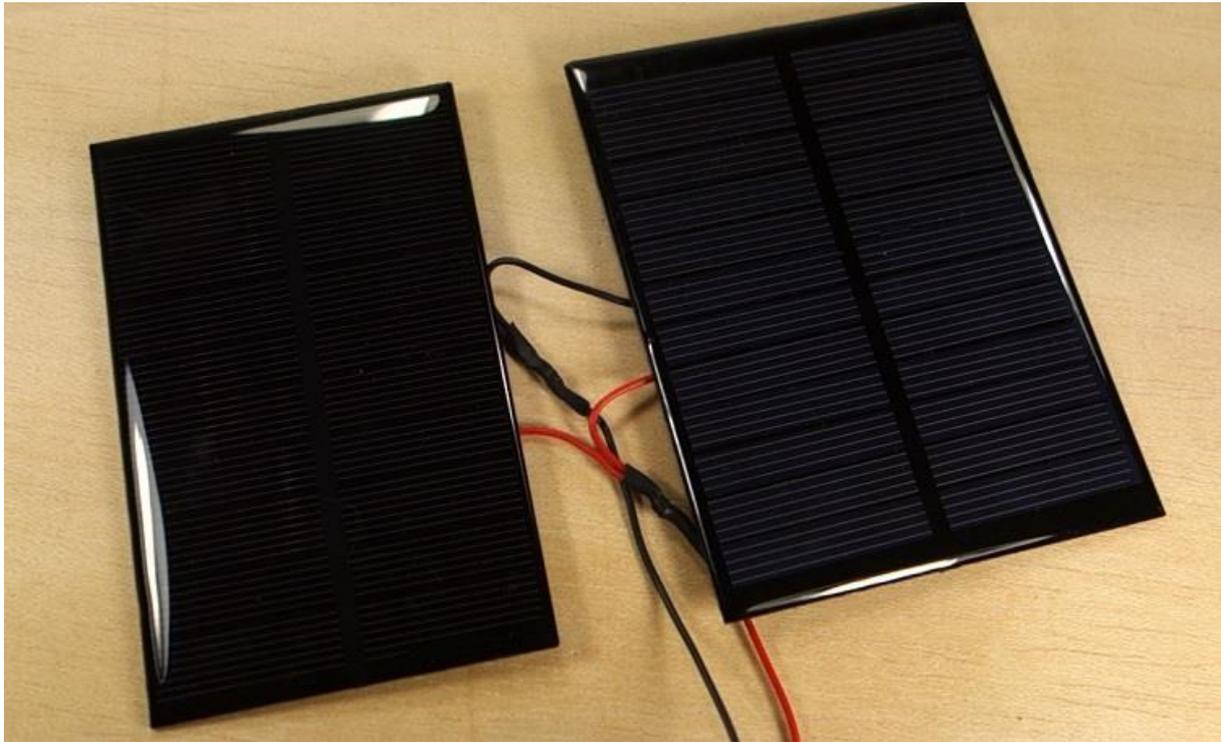


The TP4056 lithium battery charger module comes with circuit protection. It prevents battery over-voltage and reverse polarity connection.

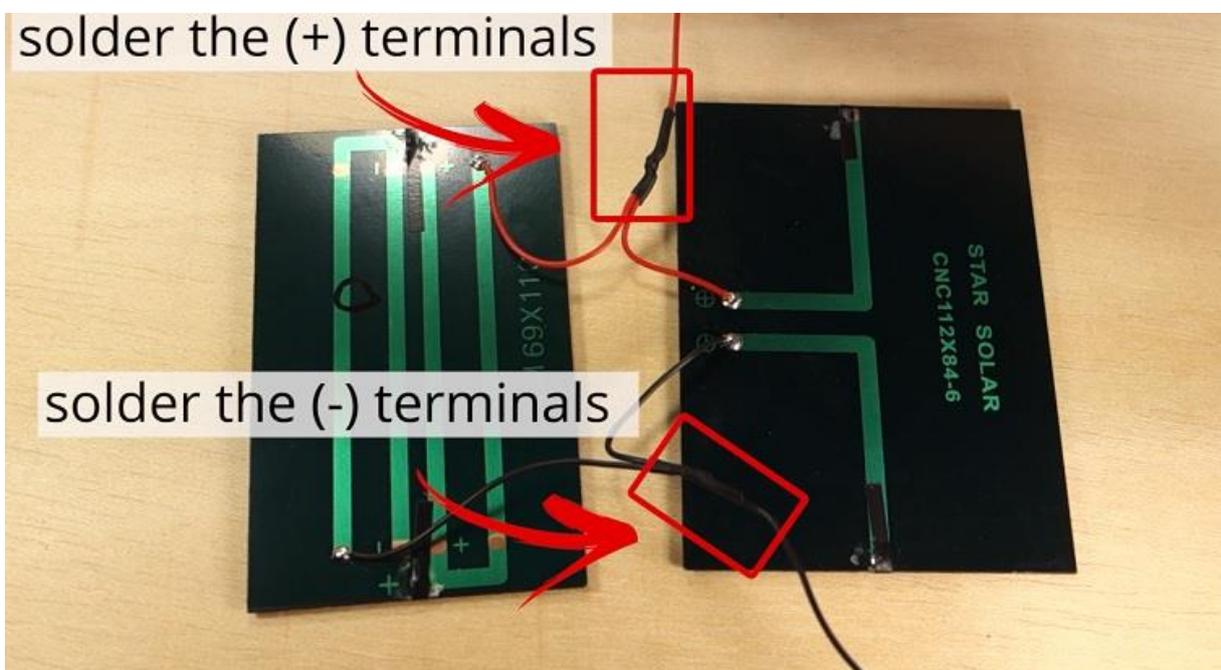


Solar panels in parallel

To charge the battery faster we're going to use two solar panels in parallel (instead of one).



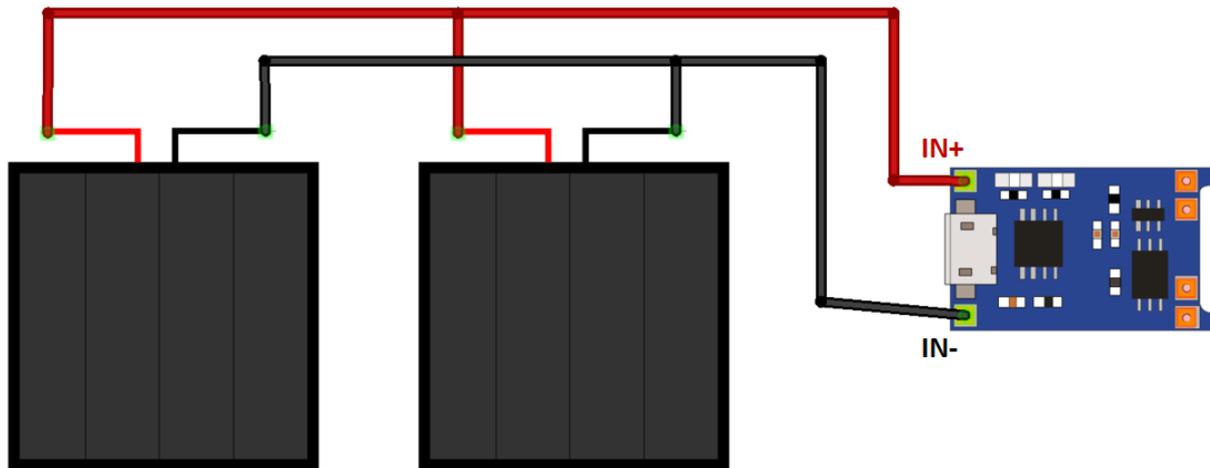
To wire solar panels in parallel solder the plus terminals with each other, as well as the minus terminals.



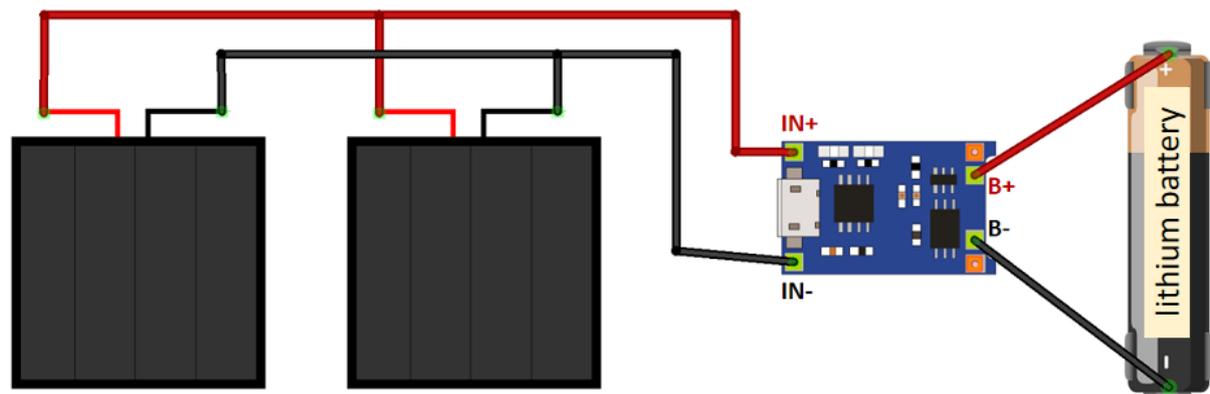
When wiring solar panels in parallel you'll get the same output voltage, and double the current (for identical solar panels).

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.



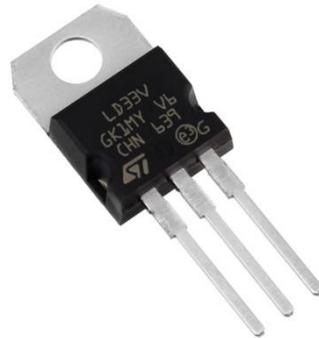
The OUT+ and OUT- will be powering the ESP32. However, lithium batteries output up to 4.2V when fully charged, but the ESP32 recommended operating voltage is 3.3V.



Important: you can't plug the Lithium battery directly to an ESP32, you need a voltage regulator circuit.

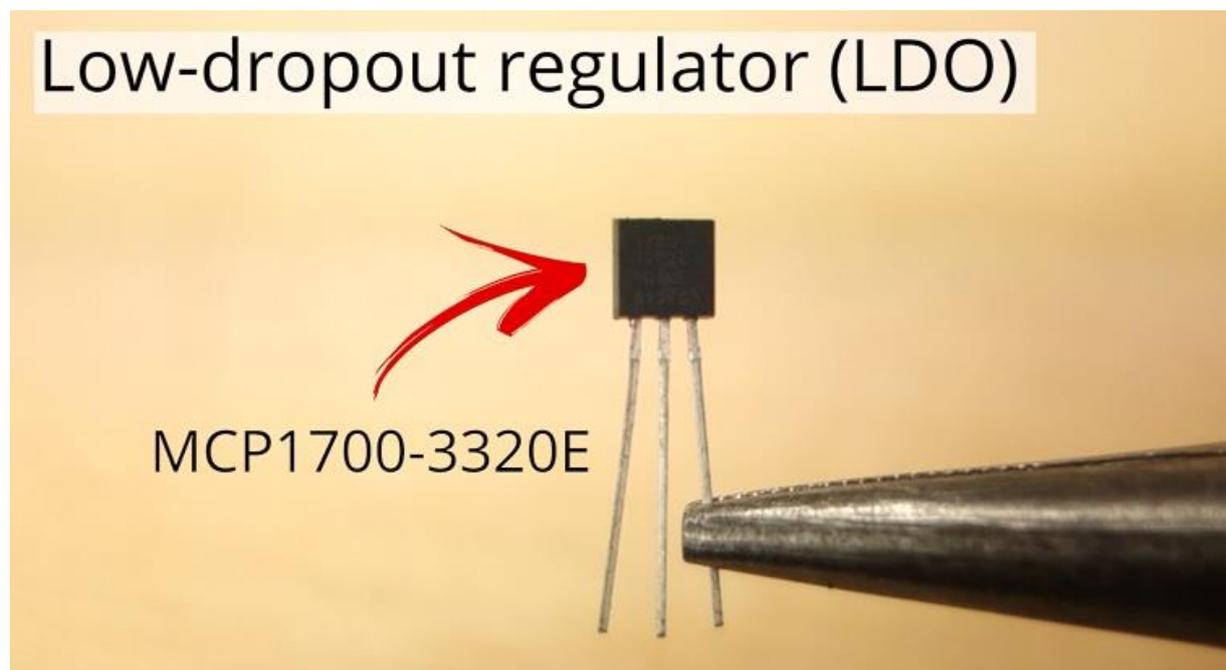
Voltage Regulator

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.



MCP1700

Low Quiescent Current LDO

Features:

- 1.6 μA 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 $\geq 2.5\text{V}$
- 200 mA Output Current for Output Voltages $< 2.5\text{V}$
- Low Dropout (LDO) Voltage
 - 178 mV Typical @ 250 mA for $V_{\text{OUT}} = 2.8\text{V}$
- 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 μF Ceramic Output Capacitor
- Short Circuit Protection
- Overtemperature Protection

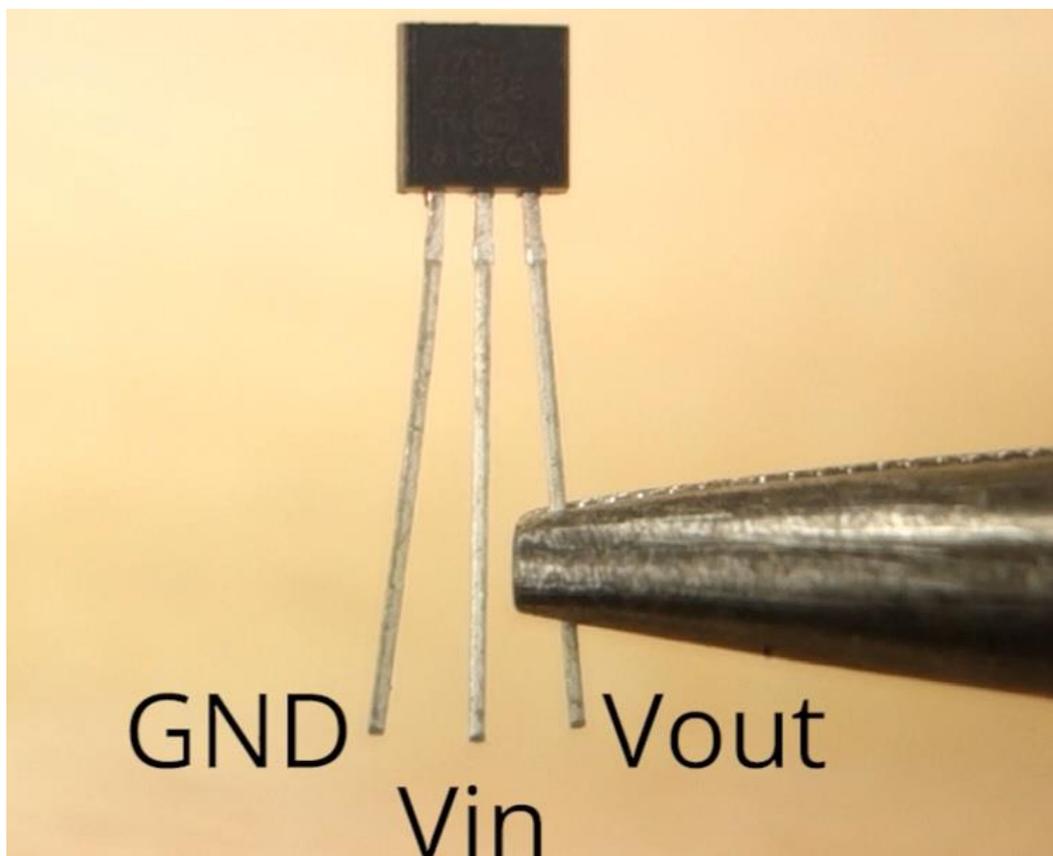
General Description:

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 μA 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 ($V_{\text{OUT}} = 2.8\text{V}$). The output voltage tolerance of the MCP1700 is typically $\pm 0.4\%$ at $+25^\circ\text{C}$ and $\pm 3\%$ maximum over the operating junction temperature range of -40°C to $+125^\circ\text{C}$.

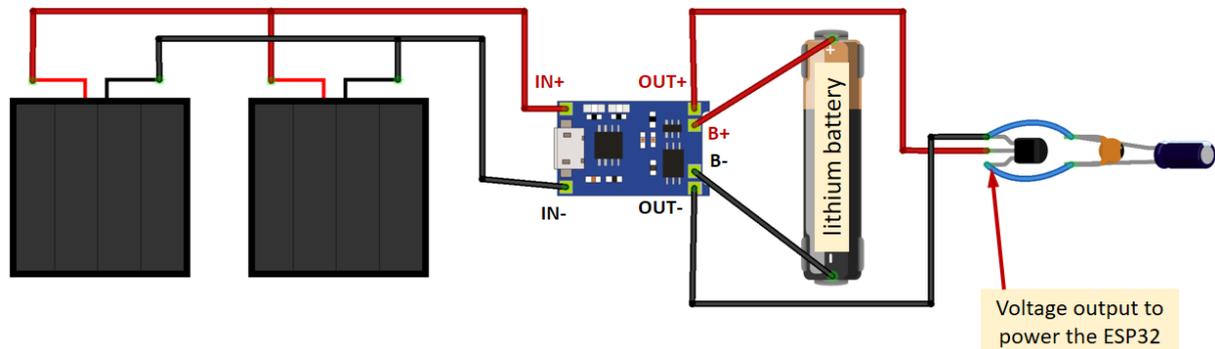
Output voltages available for the MCP1700 range from 1.2V to 5.0V. The LDO output is stable when using only 1 μF output capacitance. Ceramic, tantalum or aluminum electrolytic capacitors can all be used for

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.

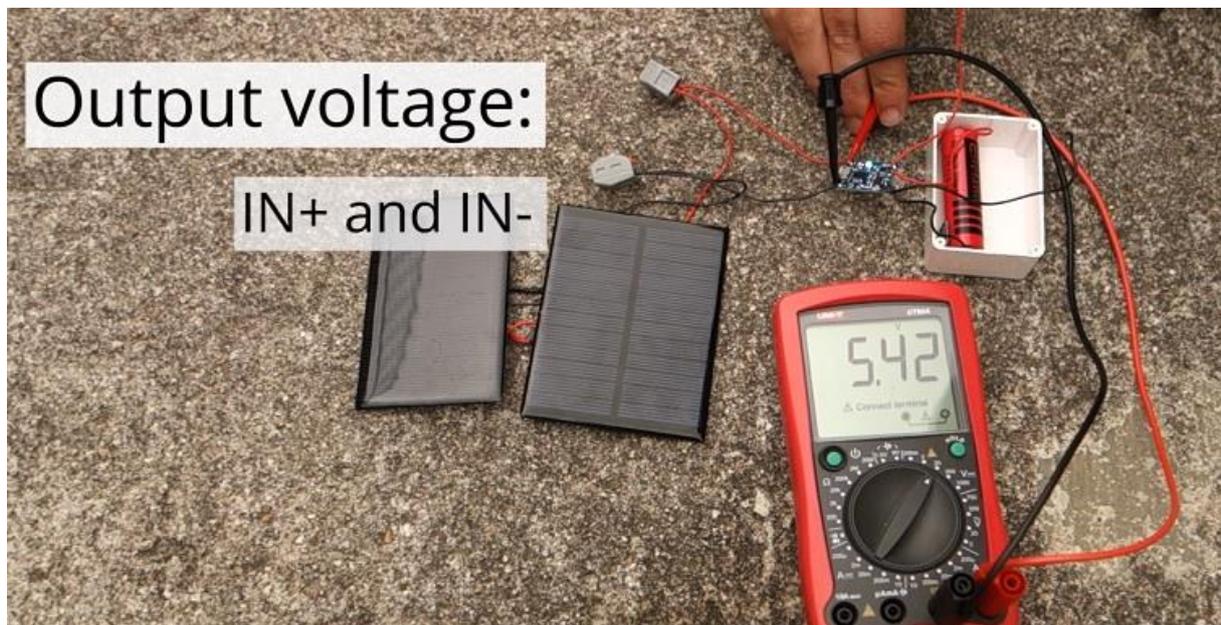


Warning: electrolytic capacitors have polarity! The lead with the white band should be connected to GND.

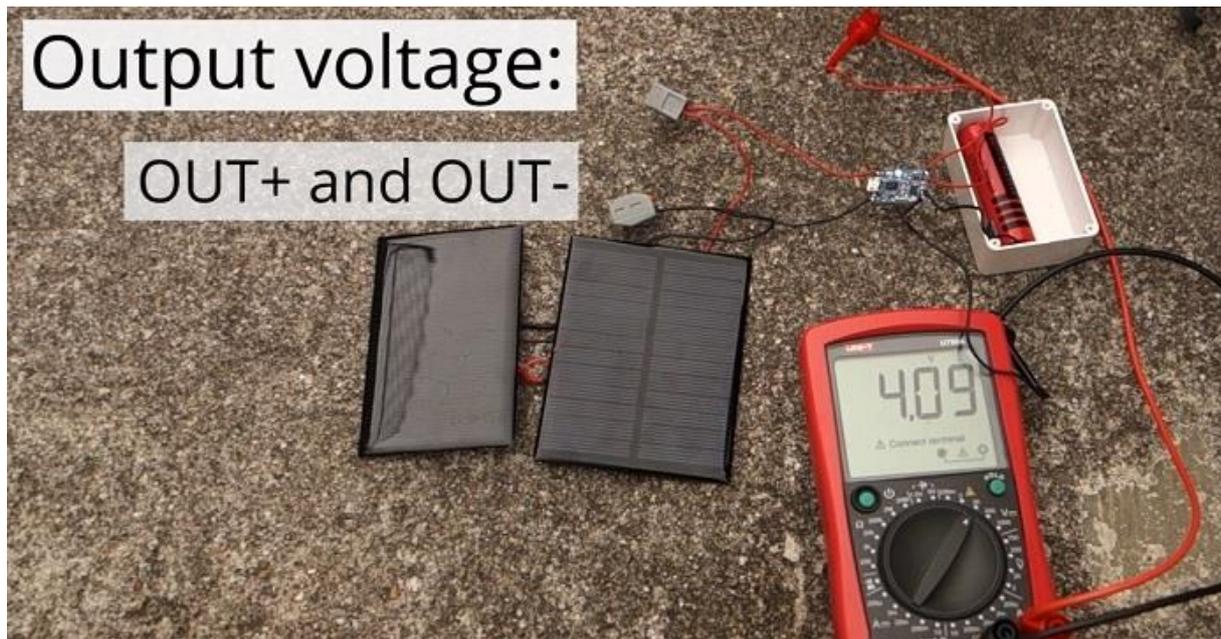
The Vout pin of the voltage regulator should output 3.3V. That is the pin that will be powering the ESP32.

Measuring Output Voltages

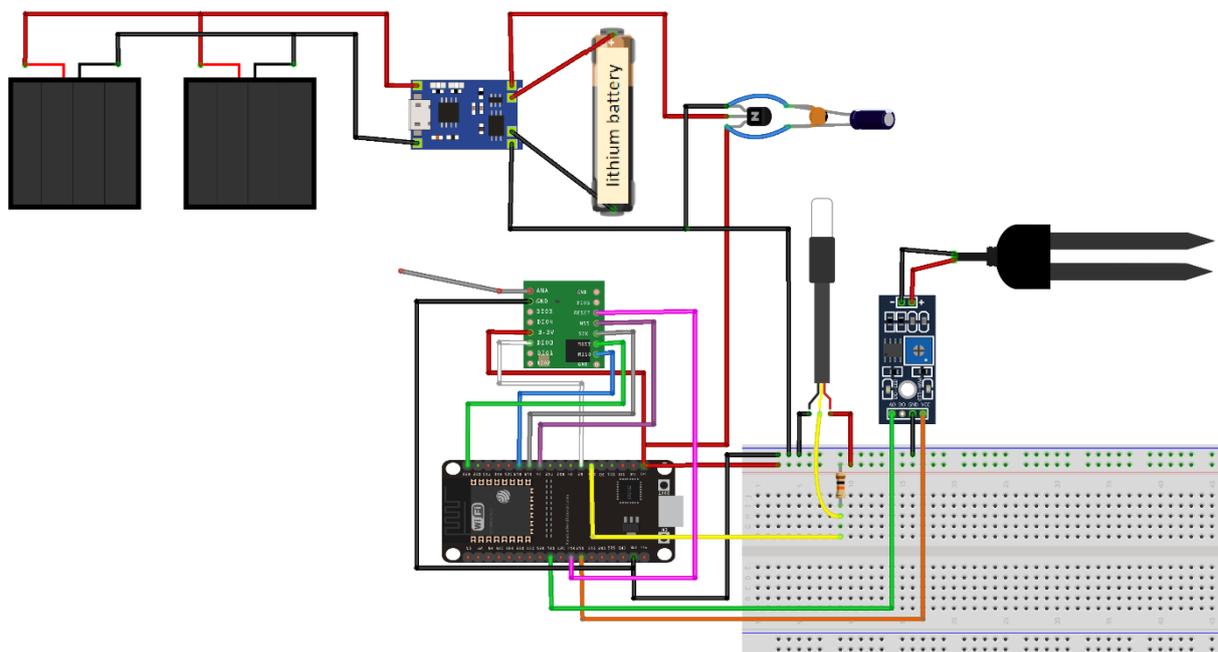
Now, let's test if everything is working properly. Start by measuring the output voltage of the IN+ and IN- pads. It should output approximately 5V, which is the output voltage of the solar panels.



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.

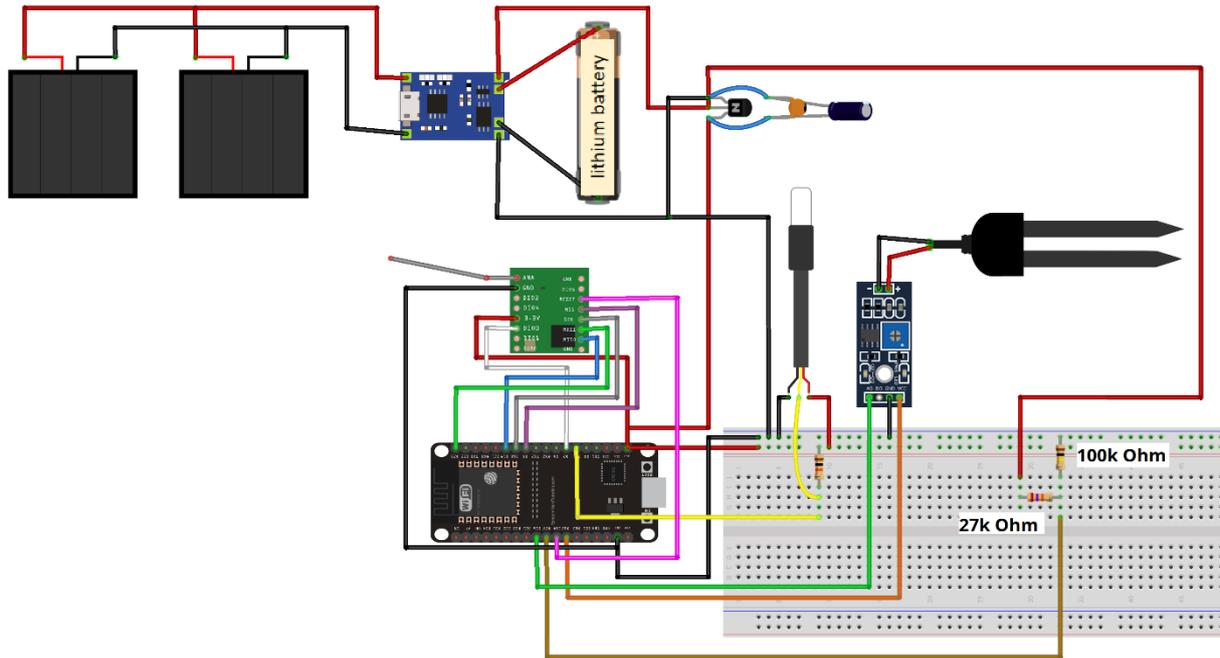


Battery Voltage Level Monitoring Circuit

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. As explained in an earlier video, the LoRa sender code is already

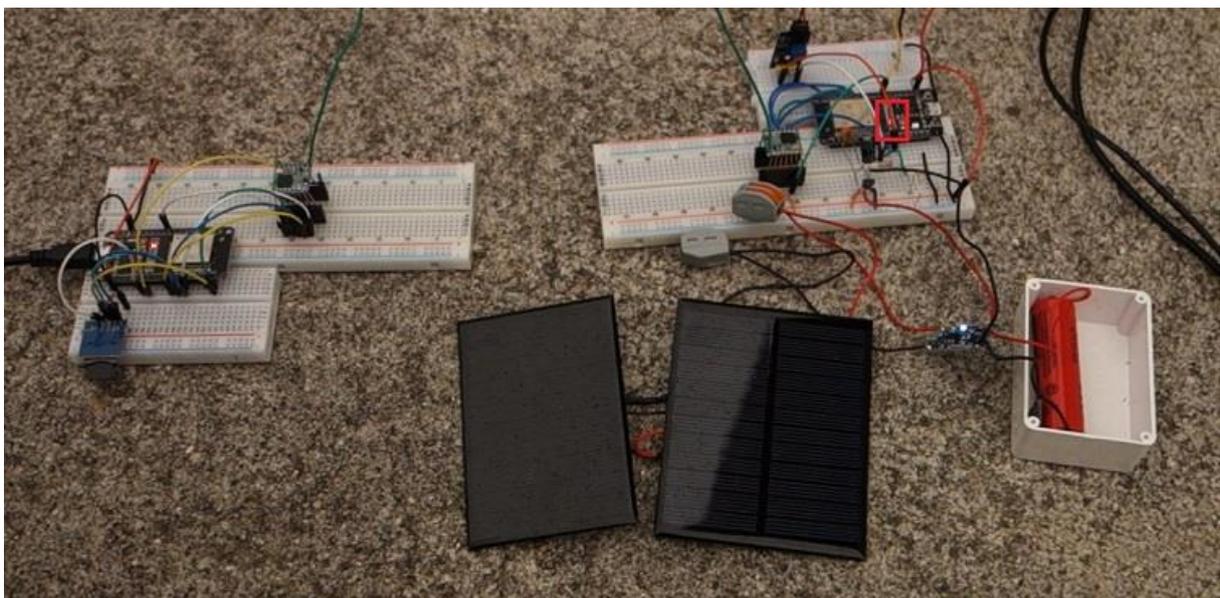
prepared to read the battery level on GPIO 27. So, you just need to add the battery voltage level monitoring circuit as shown in the following schematic diagram.



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.

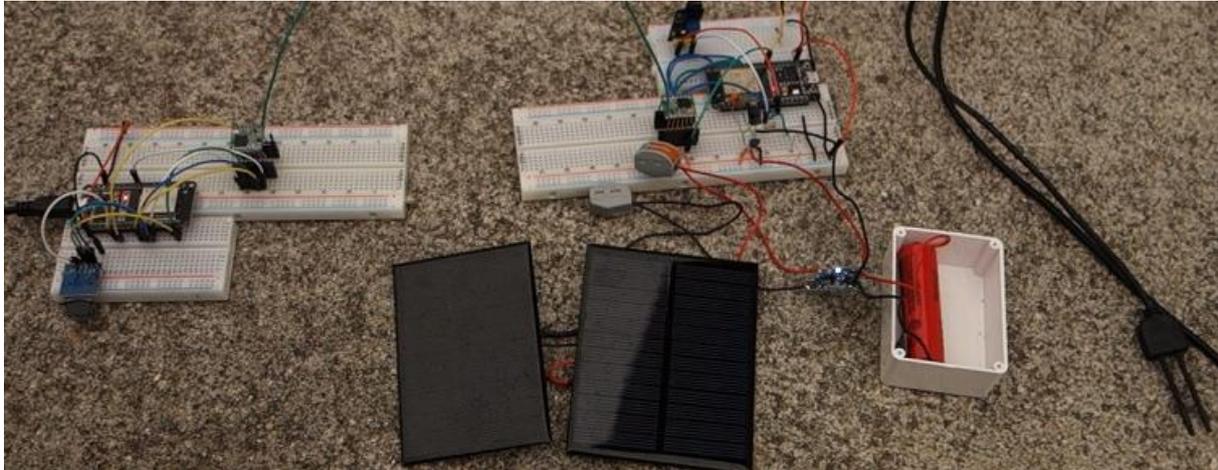
Congratulations, your project is finally completed!

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

For this project we'll leave both circuits in a breadboard, but you can use some stripboards to make a permanent circuit.

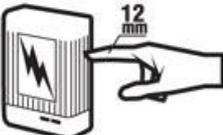
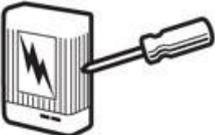
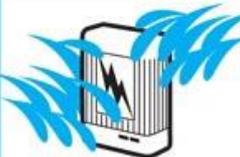
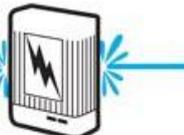
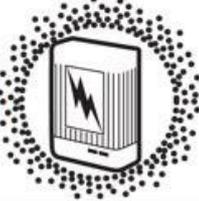
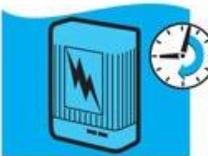
We'll be putting the LoRa sender circuit in an IP65 enclosure, because the circuit will be placed outside.



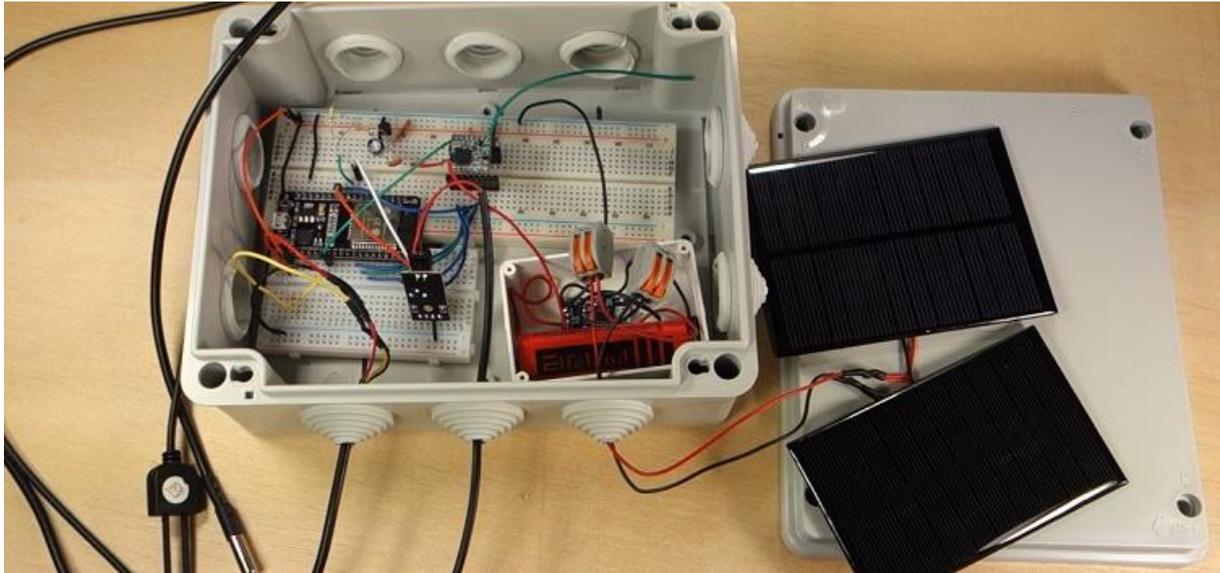
This kind of enclosure is rated as a “dust tight” and protected against water projected from a nozzle.

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		WATER	
1	 <p>Protected against a solid object greater than 50 mm such as a hand.</p>	1	 <p>Protected against vertically falling drops of water. Limited ingress permitted.</p>
2	 <p>Protected against a solid object greater than 12.5 mm such as a finger.</p>	2	 <p>Protected against vertically falling drops of water with enclosure tilted up to 15 degrees from the vertical. Limited ingress permitted.</p>
3	 <p>Protected against a solid object greater than 2.5 mm such as a screwdriver.</p>	3	 <p>Protected against sprays of water up to 60 degrees from the vertical. Limited ingress permitted for three minutes.</p>
4	 <p>Protected against a solid object greater than 1 mm such as a wire.</p>	4	 <p>Protected against water splashed from all directions. Limited ingress permitted.</p>
5	 <p>Dust Protected. Limited ingress of dust permitted. Will not interfere with operation of the equipment. Two to eight hours.</p>	5	 <p>Protected against jets of water. Limited ingress permitted.</p>
6	 <p>Dust tight. No Ingress of dust. Two to eight hours.</p>	6	 <p>Water from heavy seas or water projected in powerful jets shall not enter the enclosure in harmful quantities.</p>
<p>Rating Example:</p> <p>IP65</p> <p>INGRESS PROTECTION</p>		7	 <p>Protection against the effects of immersion in water between 15 cm and 1 m for 30 minutes.</p>
		8	 <p>Protection against the effects of immersion in water under pressure for long periods.</p>

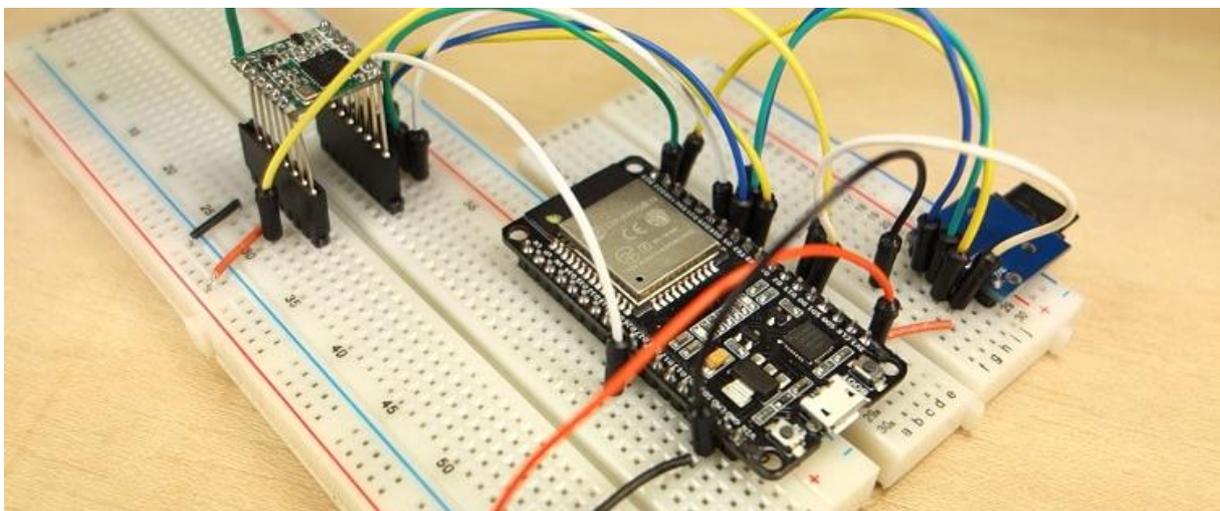
Here's how our enclosure looks like with all the circuitry inside.



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.

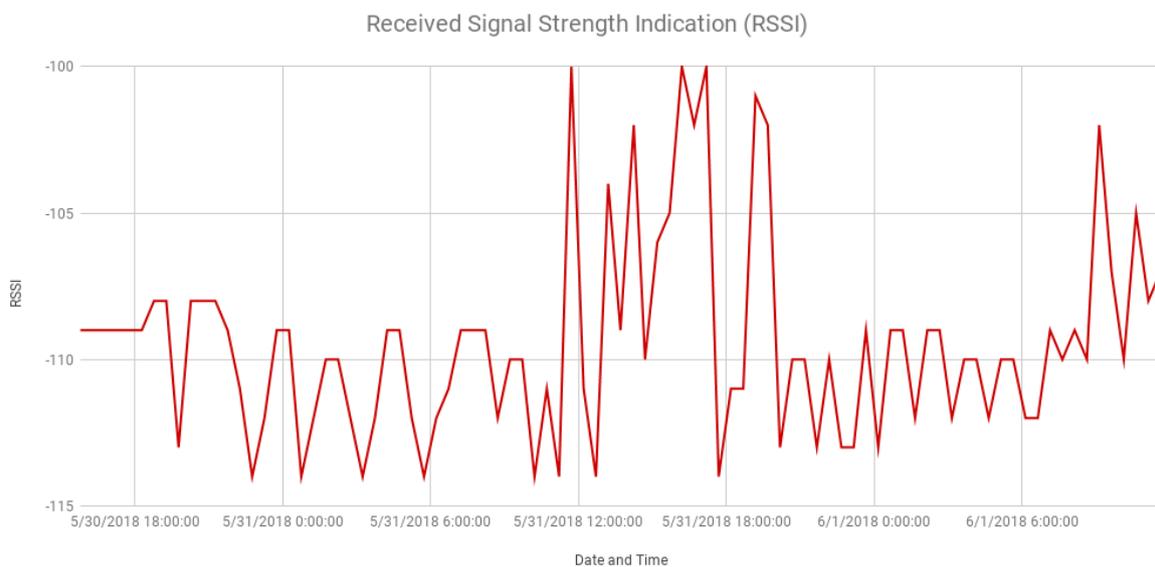


Communication Range

We've placed the LoRa sender in our field at approximately 130 meters from the receiver and it's been receiving 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 are 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 in 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 it with bright sun.

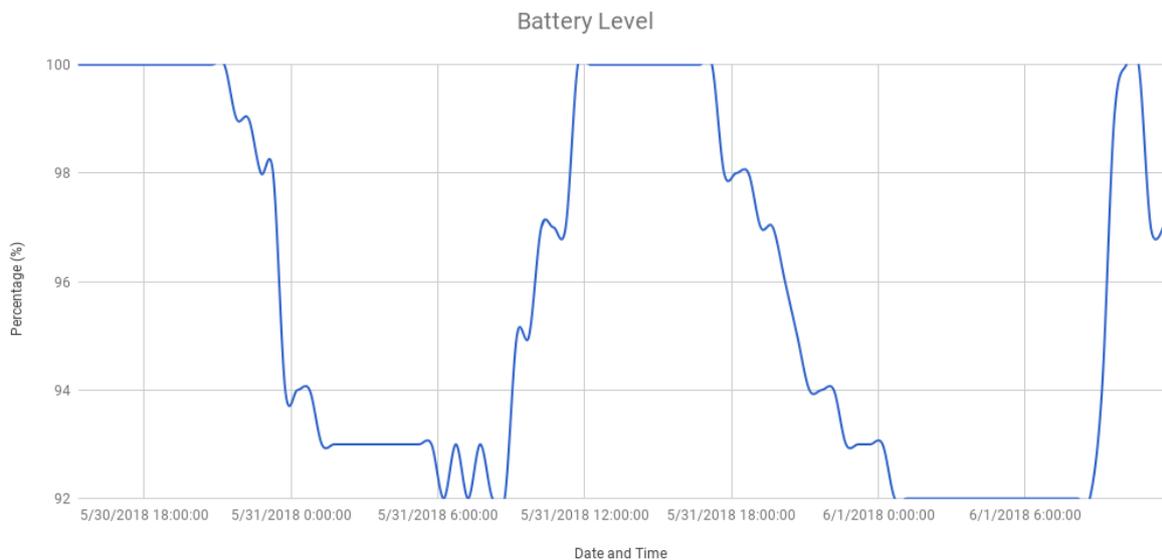
To charge your battery faster you can add more solar panels in parallel.



Our battery has 3800mAh. If you want to ensure you have enough power for many 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.



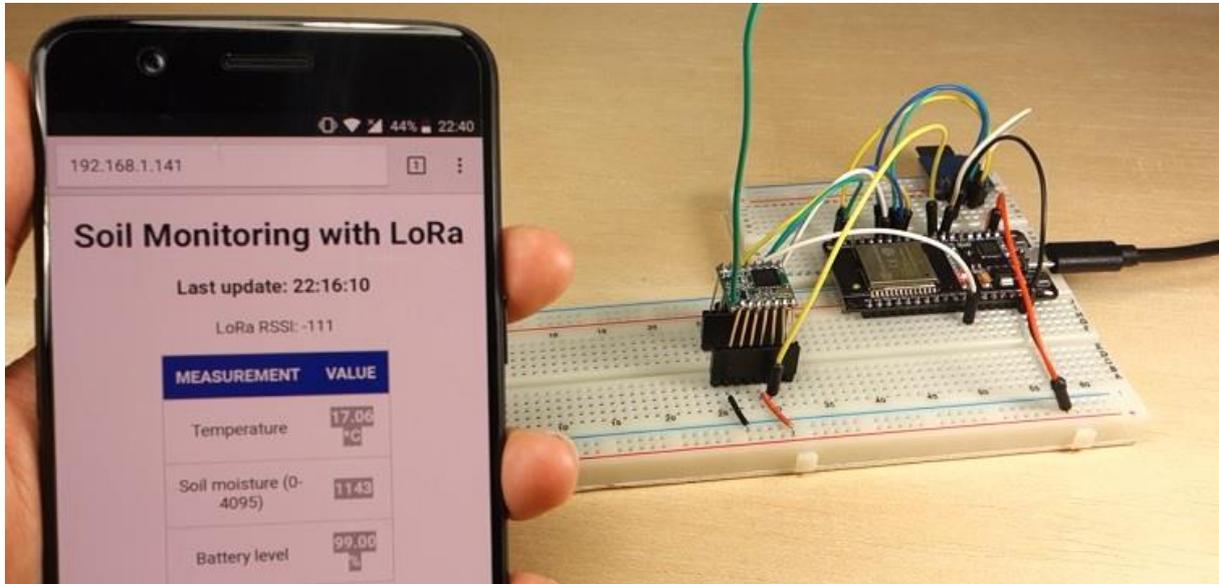
During the night, the battery discharges to about 92%. At 6:00 AM, when the sun rises, the battery starts charging. At 12:00 the battery is fully charged and keeps fully charged until 18:00 (6PM). After that, it starts decreasing again. Throughout these experimenting days, the battery never reached less than 92%. The weather in these days was clouds with light showers.



Testing the Web Server

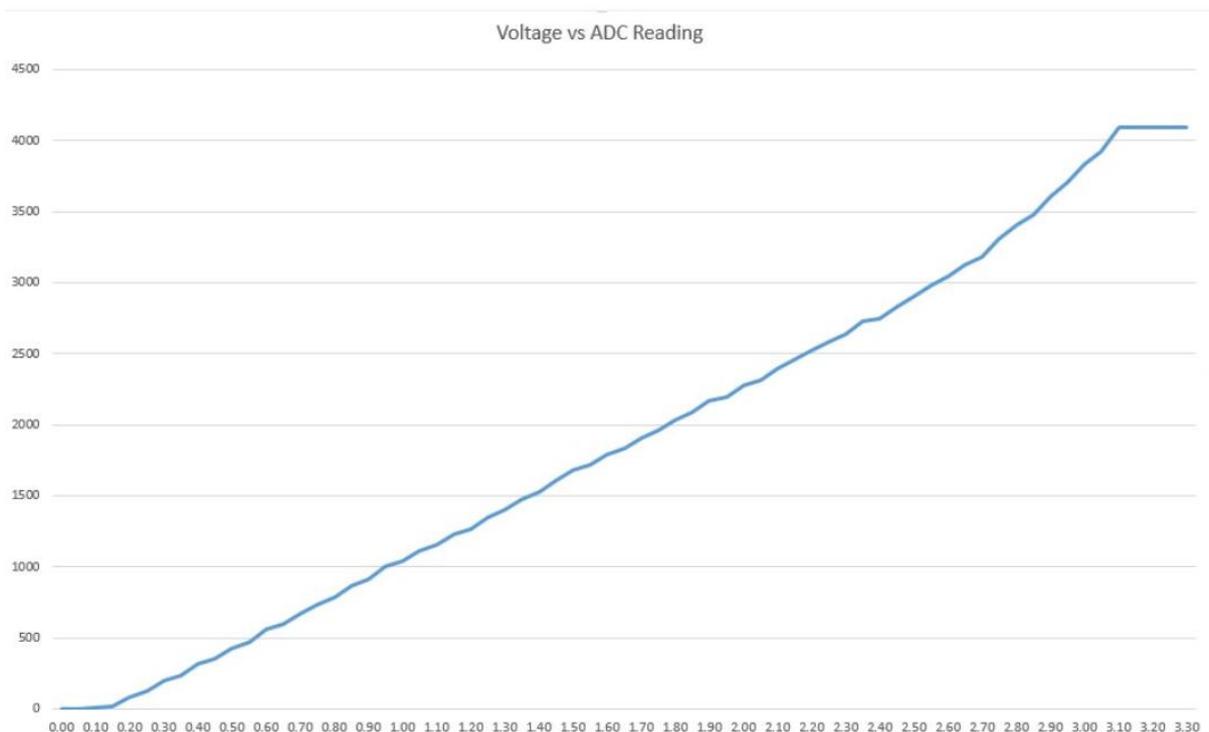
Every time you want to check the latest sensor readings, go to the ESP32 IP address to access the web server.

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 make some math to convert these values to percentage that may make more sense for what you want to do.

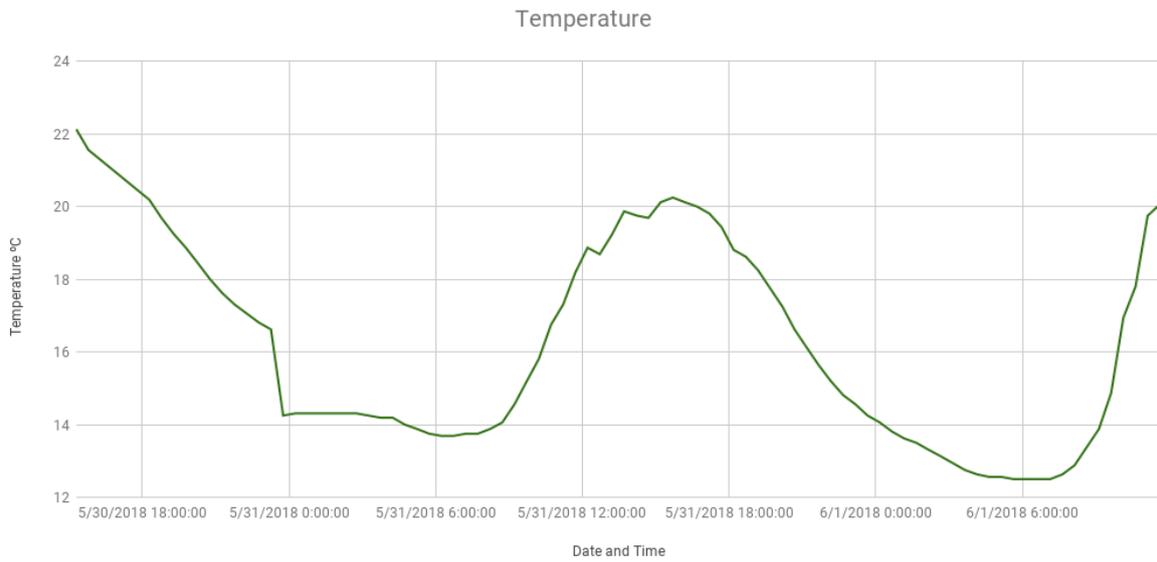
Accessing the MicroSD Card Data

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**.

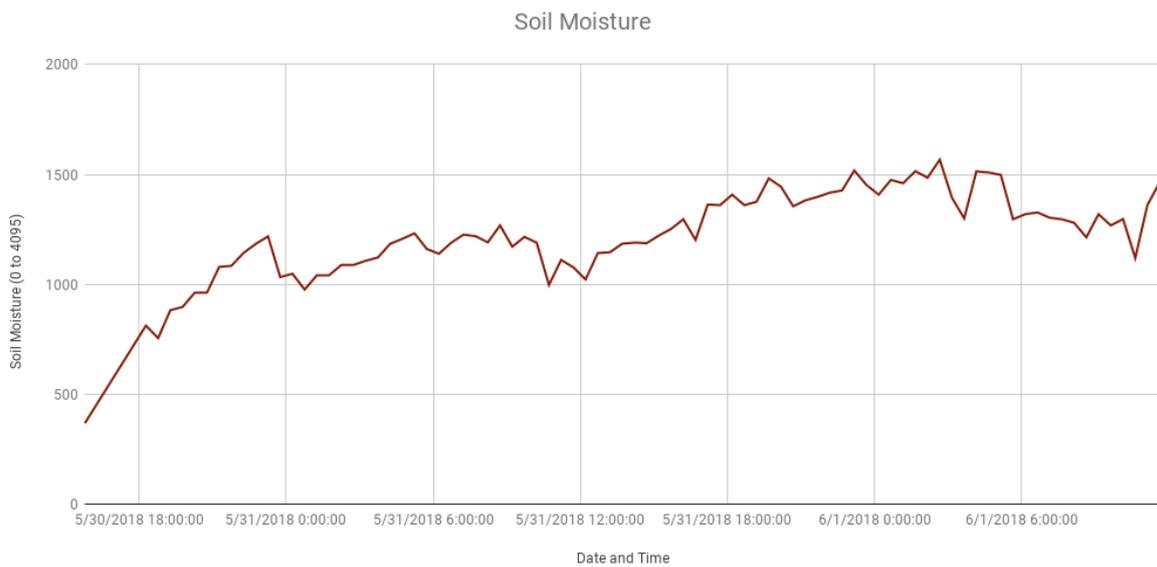
	A	B	C	D	E	F	G
1	Reading ID	Date	Hour	Temperature	Soil Moisture (0-4095)	RSSI	Battery Level (0-100)
2	0	5/30/2018	3:18:02 PM	22.13	4095	-60	100
3	1	5/30/2018	3:47:55 PM	21.56	368	-109	100
4	6	5/30/2018	6:17:15 PM	20.19	812	-109	100
5	7	5/30/2018	6:47:05 PM	19.69	755	-108	100
6	8	5/30/2018	7:16:57 PM	19.25	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 data about temperature 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.



EXTRA UNITS

ESP32 Static/Fixed IP Address

If you're running a web server or Wi-Fi client with your ESP32 and every time you restart your board, the ESP32 has a new IP address, you can follow this Unit to assign a static/fixed IP address to your ESP32 board.



Parts Required:

Parts required for this Unit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- [Code – ESP32 Fixed/Static IP Address](#)
- [Code – Print ESP32 MAC Address](#)

Static/Fixed IP Address Sketch

To show you how to fix your ESP32 IP address, we'll use the "ESP32 Web Sever – Control Outputs" code as an example. By the end of our explanation you should be able to fix your IP address regardless of the web server or Wi-Fi project you're building.

Copy the code below to your Arduino IDE, but don't upload it yet. You need to make some changes to make it work for you.

Note: if you upload the next sketch to your ESP32 board, it should automatically assign the fixed IP address **192.168.1.184**.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_Fixed_Static_IP_Address/ESP32_Fixed_Static_IP_Address.ino

```

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

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

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

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

// Variable to store the HTTP request
String header;

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

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

// 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, 0, 0);
IPAddress primaryDNS(8, 8, 8, 8); //optional
IPAddress secondaryDNS(8, 8, 4, 4); //optional

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);

  // Configures static IP address
  if (!WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS)) {
    Serial.println("STA Failed to configure");
  }

  // 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,
    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();

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

            // Display the HTML web page
            client.println("<!DOCTYPE html><html>");
            client.println("<head><meta
            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
            name=\"viewport\"

```

```

        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>");

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

        // Display current state, and ON/OFF buttons for GPIO 26
        client.println("<p>GPIO 26 - State " + output26State + "</p>");
        // If the output26State is off, it displays the ON button
        if (output26State=="off") {
            client.println("<p><a href=\\\"/26/on\\\"><button
class=\\\"button\\\">ON</button></a></p>");
        } else {
            client.println("<p><a href=\\\"/26/off\\\"><button class=\\\"button
button2\\\">OFF</button></a></p>");
        }

        // Display current state, and ON/OFF buttons for GPIO 27
        client.println("<p>GPIO 27 - State " + output27State + "</p>");
        // If the output27State is off, it displays the ON button
        if (output27State=="off") {
            client.println("<p><a href=\\\"/27/on\\\"><button
class=\\\"button\\\">ON</button></a></p>");
        } else {
            client.println("<p><a href=\\\"/27/off\\\"><button class=\\\"button
button2\\\">OFF</button></a></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("");
}
}

```

Setting Your Network Credentials

You need to modify the following lines with your network credentials: SSID and password:

```

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

```

Setting your Static IP Address

Then, outside the `setup()` and `loop()` functions, you define the following variables with your own static IP address and corresponding gateway IP address.

By default, the next code assigns the IP address **192.168.1.184** that works in the gateway **192.168.1.1**.

```
// 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, 0, 0);
IPAddress primaryDNS(8, 8, 8, 8); //optional
IPAddress secondaryDNS(8, 8, 4, 4); //optional
```

The parameters highlighted in red in the previous snippet are the ones you need to change if you want to assign your desired IP address to the ESP32.

Important: you need to use an available IP address in your local network and the corresponding gateway.

setup()

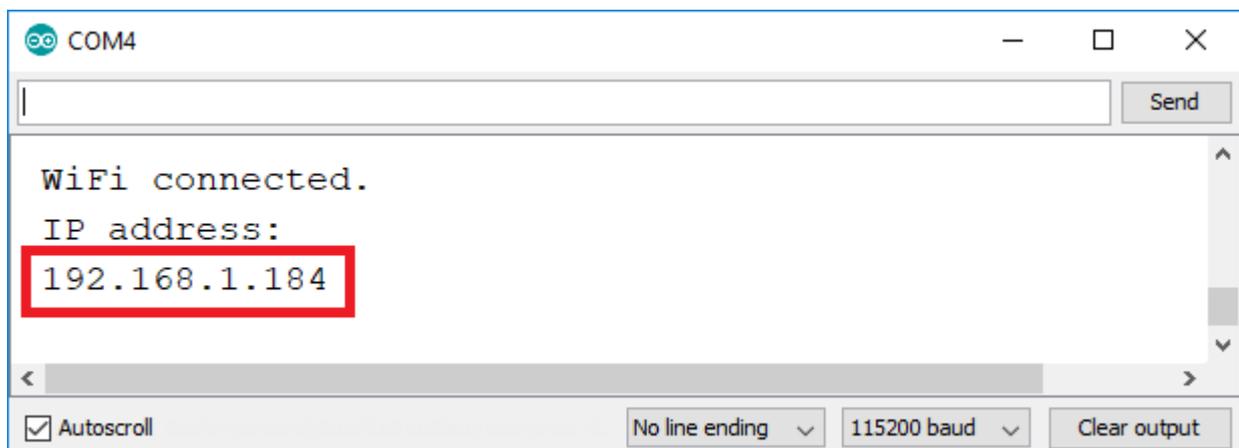
In the `setup()` you need to call the `WiFi.config()` method to assign the configurations to your ESP32.

```
if (!WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS)) {
    Serial.println("STA Failed to configure");
}
```

Note: the `primaryDNS` and `secondaryDNS` parameters are optional and you can remove them.

Testing

After uploading the code to your board, open the Arduino IDE Serial Monitor at the baud rate 115200, restart your ESP32 board and the IP address defined earlier should be assigned to your board.



As you can see, it prints the IP address **192.168.1.184**.

You can take this example and add it to all your Wi-Fi sketches to assign a fixed IP address to your ESP32.

Assigning IP Address with MAC Address

If you've tried to assign a fixed IP address to the ESP32 using the previous example and it doesn't work, we recommend assigning an IP address directly in your router settings through the ESP32 MAC Address.

SOURCE CODE

[https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Print ESP32 MAC Address/Print ESP32 MAC Address.ino](https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Print%20ESP32%20MAC%20Address/Print%20ESP32%20MAC%20Address.ino)

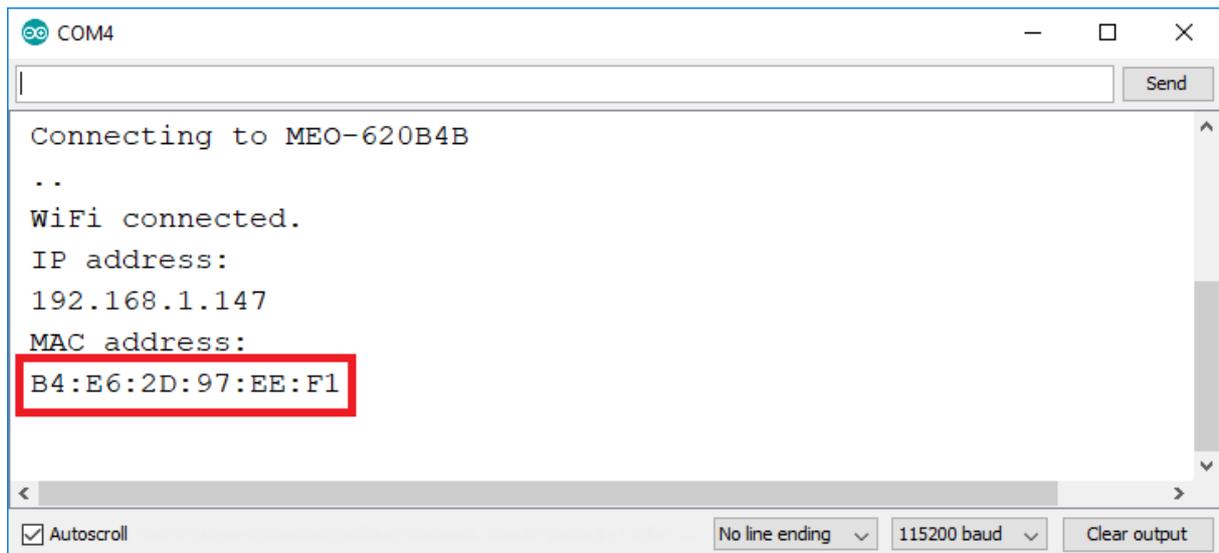
Add your network credentials (SSID and password). Then, upload the next code to your ESP32:

```
/******  
  Rui Santos  
  Complete project details at https://randomnerdtutorials.com  
*****/  
  
// Load Wi-Fi library  
#include <WiFi.h>  
  
// Replace with your network credentials  
const char* ssid      = "REPLACE_WITH_YOUR_SSID";  
const char* password  = "REPLACE_WITH_YOUR_PASSWORD";  
  
// Set web server port number to 80  
WiFiServer server(80);  
  
void setup() {  
  Serial.begin(115200);  
  
  // 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();  
  
  // Print ESP MAC Address  
  Serial.println("MAC address: ");
```

```
Serial.println(WiFi.macAddress());  
}  
  
void loop() {  
  // put your main code here, to run repeatedly:  
}
```

In the `setup()`, after connecting to your network, it prints the ESP32 MAC Address in the Serial Monitor:

```
// Print ESP MAC Address  
Serial.println("MAC address: ");  
Serial.println(WiFi.macAddress());
```



In our case, the ESP32 MAC Address is **B4:E6:2D:97:EE:F1**. Copy the MAC Address, because you'll need it in just a moment.

Router Settings

If you login into your router admin page, there should be a page/menu where you can assign an IP address to a network device. Each router has different menus and configurations. So, we can't provide instructions on how do to it for all the routers available.

We recommend Googling "assign IP address to MAC address" followed by your router name. You should find some instructions that show how to assign the IP to a MAC address for your specific router.

In summary, if you go to your router configurations menu, you should be able to assign your desired IP address to your ESP32 MAC address (for example B4:E6:2D:97:EE:F1).

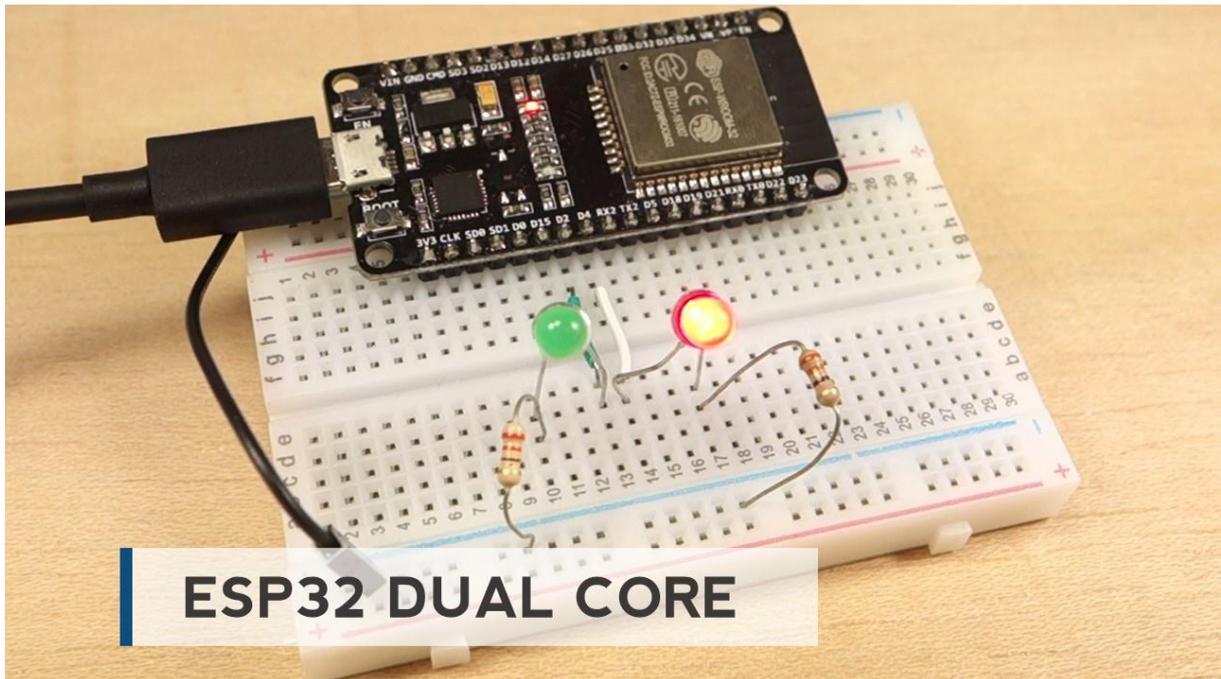
Wrapping Up

After completing this Unit, you should be able to assign a fixed/static IP address to your ESP32.

ESP32 Dual Core – Create Tasks

The ESP32 comes with 2 Xtensa 32-bit LX6 microprocessors: core 0 and core 1. So, it is dual core. When we run code on Arduino IDE, by default, it runs on 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, and make your ESP32 multitasking.

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



Parts required:

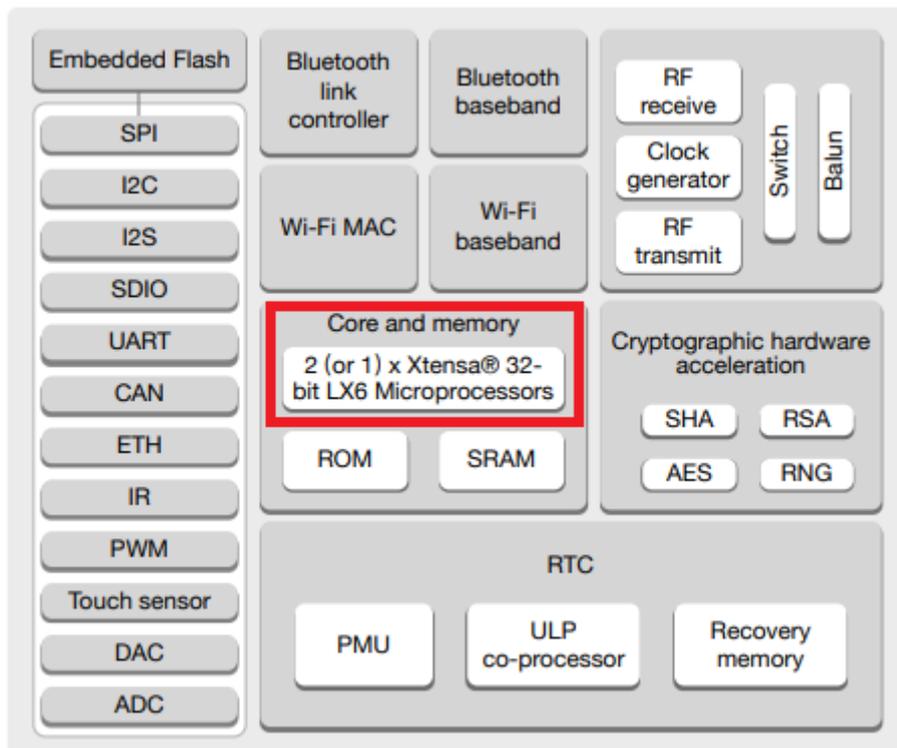
Here's a list of the parts required for this Unit:

- [ESP32 DOIT DEVKIT V1 Board](#)
- 2x [5mm LED](#)
- 2x [330 Ohm resistor](#)
- [Breadboard](#)
- [Jumper wires](#)

Introduction

The ESP32 comes with 2 Xtensa 32-bit LX6 microprocessors, so it's dual core:

- Core 0
- Core 1



When we run code on Arduino IDE, by default it runs on core 1. In this Unit we'll show you how to use the ESP32 second core by creating tasks. You can run pieces of code simultaneously on both cores, and make your ESP32 multitasking (note that you don't necessarily need to run dual core to achieve multitasking).

When we upload code to the ESP32 using the Arduino IDE, it just runs - we don't have to worry which core executes the code.

There's a function that you can use to identify 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.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Dual_Core/Get_Core_ID/Get_Core_ID.ino

```

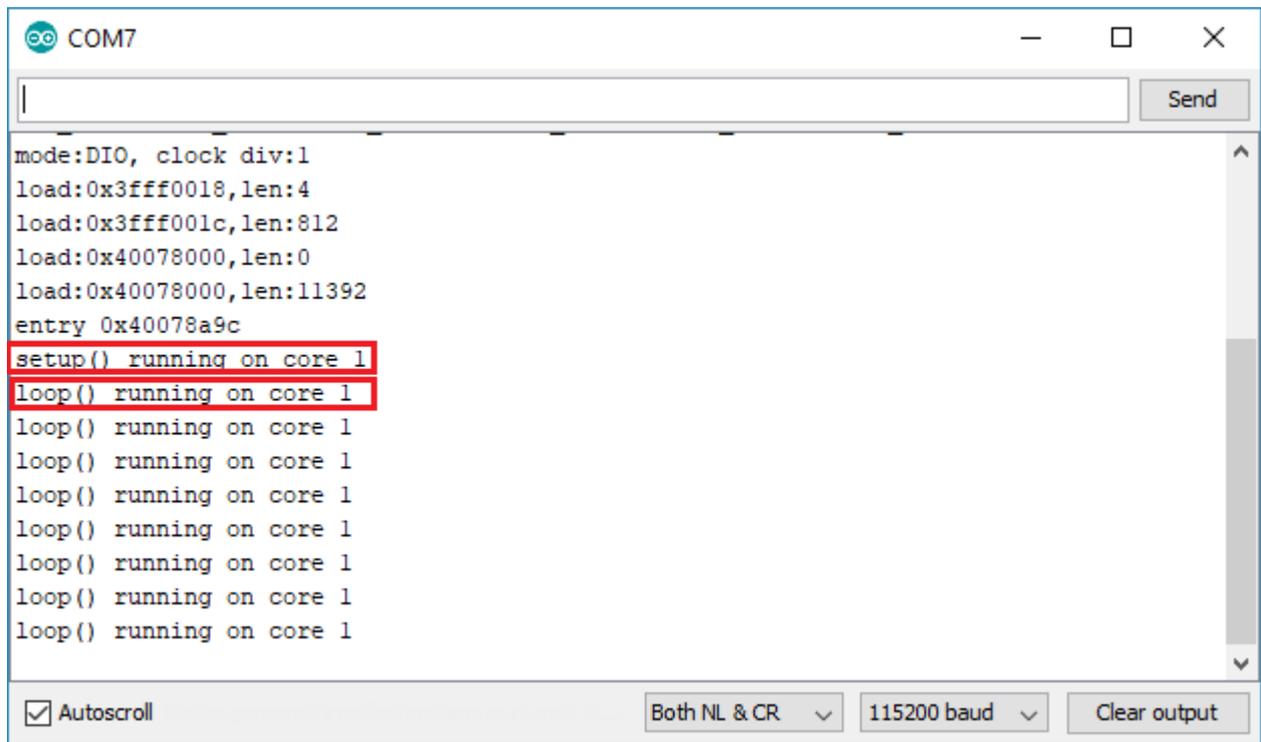
/*****
  Rui Santos
  Complete project details at http://randomnerdtutorials.com
*****/

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());
}
```



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, measuring 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. That 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(;;)` 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.

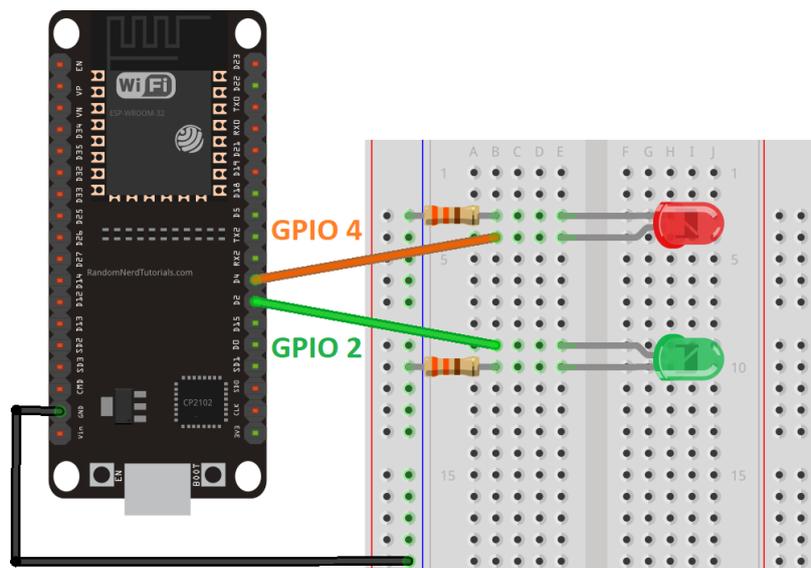
If during your code execution you want to delete the created task, you can use the `vTaskDelete()` function, that accepts the task handle (Task1) as argument:

```
vTaskDelete(Task1);
```

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

Create Tasks in Different Cores – Example

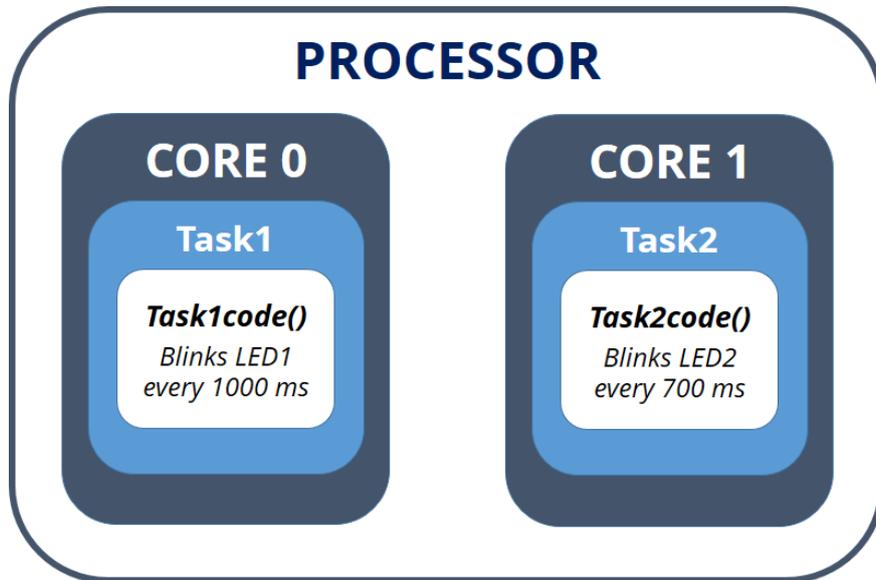
To create different tasks running on different cores we'll create two tasks that blink LEDs with different delay times. Wire two LEDs to the ESP32 as shown in the following diagram:



(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.)

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:

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/Dual_Core/Dual_Core_Blinking_LEDs/Dual_Core_Blinking_LEDs.ino

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

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, /* handle to keep track of task */
```

```

        0);          /* pin task to core0*/

    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,       /* handle to keep track of 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 the Code Works

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. So, you could 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.

However, 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,      /* handle to keep track of task */
    0);          /* pin task to core0*/
```

`Task1` will be implemented with the `Task1code()` function. So, we need to create that function later on 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,      /* handle to keep track of task */
    1);          /* pin task to core 1 */
```

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

```
//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);
  }
}
```

The function to 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:

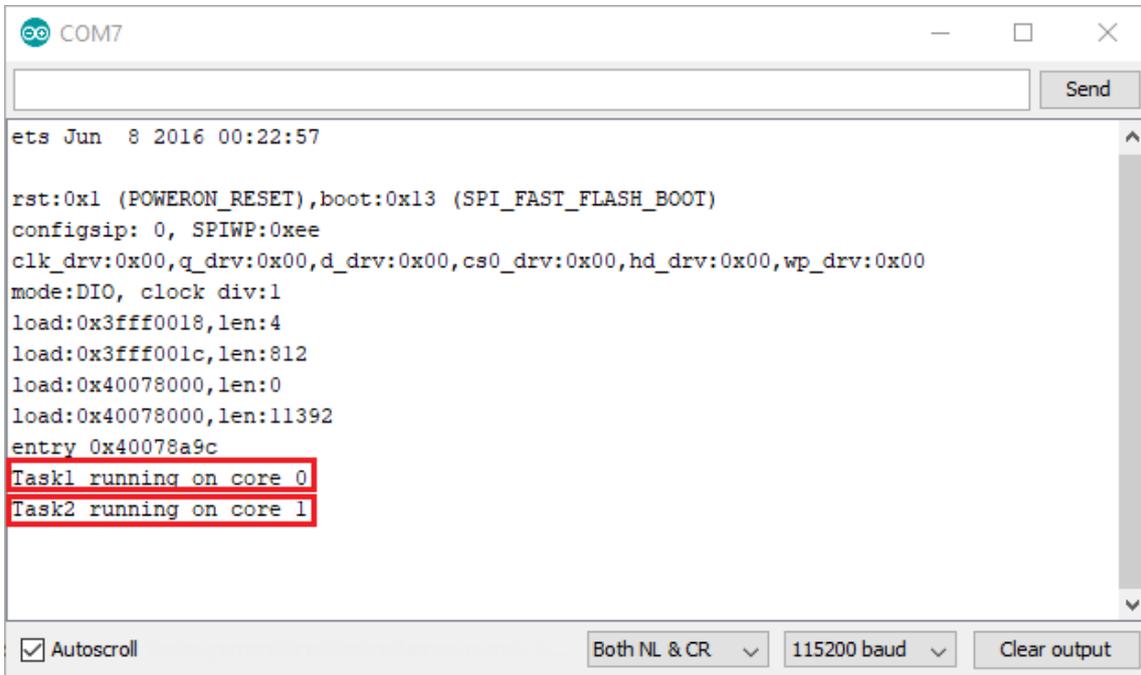
```
void loop() {
}
```

Note: as mentioned previously, the Arduino `loop()` runs on core 1. So, instead of creating a task to run on core 1, you can simply write your code inside the `loop()`.

Demonstration

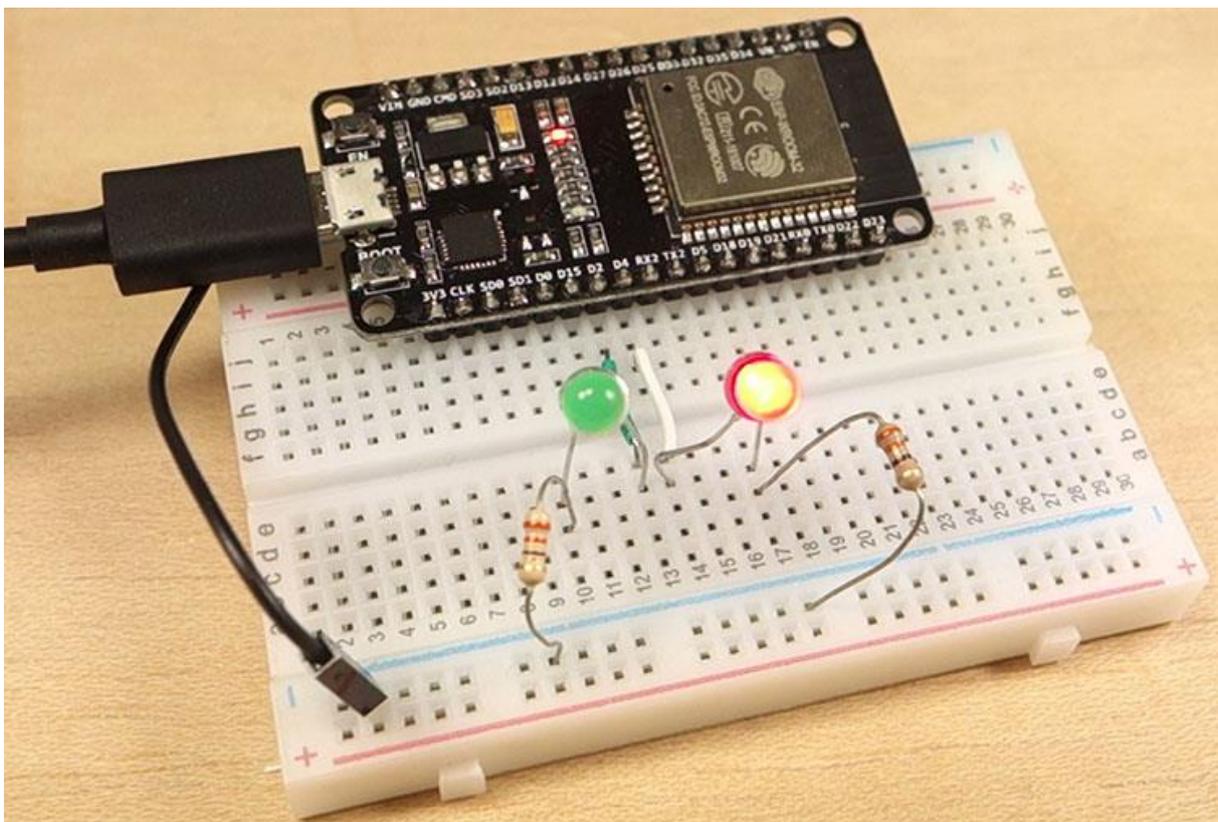
Upload the code to your ESP32. Make sure you have the right board and COM port selected.

Open the Serial Monitor at a baud rate of 115200. You should get the following messages:



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

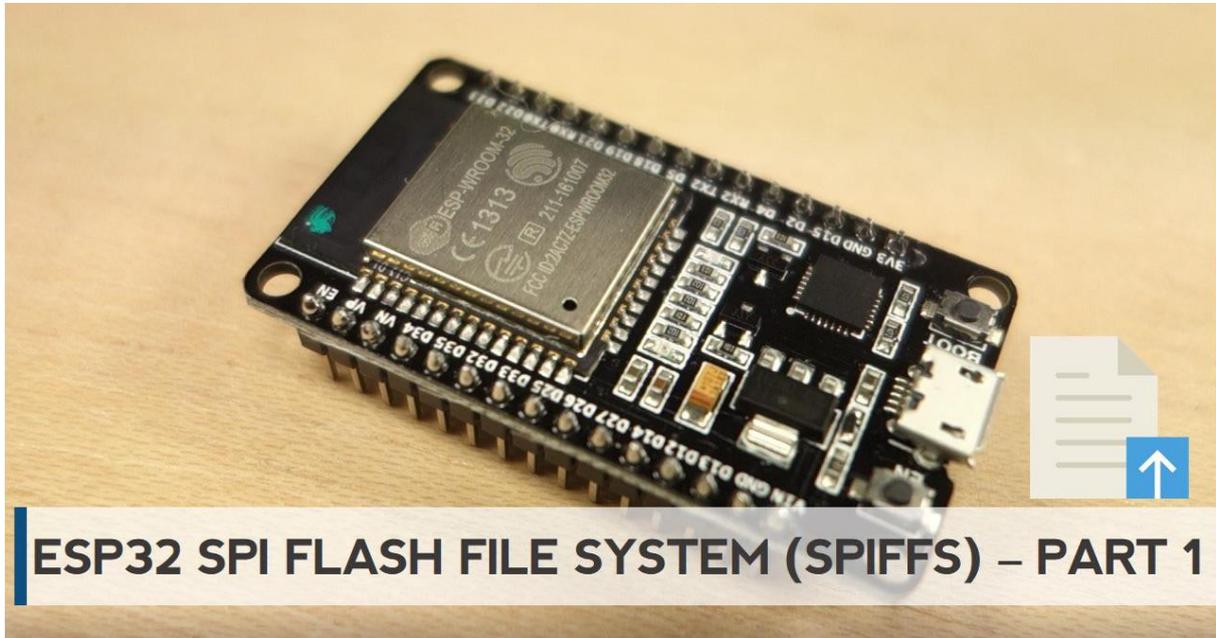
In summary:

- 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 other to publish those readings on a home automation system.

ESP32 SPIFFS (SPI Flash File System)

In this Unit we're going to show how to use the ESP32 SPIFFS: 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.



Finally, we'll also show you how to build a web server using files from the filesystem – instead of having to write the HTML and CSS code directly on the Arduino sketch, we'll use separate files saved on the filesystem (part 2).

Introducing SPIFFS

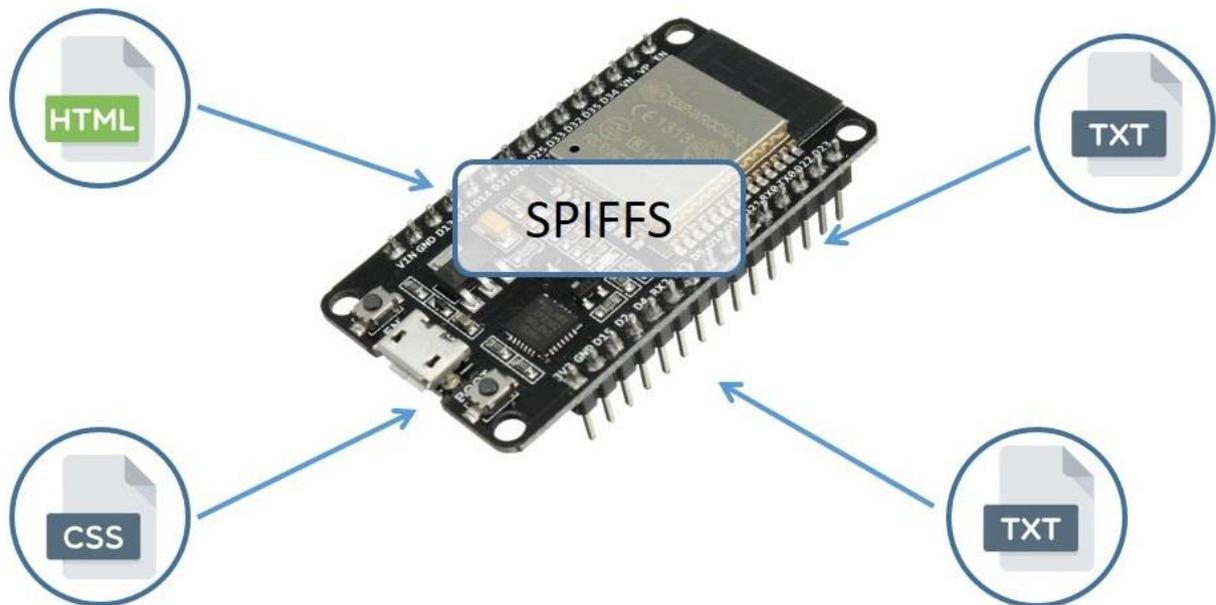
SPIFFS stands for Serial Peripheral Interface Flash File System. SPIFFS is a lightweight filesystem created for microcontrollers with a flash chip, which are connected by SPI bus, like the ESP32 flash memory.

SPIFFS lets you access the flash memory like you would do in a normal filesystem in your computer, but simpler and more limited. You can read, write, close, and delete files. At the time of writing this Unit, SPIFFS doesn't support directories, so everything is saved on a flat structure, therefore you can't create folders.

SPIFFS Applications

Using SPIFFS with the ESP32 is specially 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 and CSS files to build a web server;
- And much more.



Until now, we've written the HTML code for the web server as a String directly on the Arduino sketch. With SPIFFS, you can write the HTML and CSS in a separated file and save them on the ESP32 filesystem.

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 really easy and simple to work with files. Let's install it.

First, make sure you have the latest Arduino IDE installed, and you have the ESP32 add-on for the Arduino IDE. Follow the next steps to install the file uploader:

- 1) Go to the [releases page](#) and click the [ESP32FS-1.0.zip](#) file to download.

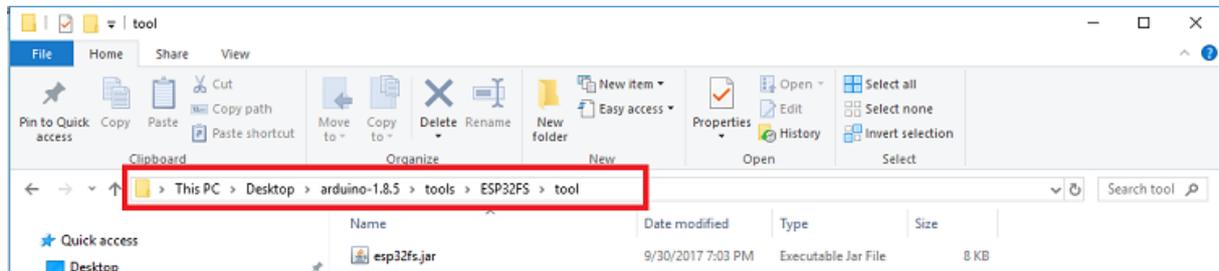
The screenshot shows a GitHub release page for 'esptool_py'. On the left, there is a 'Latest release' badge, the version number '1.0', a commit hash '278ffa0', and a 'Verified' badge. The main title is 'Release for esptool_py' in blue, with the text 'me-no-dev released this on Jan 15' below it. A description reads: 'Updates the path to esptool to work with the latest Arduino for ESP32'. Under the 'Assets' section, there are three items: 'ESP32FS-1.0.zip', 'Source code (zip)', and 'Source code (tar.gz)'. Each item has a corresponding icon (zip file, document, and document).

2) Go to the Arduino IDE directory, and open the **Tools** folder.

```
> arduino-1.8.5 > tools
```

3) Unzip the downloaded .zip folder to the **Tools** folder. You should have a similar folder structure:

```
<home_dir>/Arduino-<version>/tools/ESP32FS/tool/esp32fs.jar
```

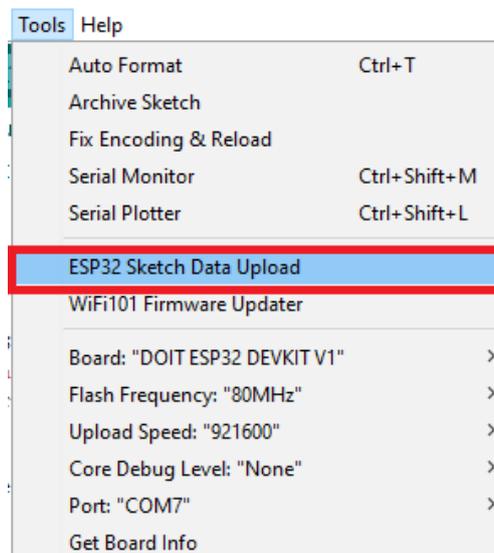


4) Finally, restart your Arduino IDE.

Note: the latest instructions can also be found on the next link:

- <https://github.com/me-no-dev/arduino-esp32fs-plugin>

To check if the plugin was successfully installed, open your Arduino IDE. Select an ESP32 board, go to **Tools** and check that you have the “**ESP32 Sketch Data Upload**” option.

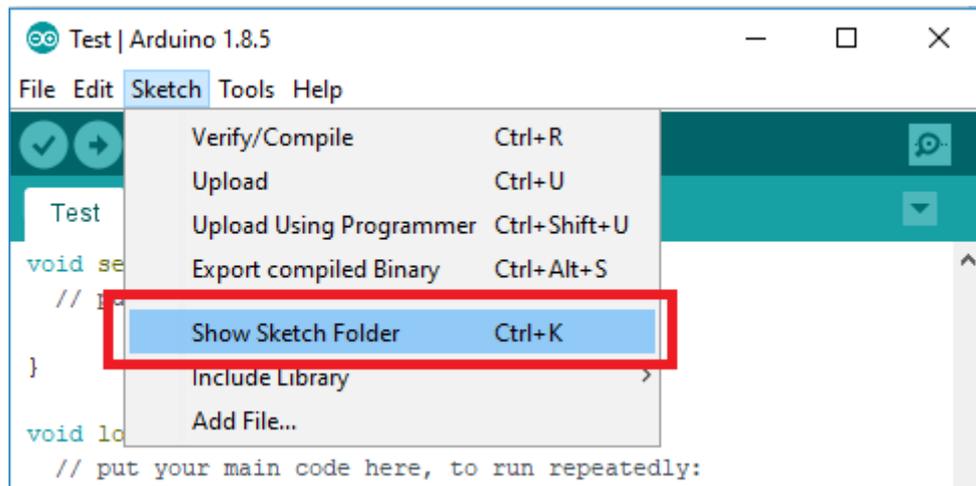


Uploading Files using the Filesystem Uploader

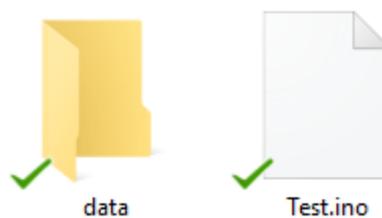
To upload files to the ESP32 filesystem follow the next instructions.

1) Create an Arduino sketch and save it. For demonstration purposes, you can save an empty sketch.

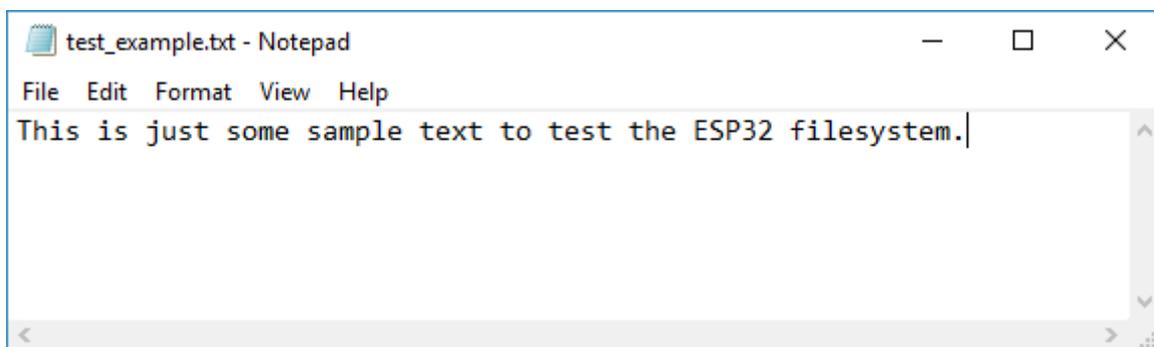
2) Then, open the sketch folder. You can go to **Sketch** ▶ **Show Sketch Folder**. The folder where your sketch is saved should open.



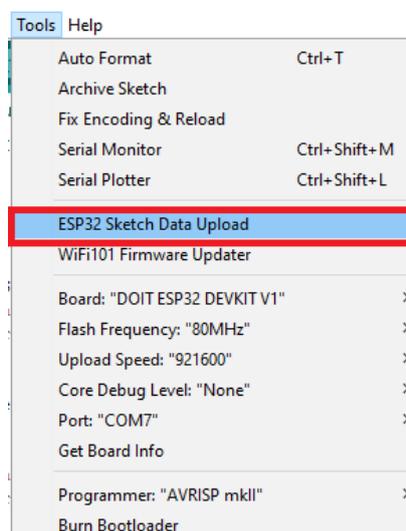
3) Inside that folder, create a new folder called **data**.



4) Inside the **data** folder is where you should put the files you want to be saved into the ESP32 filesystem. As an example, create a **.txt** file with some text called **test_example**.



5) Then, to upload the files, in the Arduino IDE, you just need to go to **Tools** ▶ **ESP32 Sketch Data Upload**.



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.

```
SPIFFS Uploading Image...
[SPIFFS] page   : 256
[SPIFFS] block  : 4096
/desktop.ini

/test_example.txt

[SPIFFS] upload : C:\Users\Sara\AppData\Local\Temp\arduino_build_516333/Test.spiffs.bin
[SPIFFS] address: 2691072
[SPIFFS] port   : COM7
[SPIFFS] speed  : 921600
[SPIFFS] mode   : dio
[SPIFFS] freq   : 80m

esptool.py v2.1
Connecting.....____.....
DOIT ESP32 DEVKIT V1, 80MHz, 921600, None on COM7
```

When you see the message “**SPIFFS Image Uploaded**”, the files were successfully uploaded to the ESP32 filesystem.

```
SPIFFS Image Uploaded
Hash of data verified.

Leaving...

Hard resetting...
DOIT ESP32 DEVKIT V1, 80MHz, 921600, None on COM7
```

Testing the Uploader

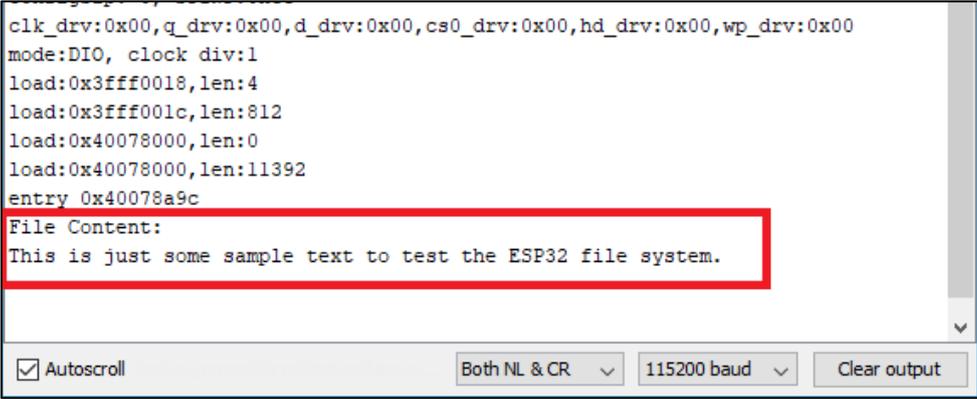
Now, let's just check if the file was actually saved into the ESP32 filesystem. Simply upload the following code to your ESP32 board.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/SPIFFS/SPIFFS_Test/SPIFFS_Test.ino

```
/******  
  Rui Santos  
  Complete project details at https://randomnerdtutorials.com  
*****/  
  
#include "SPIFFS.h"  
  
void setup() {  
  Serial.begin(115200);  
  
  if(!SPIFFS.begin(true)){  
    Serial.println("An Error has occurred while mounting SPIFFS");  
    return;  
  }  
  
  File file = SPIFFS.open("/test_example.txt");  
  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 **“ENABLE”** button. It should print the content of your `.txt` file on the Serial Monitor.



```
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  
File Content:  
This is just some sample text to test the ESP32 file system.  
  
 Autoscroll    Both NL & CR    115200 baud    Clear output
```

Congratulations! You've successfully uploaded files to the ESP32 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 check the file size. The next sketch shows how to do that.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/SPIFFS/SPIFFS_Manipulating_Files/SPIFFS_Manipulating_Files.ino

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

#include "FS.h"
#include "SPIFFS.h"

// Create a File object to manipulate your file
File myFile;

// File path
const char* myFilePath = "/new_file.txt";

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

  // Initialize SPIFFS
  if(!SPIFFS.begin(true)) {
    Serial.println("Error while mounting SPIFFS");
    return;
  }

  // Open file and write data to it
  myFile = SPIFFS.open(myFilePath, FILE_WRITE);
  if (myFile.print("Example message in write mode")) {
    Serial.println("Message successfully written");
  }
  else {
    Serial.print("Writing message failed!!");
  }
  myFile.close();

  // Append data to file
  myFile = SPIFFS.open(myFilePath, FILE_APPEND);
  if(myFile.print(" - Example message appended to file")) {
    Serial.println("Message successfully appended");
  }
  else {
    Serial.print("Appending failed!");
  }
  myFile.close();

  // Read file content
  myFile = SPIFFS.open(myFilePath, FILE_READ);
```

```

Serial.print("File content: \");
while(myFile.available()) {
  Serial.write(myFile.read());
}
Serial.println("\n");

// Check file size
Serial.print("File size: ");
Serial.println(myFile.size());

myFile.close();

// Delete file
if(SPIFFS.remove(myFilePath)){
  Serial.println("File successfully deleted");
}
else{
  Serial.print("Deleting file failed!");
}
}
void loop(){
}

```

Note: for more advanced examples on how to manipulate the files, check the SPIFFS library example code. It has several functions you can copy to your sketch to manipulate the files. You can take a look at those functions on the following link:

- https://github.com/esp8266/arduino-esp32/blob/master/libraries/SPIFFS/examples/SPIFFS_Test/SPIFFS_Test.ino

Let's take a look at code. It uses the SPIFFS and the FS libraries:

```

#include "FS.h"
#include "SPIFFS.h"

```

First, create a File object to manipulate the file. In this case, it is called `myFile`, but you can call it whatever you want.

```
File myFile;
```

Create a const char* variable with the path to your file:

```
const char* myFilePath = "/new_file.txt";
```

In the `setup()`, initialize the Serial Monitor and SPIFFS:

```

Serial.begin(115200);

// Initialize SPIFFS
if(!SPIFFS.begin(true)){
  Serial.println("Error while mounting SPIFFS");
  return;
}

```

Write data to a file

To write data to a file, first open the file in writing mode using the `open()` method. It takes as first argument the path of the file. As second argument pass `FILE_WRITE` to indicate that the file will be opened in writing mode.

Note: if the file doesn't exist, the file will be create automatically.

```
myFile = SPIFFS.open(myFilePath, FILE_WRITE);
```

To write text to the file, use the `print()` method on the `File` object.

```
if (myFile.print("Example message in write mode")){
```

We've aded an `if... else` statement in-between to be sure the content was written to the file. After successfully writing some data to the file, close the file using the `close()` method.

```
myFile.close();
```

Append data to a file

To append data to a file, open it file in append mode:

```
myFile = SPIFFS.open(myFilePath, FILE_APPEND);
```

Then, write to the file using the `print()` method:

```
myFile.print(" - Example message appended to file")
```

Read the file content

To read the file content, open the file in reading mode:

```
myFile = SPIFFS.open(myFilePath, FILE_READ);
```

Then, use the `read()` method to read the bytes from the file. While there are bytes available to read, print them data in the Serial Monitor:

```
while(myFile.available()) {  
    Serial.write(myFile.read());  
}
```

Checking the file size

To check the file size use the `size()` method:

```
Serial.println(myFile.size());
```

Delete a file

To delete a file use the `remove()` method and pass as argument the file path:

```
SPIFFS.remove(myFilePath
```


Build an ESP32 Web Server using Files from Filesystem (SPIFFS)

In the previous Unit 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 "[Module 4, Unit 2: ESP32 Web Server – Control Outputs](#)".

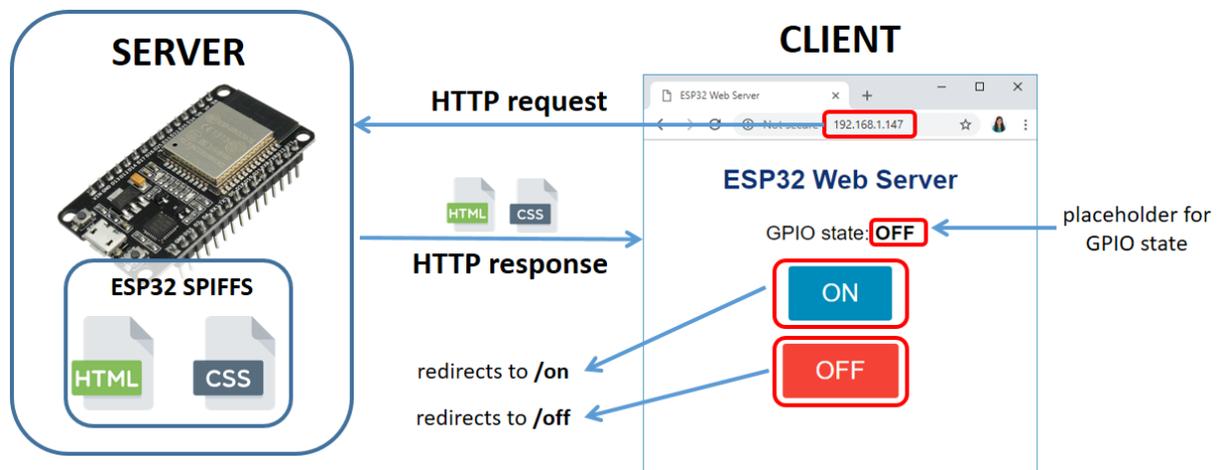
To continue with this Unit you should have the ESP32 Filesystem Uploader plugin installed in your Arduino IDE. If you haven't, follow the previous Unit to install it first.

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 (SPIFFS);
- 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 SPIFFS, you can write the HTML and CSS in separated 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>

Installing the ESPAsyncWebServer library

Follow the next steps to install the [ESPAsyncWebServer](#) library:

- 1) [Click here to download](#) the ESPAsyncWebServer library. You should have a `.zip` folder in your *Downloads* folder
- 2) Unzip the `.zip` folder and you should get `ESPAsyncWebServer-master` folder
- 3) Rename your folder from `ESPAsyncWebServer-master` to `ESPAsyncWebServer`
- 4) Move the `ESPAsyncWebServer` folder to your Arduino IDE installation *libraries* folder

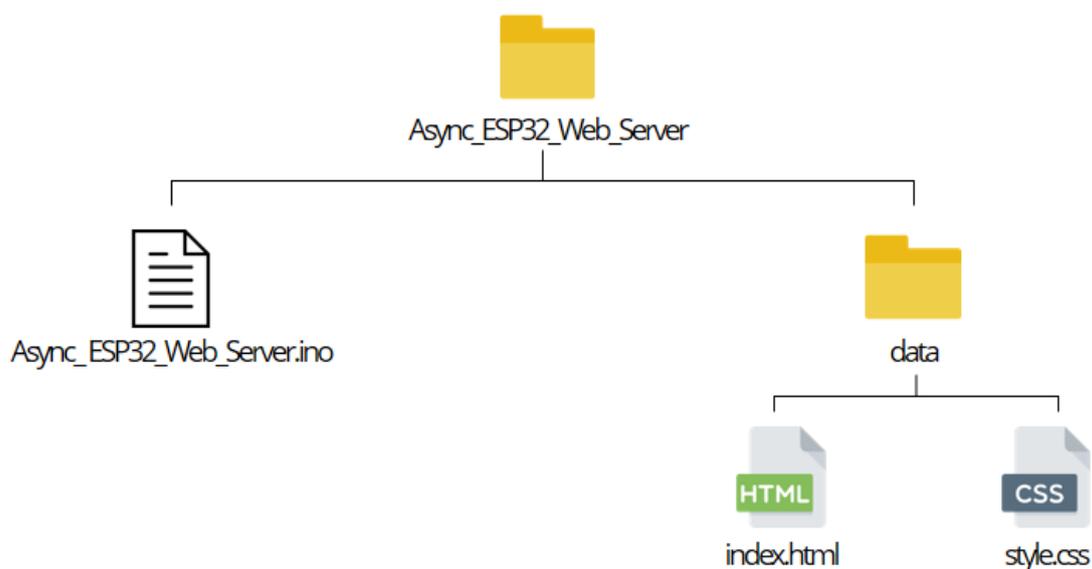
Installing the Async TCP Library for ESP32

The [ESPAsyncWebServer](#) library requires the [AsyncTCP](#) library to work. Follow the next steps to install that library:

- 1) [Click here to download](#) the AsyncTCP library. You should have a `.zip` folder in your Downloads folder
- 2) Unzip the `.zip` folder and you should get `AsyncTCP-master` folder
- 3) Rename your folder from `AsyncTCP-master` to `AsyncTCP`
- 4) Move the `AsyncTCP` folder to your Arduino IDE installation *libraries* folder
- 5) Finally, re-open your Arduino IDE

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 “[Module 4, Unit 3: ESP32 Web Server – HTML and CSS Basics](#)”.

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>` `</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><a href="#">GPIO state: <strong> %STATE% </strong></a></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 to 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.

SOURCE CODE

<https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/SPIFFS/ESP32 Async Web Server/ESP32 Async Web Server.ino>

```
/******  
  Rui Santos  
  Complete project details at https://randomnerdtutorials.com  
*****/  
  
// Import required libraries  
#include "WiFi.h"  
#include "ESPAsyncWebServer.h"  
#include "SPIFFS.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 SPIFFS  
  if(!SPIFFS.begin(true)){  
    Serial.println("An Error has occurred while mounting SPIFFS");  
  }  
}
```

```

    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, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/index.html", String(), false, processor);
});

// Route to load style.css file
server.on("/style.css", HTTP_GET, [](AsyncWebServerRequest
*request){
    request->send(SPIFFS, "/style.css", "text/css");
});

// Route to set GPIO to HIGH
server.on("/on", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, HIGH);
    request->send(SPIFFS, "/index.html", String(), false, processor);
});

// Route to set GPIO to LOW
server.on("/off", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, LOW);
    request->send(SPIFFS, "/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 "SPIFFS.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 `AsyncWebServer` 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 argument the placeholder and should return a `String` that will replace the placeholder. The `processor()` function should have the following structure:

```
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, accordingly 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 and setting the GPIO as an output.

```
Serial.begin(115200);
pinMode(ledPin, OUTPUT);
```

```
// Initialize SPIFFS
if(!SPIFFS.begin(true)){
  Serial.println("An Error has occurred while mounting SPIFFS");
  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, [] (AsyncWebServerRequest *request){
  request->send(SPIFFS, "/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 are able to 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, [] (AsyncWebServerRequest *request){
  request->send(SPIFFS, "/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, [] (AsyncWebServerRequest *request){
  digitalWrite(ledPin, HIGH);
  request->send(SPIFFS, "/index.html", String(), false, processor);
});

// Route to set GPIO to LOW
server.on("/off", HTTP_GET, [] (AsyncWebServerRequest *request){
  digitalWrite(ledPin, LOW);
  request->send(SPIFFS, "/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.

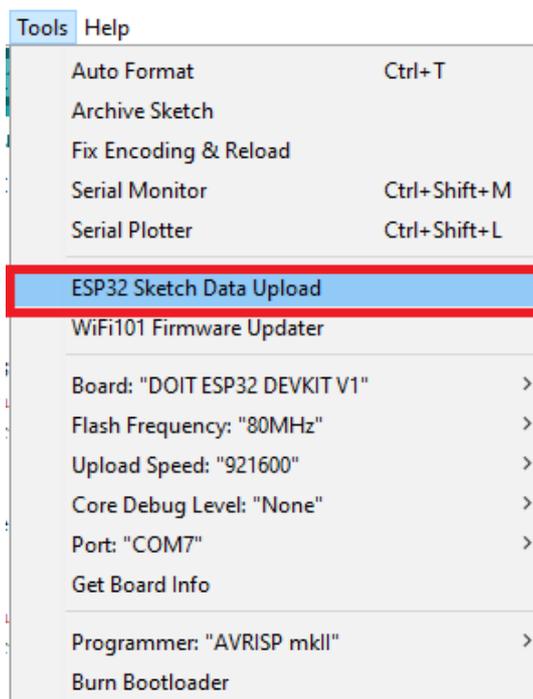
Uploading Code and Files

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.

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 networks credentials to the code.



After uploading the code, you need to upload the files. Go to **Tools** ▶ **ESP32 Data Sketch Upload** and wait for the files to be uploaded.



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.



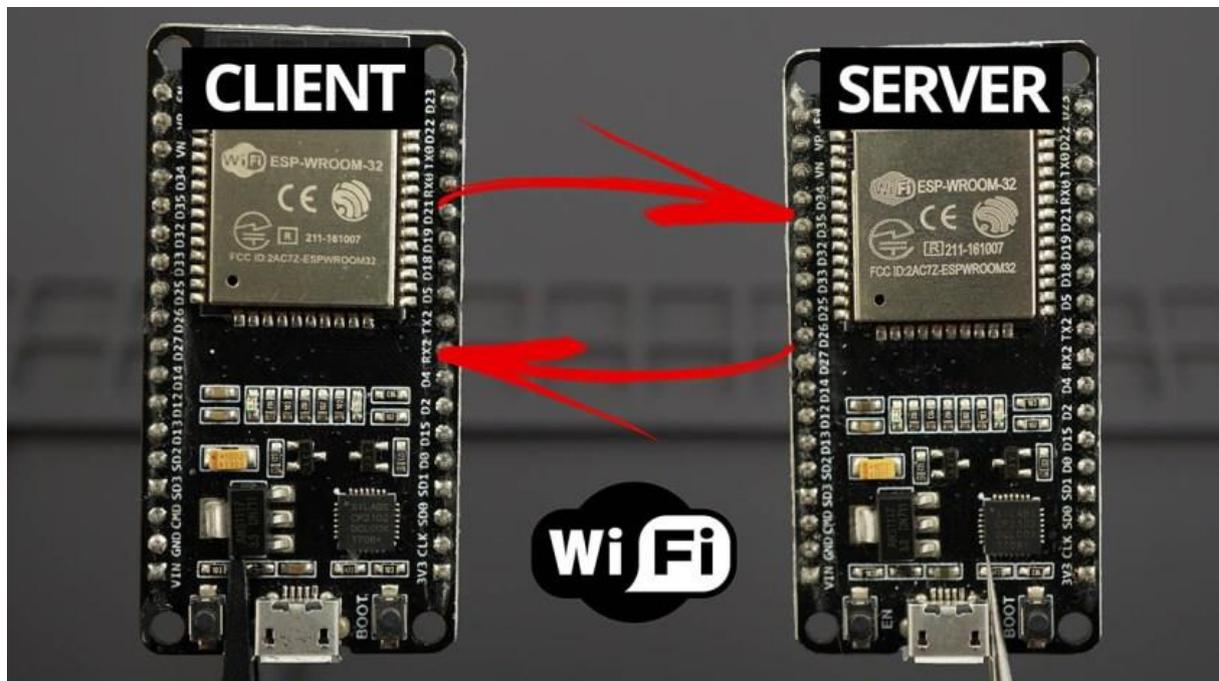
Wrapping Up

Using SPI Flash File System (SPIFFS) is specially 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.

ESP32 Client-Server Wi-Fi

Communication Between Two Boards



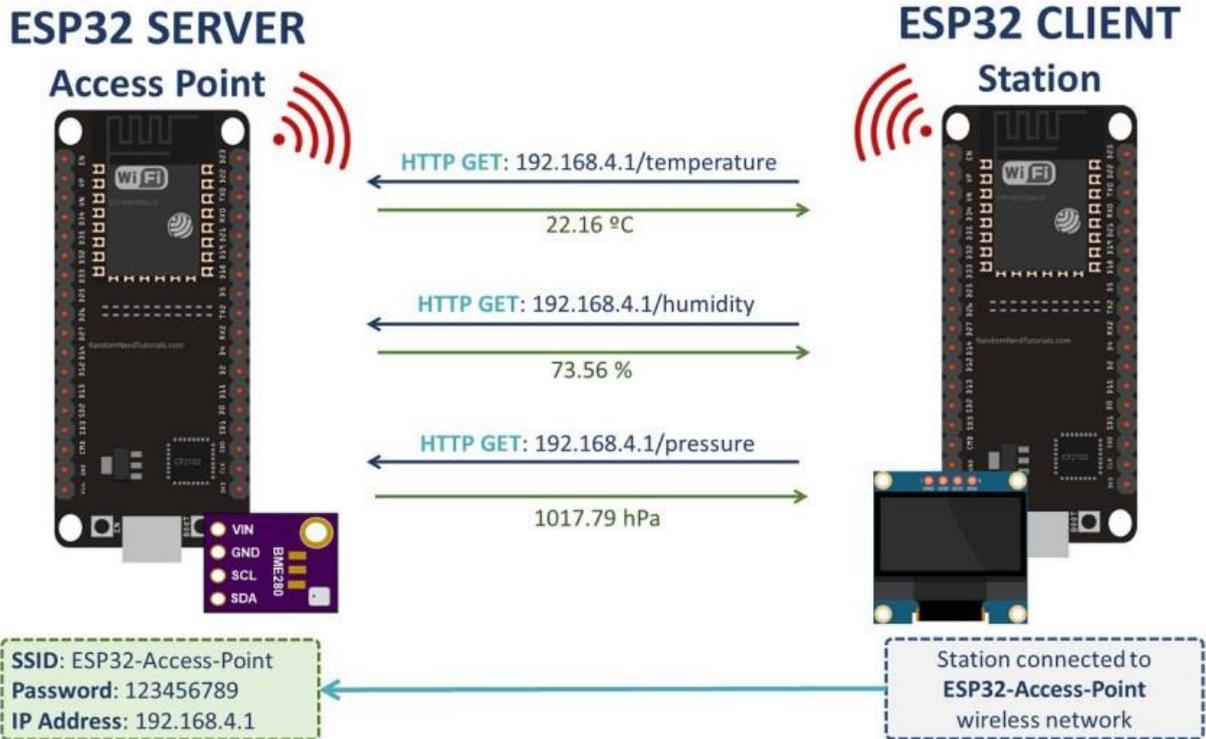
This extra Unit shows how to setup an HTTP communication between two ESP32 boards to exchange data via Wi-Fi without an internet connection (router). In simple words, you'll learn how to send data from one board to the other using HTTP requests.

Project Overview

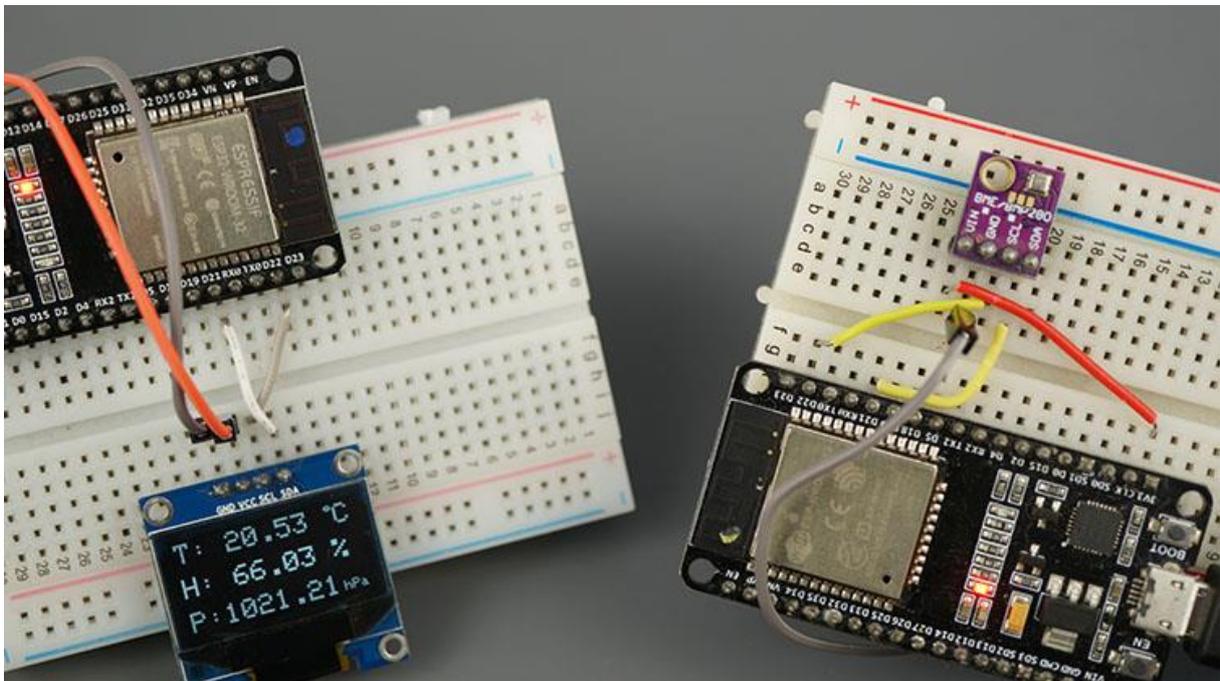
One ESP32 board will act as a server and the other ESP32 board will act as a client. Here's an overview on how things work:

- The ESP32 server creates its own wireless network (ESP32 Soft-Access Point). So, other Wi-Fi devices can connect to that network (SSID: ESP32-Access-Point, Password: 123456789).
- The ESP32 client is set as a station. So, it can connect to the ESP32 server wireless network.
- The client can make HTTP GET requests to the server to request sensor data or any other information. It just needs to use the IP address of the server to make a request on a certain route: */temperature*, */humidity* or */pressure*.
- The server listens for incoming requests and sends an appropriate response with the readings.
- The client receives the readings and displays them on the OLED display.
- As an example, the ESP32 client requests temperature, humidity and pressure to the server by making requests on the server IP address followed by */temperature*, */humidity* and */pressure*, respectively.

- The ESP32 server is listening on those routes and when a request is made, it sends the corresponding sensor readings via HTTP response.



Parts Required



For this tutorial, you need the following parts:

- [2x ESP32 Development boards](#)
- [BME280 sensor](#)
- [I2C SSD1306 OLED display](#)
- [Jumper Wires](#)
- [Breadboard](#)

Installing Libraries

For this tutorial you need to install the following libraries:

Asynchronous Web Server Libraries

We'll use the following libraries to handle HTTP requests (if you've followed previous units, you should already have these libraries installed):

- [ESPAsyncWebServer](#) library ([download ESPAsyncWebServer library](#))
- [Async TCP](#) library ([download AsyncTCP library](#))

These libraries are not available to install through the Library Manager. So, you need to unzip the libraries and move them to the Arduino IDE installation libraries folder.

Alternatively, you can go to **Sketch** ▶ **Include Library** ▶ **Add .ZIP library...** and select the libraries you've just downloaded.

BME280 Libraries

Install the following libraries to interface with the BME280 sensor. If you've followed previous units, you should have installed these libraries already. These libraries can be installed through the Arduino Library Manager. Go to **Sketch** ▶ **Include Library** ▶ **Manage Libraries** and search for the library name.

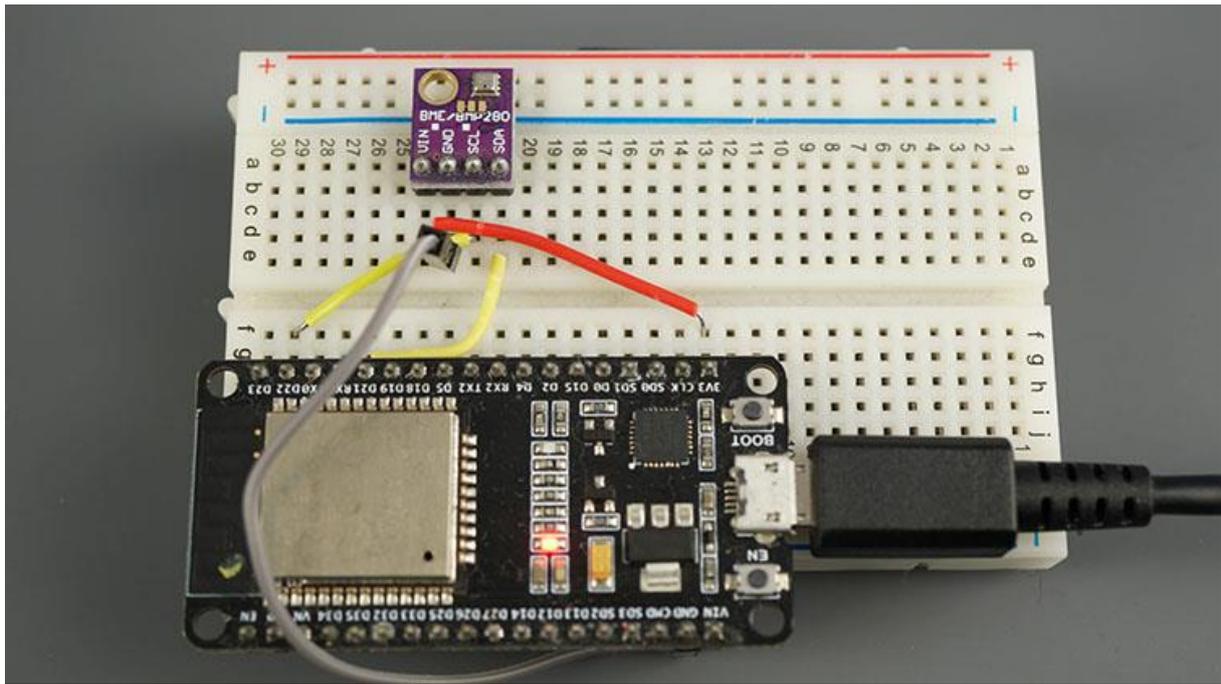
- [Adafruit BME280 library](#)
- [Adafruit unified sensor library](#)

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)

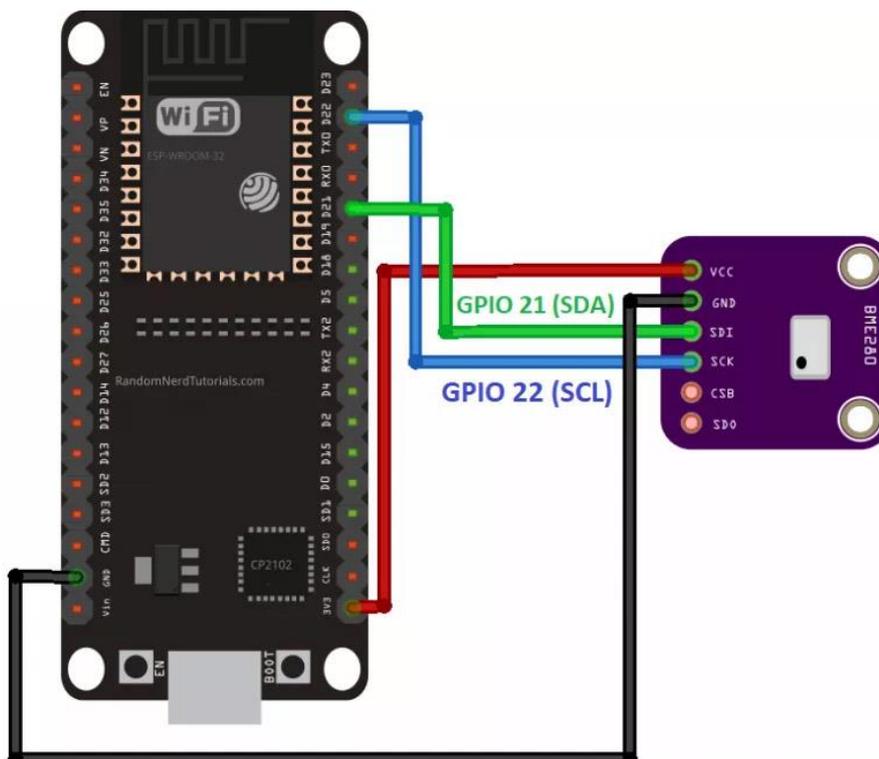


The ESP32 server is an Access Point (AP), that listens for requests on the `/temperature`, `/humidity` and `/pressure` URLs. When it gets requests on those URLs, it sends the latest BME280 sensor readings.

For demonstration purposes, we're using a BME280 sensor, but you can use any other sensor by modifying a few lines of code.

Schematic Diagram

Wire the ESP32 to the BME280 sensor as shown in the following schematic diagram.



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

Arduino Sketch for #1 ESP32 Server

Upload the following code to your board.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_Wi_Fi_Client_Server/Wi_Fi_Server/Wi_Fi_Server.ino

```
// Import required libraries
#include "WiFi.h"
#include "ESPAsyncWebServer.h"

#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

// Set your access point network credentials
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";

/*#include <SPI.h>
#define BME_SCK 18
#define BME_MISO 19
#define BME_MOSI 23
#define BME_CS 5*/

Adafruit_BME280 bme; // I2C
//Adafruit_BME280 bme(BME_CS); // hardware SPI
//Adafruit_BME280 bme(BME_CS, BME_MOSI, BME_MISO, BME_SCK); // software SPI

// Create AsyncWebServer object on port 80
AsyncWebServer 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 port for debugging purposes
    Serial.begin(115200);
    Serial.println();

    // Setting the ESP as an access point
    Serial.print("Setting AP (Access Point)...");
    // 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);

    server.on("/temperature", HTTP_GET, [] (AsyncWebServerRequest
*request) {
        request->send_P(200, "text/plain", readTemp().c_str());
    });
    server.on("/humidity", HTTP_GET, [] (AsyncWebServerRequest
*request) {
        request->send_P(200, "text/plain", readHumi().c_str());
    });
    server.on("/pressure", HTTP_GET, [] (AsyncWebServerRequest
*request) {
        request->send_P(200, "text/plain", readPres().c_str());
    });

    bool status;

    // default settings
    // (you can also pass in a Wire library object like &Wire2)
    status = bme.begin(0x76);
    if (!status) {
        Serial.println("Could not find a valid BME280 sensor, check
wiring!");
        while (1);
    }

    // Start server
    server.begin();
}

void loop(){
}

```

How the code works

Start by including the necessary libraries. Include the `WiFi.h` library and the `ESPAsyncWebServer.h` library to handle incoming HTTP requests.

```
#include "WiFi.h"
#include "ESPAsyncWebServer.h"
```

Include the following libraries to interface with the BME280 sensor.

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
```

In the following variables, define your access point network credentials:

```
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";
```

We're setting the SSID to `ESP32-Access-Point`, but you can give it any other name. You can also change the password. By default, its set to `123456789`.

Create an instance for the BME280 sensor called `bme`.

```
Adafruit_BME280 bme;
```

Create an asynchronous web server on port 80.

```
AsyncWebServer server(80);
```

Then, create three functions that return the temperature, humidity, and pressure as String variables.

```
String readTemp() {
    return String(bme.readTemperature());
    //return String(1.8 * bme.readTemperature() + 32);
}

String readHumi() {
    return String(bme.readHumidity());
}

String readPres() {
    return String(bme.readPressure() / 100.0F);
}
```

In the `setup()`, initialize the Serial Monitor for demonstration purposes.

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

Set your ESP32 as an access point with the SSID name and password defined earlier.

```
WiFi.softAP(ssid, password);
```

Then, handle the routes where the ESP32 will be listening for incoming requests.

For example, when the ESP32 server receives a request on the `/temperature` URL, it sends the temperature returned by the `readTemp()` function as a char (that's why we use the `c_str()` method).

```
server.on("/temperature", HTTP_GET, [] (AsyncWebServerRequest *request) {
    request->send_P(200, "text/plain", readTemp().c_str());
});
```

The same happens when the ESP receives a request on the `/humidity` and `/pressure` URLs.

```
server.on("/humidity", HTTP_GET, [] (AsyncWebServerRequest *request) {
  request->send_P(200, "text/plain", readHumi().c_str());
});
server.on("/pressure", HTTP_GET, [] (AsyncWebServerRequest *request) {
  request->send_P(200, "text/plain", readPres().c_str());
});
```

The following lines initialize the BME280 sensor.

```
bool status;

// default settings
// (you can also pass in a Wire library object like &Wire2)
status = bme.begin(0x76);
if (!status) {
  Serial.println("Could not find a valid BME280 sensor, check
wiring!");
  while (1);
}
```

Finally, start the server.

```
server.begin();
```

Because this is an asynchronous web server, there's nothing in the `loop()`.

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

Testing the ESP32 Server

Upload the code to your board and open the Serial Monitor. You should get something as follows:



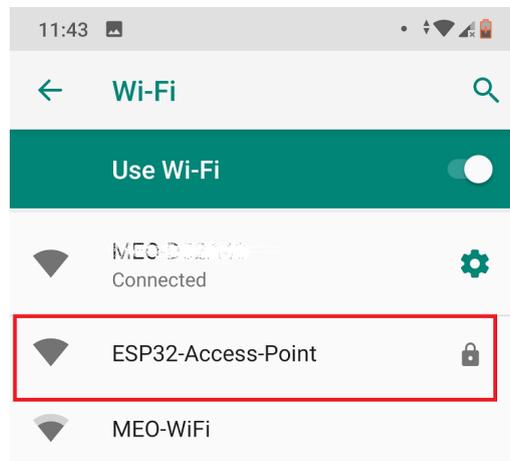
```
ets Jun 8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
config: 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:1044
load:0x40078000,len:8896
load:0x40080400,len:5816
entry 0x400806ac
Setting AP (Access Point)...AP IP address: 192.168.4.1
```

This means that the access point was set successfully.

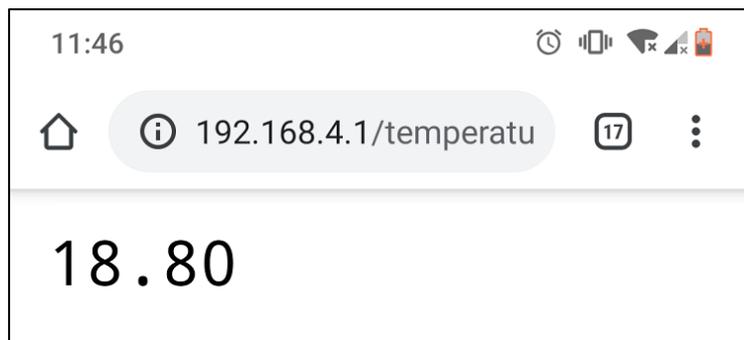
Now, to make sure it is listening for temperature, humidity and pressure requests, you need to connect to its network.

In your smartphone, go to the Wi-Fi settings and connect to the ESP32-Access-Point. The password is 123456789.

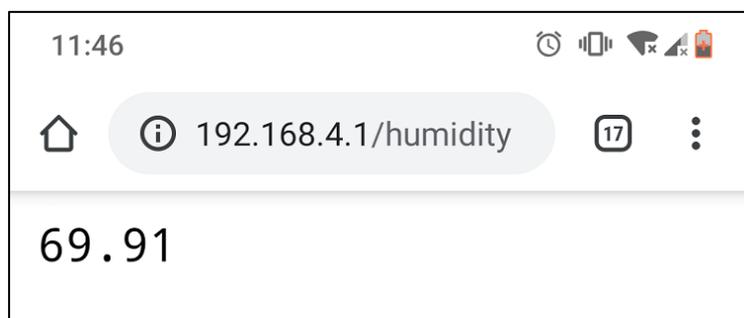


While connected to the access point, open your browser and type *192.168.4.1/temperature*.

You should get the temperature value in your browser:



Try this URL path for the humidity *192.168.4.1/humidity*:

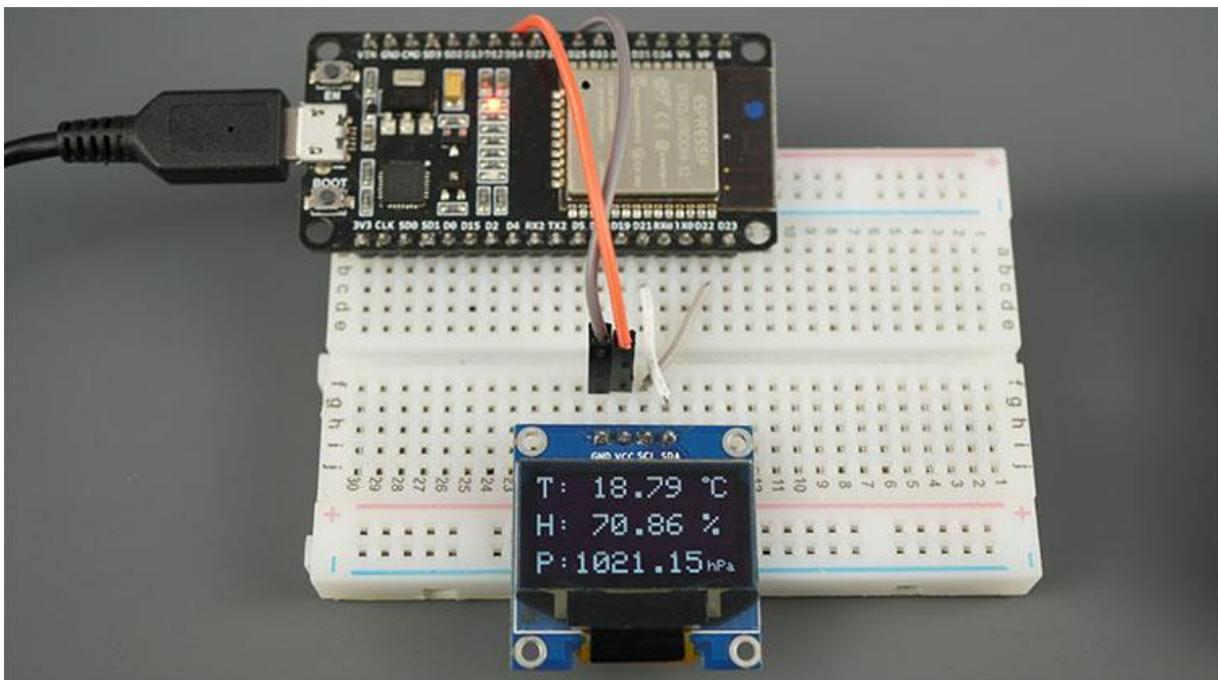


Finally, go to *192.168.4.1/pressure* URL:



If you're getting valid readings, it means that everything is working properly. Now, you need to prepare the other ESP32 board (client) to make those requests for you and display them on the OLED display.

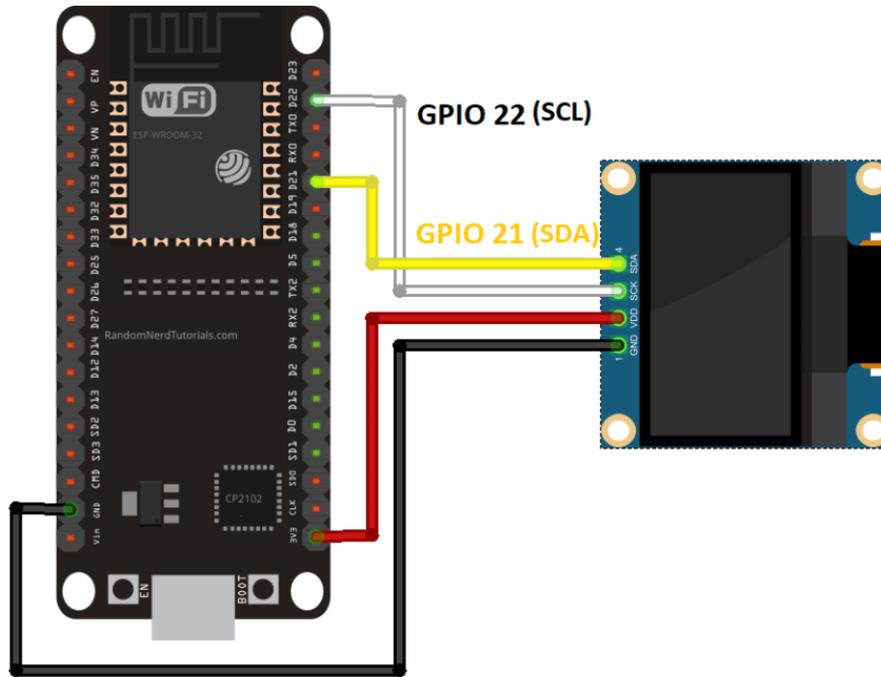
#2 ESP32 Client (Station)



The ESP32 Client is a Wi-Fi station that is connected to the ESP32 Server. The client requests the temperature, humidity and pressure from the server by making HTTP GET requests on the */temperature*, */humidity*, and */pressure* URL routes. Then, it displays the readings on an OLED display.

Schematic Diagram

Wire the ESP32 to the OLED display as shown in the following schematic diagram.



You can 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

Arduino Sketch for #2 ESP32 Client

Upload the following code to the other ESP32:

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_Wi_Fi_Client_Server/Wi_Fi_Client/Wi_Fi_Client.ino

```
#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 with IP Address: ");
  Serial.println(WiFi.localIP());
}

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:

```
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
  Serial.println(F("SSD1306 allocation failed"));
  for(;;); // Don't proceed, loop forever
}
```

```
}  
display.clearDisplay();  
display.setTextColor(WHITE);
```

Note: if your OLED display is not working, check its I2C address using an [I2C scanner sketch](#) and change the code accordingly.

Connect the ESP32 client to the ESP32 server network.

```
WiFi.begin(ssid, password);  
Serial.println("Connecting");  
while(WiFi.status() != WL_CONNECTED) {  
  delay(500);  
  Serial.print(".");  
}  
Serial.println("");  
Serial.print("Connected to WiFi network with IP Address: ");  
Serial.println(WiFi.localIP());
```

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 in the OLED display:

```
display.setTextSize(2);  
display.setTextColor(WHITE);  
display.setCursor(0,0);  
display.print("T: ");  
display.print(temperature);  
display.print(" ");  
display.setTextSize(1);  
display.cp437(true);  
display.write(248);  
display.setTextSize(2);  
display.print("C");  
  
// 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 x number of 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.

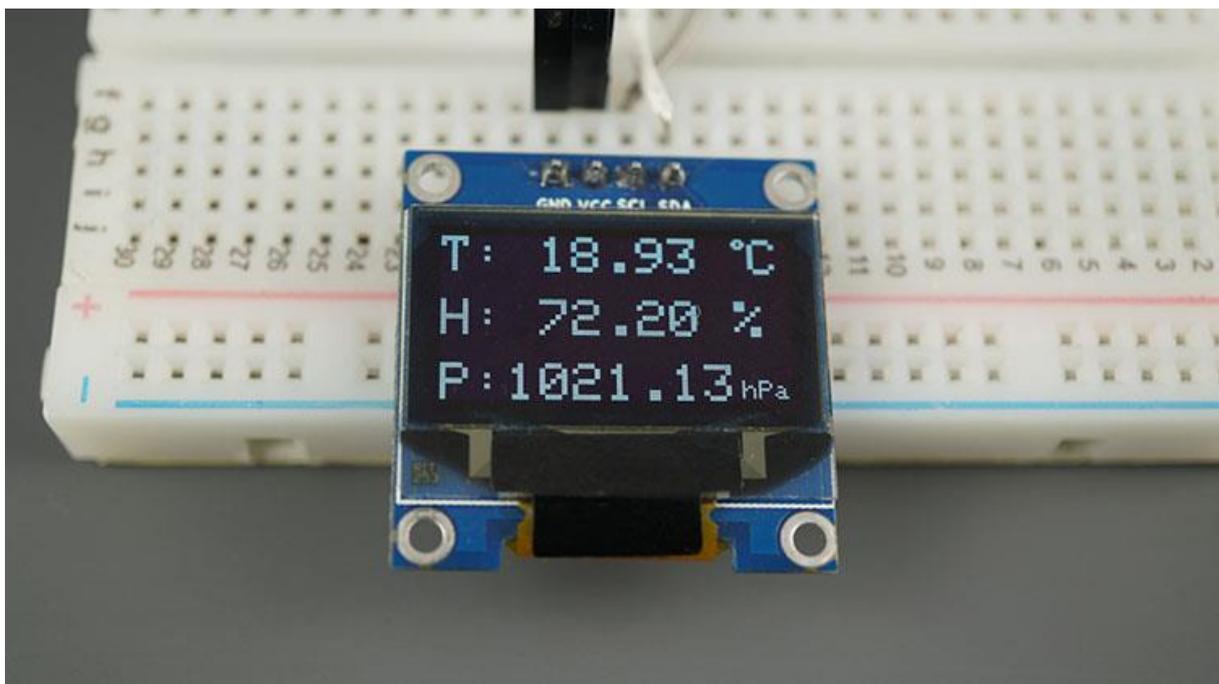
Testing the ESP32 Client

Having both boards fairly close and powered, you'll see that ESP #2 is receiving new temperature, humidity and pressure readings every 5 seconds from ESP #1.

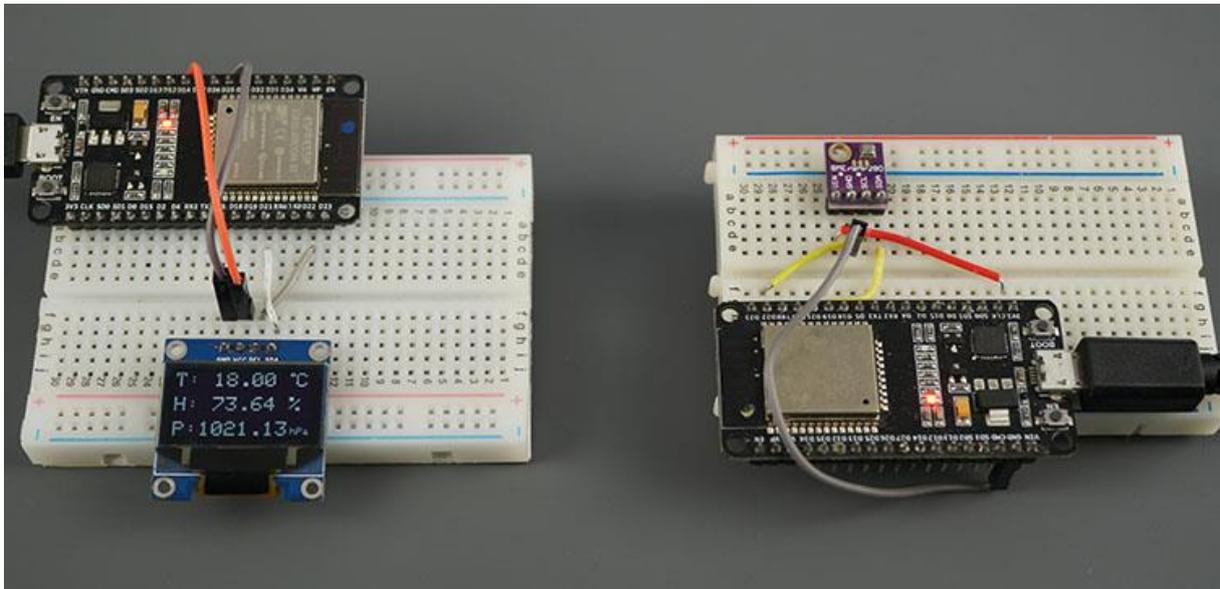
This is what you should see on the ESP32 Client Serial Monitor.

```
COM3
Send
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
Temperature: 18.89 *C - Humidity: 74.41 % - Pressure: 1021.20 hPa
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
Temperature: 18.92 *C - Humidity: 74.41 % - Pressure: 1021.18 hPa
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
Temperature: 18.96 *C - Humidity: 75.60 % - Pressure: 1021.20 hPa
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
Temperature: 18.99 *C - Humidity: 75.39 % - Pressure: 1021.17 hPa
 Autoscroll  Show timestamp
Both NL & CR v 115200 baud v Clear output
```

The sensor readings are also displayed in the OLED.



That's it! Your two boards are talking with each other.

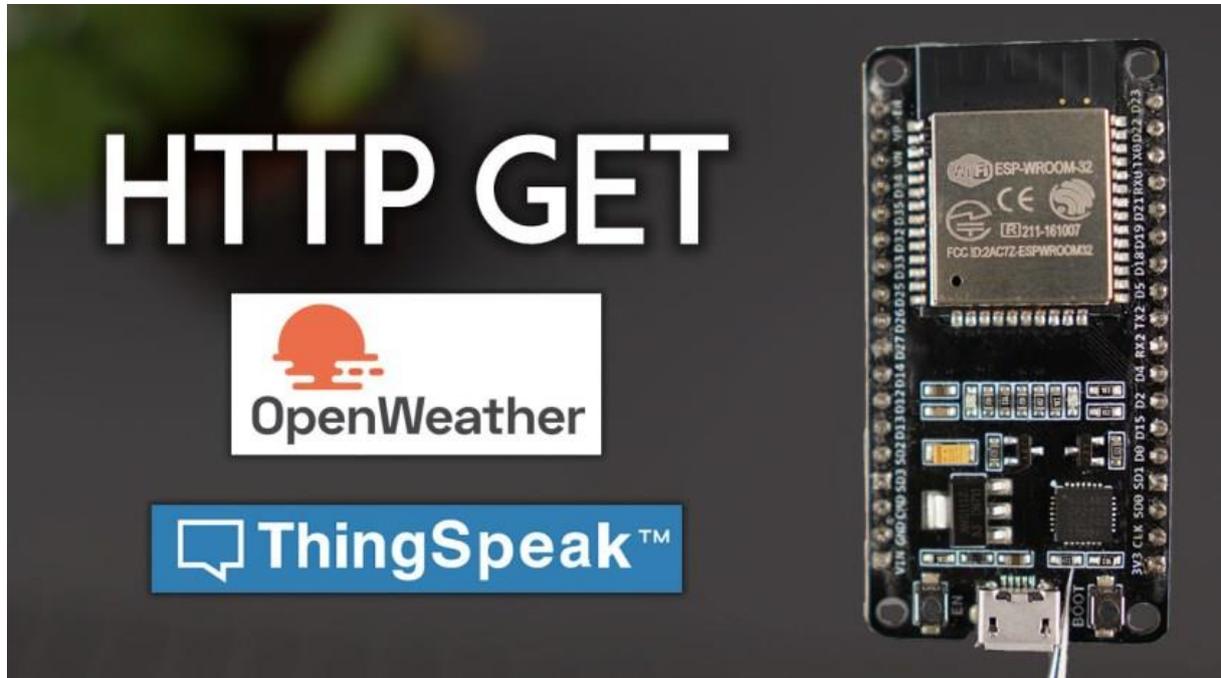


Wrapping Up

In this Unit you've learned how to send data from one ESP32 to another ESP32 board via Wi-Fi using HTTP requests without the need to connect to the internet. For demonstration purposes, we've shown how to send BME280 sensor readings, but you can use any other sensor or send any other data.

ESP32 HTTP GET

(OpenWeatherMap and ThingSpeak)



In this Unit, you'll learn how to make HTTP GET requests using the ESP32 board with Arduino IDE. We'll demonstrate how to decode JSON data from OpenWeatherMap.org and plot values in charts using ThingSpeak.

HTTP GET Request Method

The Hypertext Transfer Protocol (HTTP) works as a request-response protocol between a client and server. Here's an example:

- The ESP32 (client) submits an HTTP request to a Server (for example: OpenWeatherMap.org or ThingSpeak);
- The server returns a response to the ESP32 (client);
- Finally, the response contains status information about the request and may also contain the requested content.

HTTP GET

GET is used to request data from a specified resource. It is often used to get values from APIs.

For example, you can use a simple request to return a value or JSON object:

```
GET /weather?countryCode=PT
```

Additionally, you can also make a GET request to update a value (like with ThingSpeak). For example, you can use:

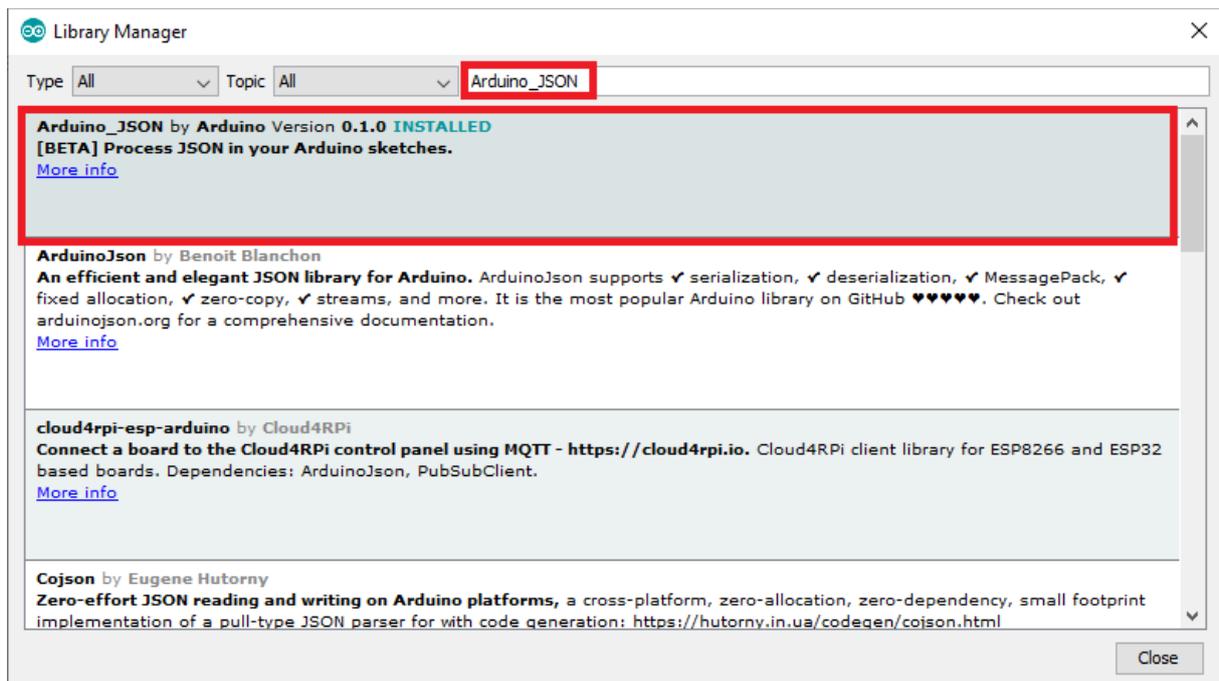
```
GET /update?field1=value1
```

Note that the query string (name = field1 and value = value1) is sent in the URL of the HTTP GET request.

(With HTTP GET, data is visible to everyone in the URL request.)

Arduino_JSON Library

To follow this tutorial, 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:



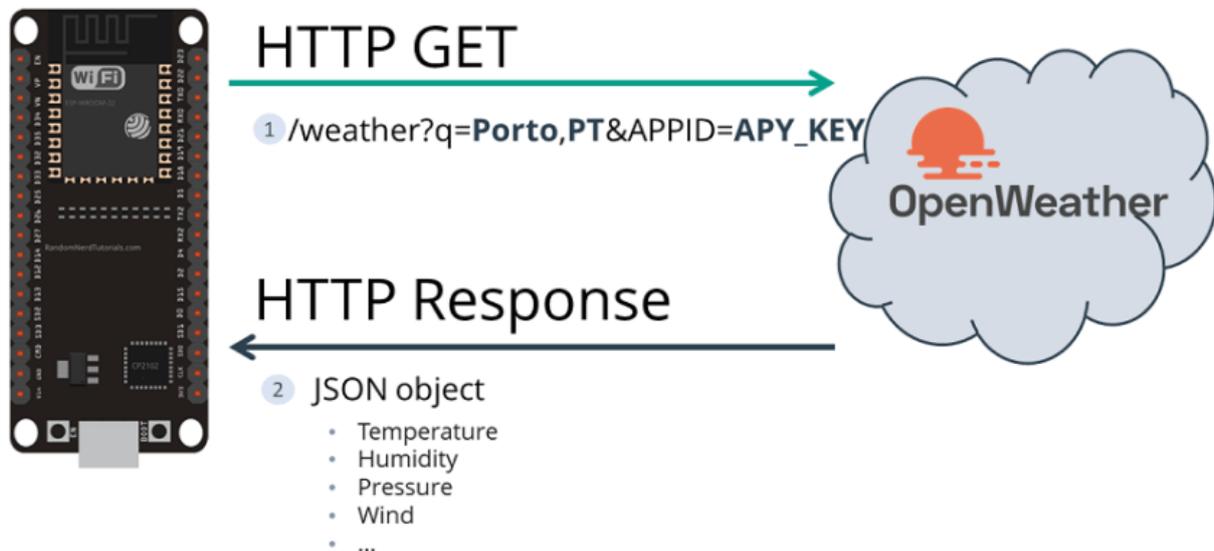
Other Web Services or APIs

In this guide, you'll learn how to setup your ESP32 board to perform HTTP requests to OpenWeatherMap.org and ThingSpeak. All examples presented in this guide also work with other APIs.

In summary, to make this guide compatible with any service, you need to search for the service API documentation. Then, you need the server name (URL or IP address), and parameters to send in the request (URL path or request body). Finally, modify our examples to integrate with any API you want to use.

1. ESP32 HTTP GET: JSON Data (OpenWeatherMap.org)

In this example you'll learn how to make API requests to access data. As an example, we'll use the OpenWeatherMap API. This API has a free plan and provides lots of useful information about the weather in almost any location in the world.



Using OpenWeatherMap API

An application programming interface (API) is a set of functions written by software developers to enable anyone to use their data or services. The [OpenWeatherMap](#) project has an API that enables users to request weather data.

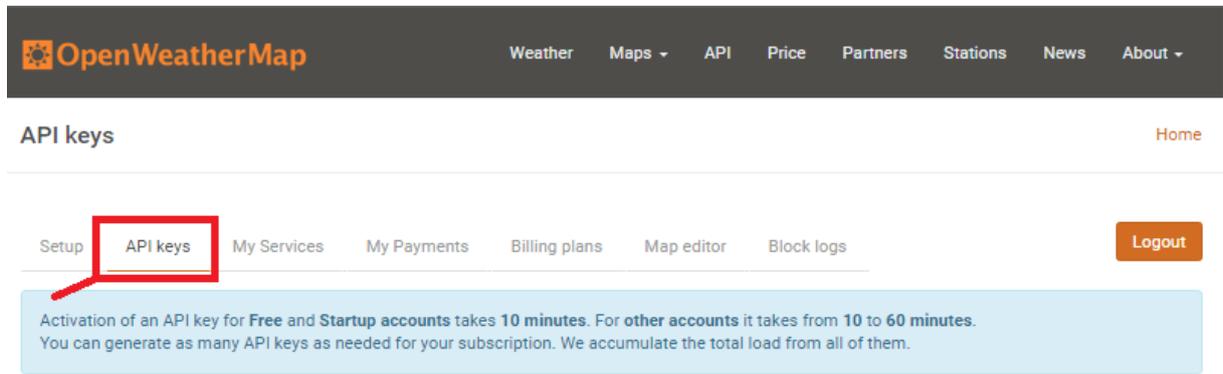


In this project, you'll use that API to request the day's weather forecast for your chosen location. Learning to use APIs is a great skill because it allows you access to a wide variety of constantly changing information, such as current stock prices, currency exchange rates, the latest news, traffic updates, tweets, and much more.

Note: API keys are unique to the user and shouldn't be shared with anyone.

OpenWeatherMap's free plan provides everything you need to complete this project. To use the API you need an API key, known as the APIID. To get the APIID:

- 1) Open a browser and go to <https://openweathermap.org/appid/>
- 2) Press the **Sign up** button and create a free account.
- 3) Go to this link: https://home.openweathermap.org/api_keys and get your API key.



- 4) On the API keys tab, you'll see a default key (highlighted in a red rectangle in figure above); this is a unique key you'll need to pull information from the site. Copy and paste this key somewhere; you'll need it in a moment.

To pull information on weather in your chosen location, enter the following URL:

```
http://api.openweathermap.org/data/2.5/weather?q=yourCityName,yourCountryCode&APPID=yourUniqueAPIkey
```

Replace **yourCityName** with the city you want data for, **yourCountryCode** with the country code for that city, and **yourUniqueAPIkey** with the unique API key from step 4. For example, the updated API URL for the city of Porto, Portugal, would be:

```
http://api.openweathermap.org/data/2.5/weather?q=Porto,PT&APPID=801d2603e9f2e1c70e042e4f5f6e0---
```

Copy your URL into your browser, and the API will return a bunch of information corresponding to your local weather. We got the following information about the weather in Porto, Portugal, on the day we wrote this tutorial.

```
{ "coord": { "lon": -8.61, "lat": 41.15 }, "weather": [ { "id": 801, "main": "Clouds", "description": "few clouds", "icon": "02d" } ], "base": "stations", "main": { "temp": 294.44, "feels_like": 292.82, "temp_min": 292.15, "temp_max": 297.04, "pressure": 1008, "humidity": 63, "visibility": 10000, "wind": { "speed": 4.1, "deg": 240 }, "clouds": { "all": 20 }, "dt": 1589288330, "sys": { "type": 1, "id": 6900, "country": "PT", "sunrise": 1589260737, "sunset": 1589312564 }, "timezone": 3600, "id": 2735943, "name": "Porto", "cod": 200 }
```

Next, you'll see how to use this information to get specific data like temperature, humidity, pressure, wind speed, etc.

Code ESP32 HTTP GET OpenWeatherMap.org

Copy the following code to your Arduino IDE, but don't upload it yet. You need to make some changes to make it work for you.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_HTTP_GET_POST/ESP32_HTTP_GET_OpenWeatherMap/ESP32_HTTP_GET_OpenWeatherMap.ino

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

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

// Your Domain name with URL path or IP address with path
String openWeatherMapApiKey = "REPLACE_WITH_YOUR_OPEN_WEATHER_MAP_API_KEY";
// Example:
//String openWeatherMapApiKey = "bd939aa3d23ff33d3c8f5dd1dd435";

// Replace with your country code and city
String city = "Porto";
String countryCode = "PT";

// 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;

String jsonBuffer;

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.");
}

void loop() {
  // Send an HTTP GET request
  if ((millis() - lastTime) > timerDelay) {
    // Check WiFi connection status
    if(WiFi.status() == WL_CONNECTED){
```

```

    String serverPath =
"http://api.openweathermap.org/data/2.5/weather?q=" + city + "," +
countryCode + "&APPID=" + openWeatherMapApiKey;

    jsonBuffer = httpGETRequest(serverPath.c_str());
    Serial.println(jsonBuffer);
    JSONVar myObject = JSON.parse(jsonBuffer);

    // JSON.typeof(jsonVar) can be used to get the type of the var
    if (JSON.typeof(myObject) == "undefined") {
        Serial.println("Parsing input failed!");
        return;
    }

    Serial.print("JSON object = ");
    Serial.println(myObject);
    Serial.print("Temperature: ");
    Serial.println(myObject["main"]["temp"]);
    Serial.print("Pressure: ");
    Serial.println(myObject["main"]["pressure"]);
    Serial.print("Humidity: ");
    Serial.println(myObject["main"]["humidity"]);
    Serial.print("Wind Speed: ");
    Serial.println(myObject["wind"]["speed"]);
}
else {
    Serial.println("WiFi Disconnected");
}
lastTime = millis();
}
}

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;
}

```

Setting your network credentials

Modify the next lines with your network credentials: SSID and password. The code is well commented on where you should make the changes.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

Setting your OpenWeatherMap.org API Key

Insert your API key in the following like:

```
String openWeatherMapApiKey = "REPLACE_WITH_YOUR_OPEN_WEATHER_MAP_API_KEY";
```

Setting your city and country

Enter the city you want to get data for, as well as the country code in the following variables:

```
String city = "Porto";  
String countryCode = "PT";
```

After making these changes, you can upload the code to your board. Continue reading to learn how the code works.

HTTP GET Request (JSON Object)

In the `loop()`, call the `httpGETRequest()` function to make the HTTP GET request:

```
String serverPath = "http://api.openweathermap.org/data/2.5/weather?q=" +  
city + "," + countryCode + "&APPID=" + openWeatherMapApiKey;  
jsonBuffer = httpGETRequest(serverPath.c_str());
```

The `httpGETRequest()` function makes a request to OpenWeatherMap and it retrieves a string with a JSON object that contains all the information about the weather for your city.

```
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;  
}
```

Decoding JSON Object

To get access to the values, decode the JSON object and store all values in the `jsonBuffer` array.

```

if (JSON.typeof(myObject) == "undefined") {
  Serial.println("Parsing input failed!");
  return;
}
Serial.print("JSON object = ");
Serial.println(myObject);
Serial.print("Temperature: ");
Serial.println(myObject["main"]["temp"]);
Serial.print("Pressure: ");
Serial.println(myObject["main"]["pressure"]);
Serial.print("Humidity: ");
Serial.println(myObject["main"]["humidity"]);
Serial.print("Wind Speed: ");
Serial.println(myObject["wind"]["speed"]);

```

HTTP GET Demonstration

After uploading the code, open the Serial Monitor and you'll see that it's receiving the following JSON data:

```

{"coord":{"lon":-8.61,"lat":41.15},"weather":[{"id":801,"main":"Clou
ds","description":"few clouds","icon":"02d"}],"base":"stations","mai
n":{"temp":294.44,"feels_like":292.82,"temp_min":292.15,"temp_max":2
97.04,"pressure":1008,"humidity":63},"visibility":10000,"wind":{"spe
ed":4.1,"deg":240},"clouds":{"all":20},"dt":1589288330,"sys":{"type"
:1,"id":6900,"country":"PT","sunrise":1589260737,"sunset":1589312564
},"timezone":3600,"id":2735943,"name":"Porto","cod":200}

```

Then, it prints the decoded JSON object in the Arduino IDE Serial Monitor to get the temperature (in Kelvin), pressure, humidity and wind speed values.

The screenshot shows the Serial Monitor window for COM3. The output text is as follows:

```

Connecting
.
Connected to WiFi network with IP Address: 192.168.1.75
Timer set to 10 seconds (timerDelay variable), it will take 10
HTTP Response code: 200
{"coord":{"lon":-8.61,"lat":41.15},"weather":[{"id":500,"main":
JSON object = {"coord":{"lon":-8.61,"lat":41.15},"weather":[{"i
Temperature: 292.41
Pressure: 1007
Humidity: 72
Wind Speed: 6.2
HTTP Response code: 200
{"coord":{"lon":-8.61,"lat":41.15},"weather":[{"id":500,"main":
JSON object = {"coord":{"lon":-8.61,"lat":41.15},"weather":[{"i
Temperature: 292.41
Pressure: 1007
Humidity: 72
Wind Speed: 6.2

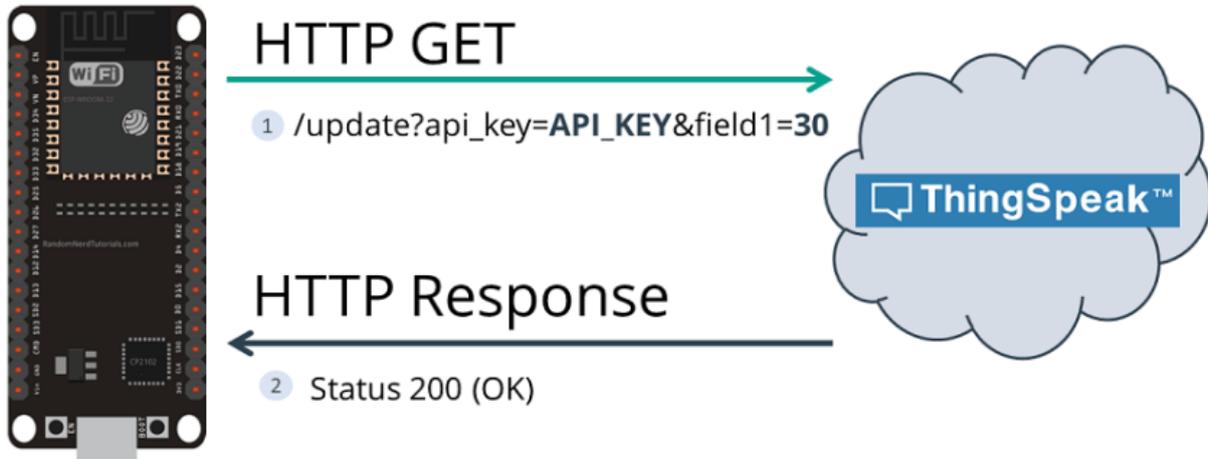
```

At the bottom of the window, there are checkboxes for 'Autoscroll' (checked) and 'Show timestamp' (unchecked). On the right, there are dropdown menus for 'Newline' and '115200 baud', and a 'Clear output' button.

For demonstration purposes, we're requesting new data every 10 seconds. However, for a long term project you should increase the timer or check the API call limits per hour/minute to avoid getting blocked/banned.

2. ESP32 HTTP GET: Update Value (ThingSpeak)

In this example, the ESP32 makes an HTTP GET request to update a reading in ThingSpeak.

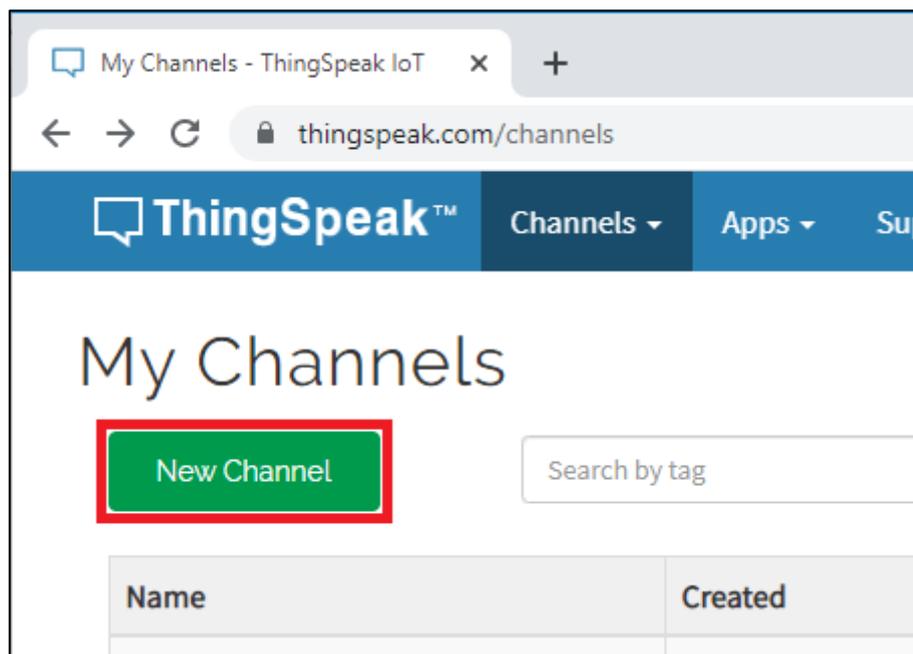


Using ThingSpeak API

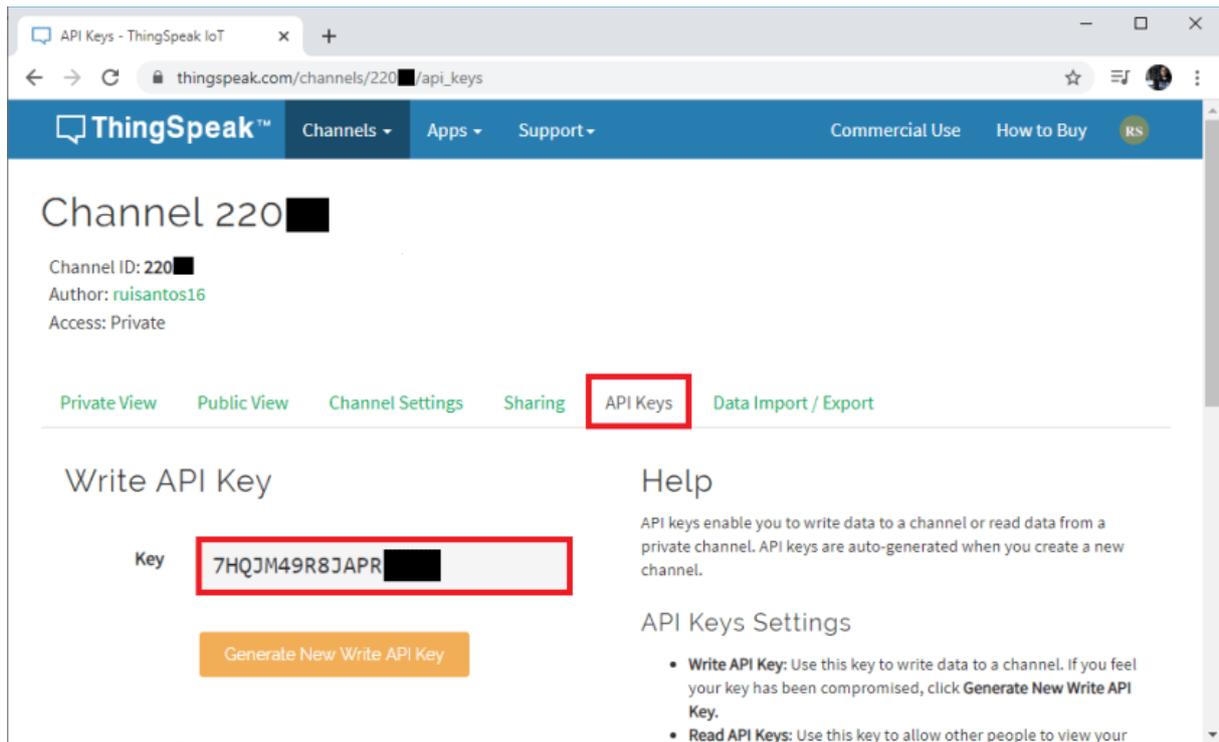
ThingSpeak has a free API that allows you to store and retrieve data using HTTP. In this tutorial, you'll use the ThingSpeak API to publish and visualize data in charts from anywhere.

To use ThingSpeak with your ESP, you need an API key. Follow the next steps:

- 1) Go to [ThingSpeak.com](https://thingspeak.com) and create a free account.
- 2) Then, open the [Channels](#) tab.
- 3) Create a New Channel.



4) Open your newly created channel and select the API Keys tab to copy your API Key.



Code ESP32 HTTP GET ThingSpeak

Copy the next sketch to your Arduino IDE (type your SSID, password, and API Key):

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_HTTP_GET_POST/ESP32_HTTP_GET_ThingSpeak/ESP32_HTTP_GET_ThingSpeak.ino

```
#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=7HQM49R8JAPR";

// 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) {
            HTTPClient http;

            String serverPath = serverName + "&field1=" +
String(random(40));

            // Your Domain name with URL path or IP address with path
            http.begin(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. The code is well commented on where you should make the changes.

```
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.

```
String serverName = "http://api.thingspeak.com/update?api_key=REPLAC
E_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.

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(serverPath.c_str());
// Send HTTP GET request
int httpResponseCode = http.GET();
```

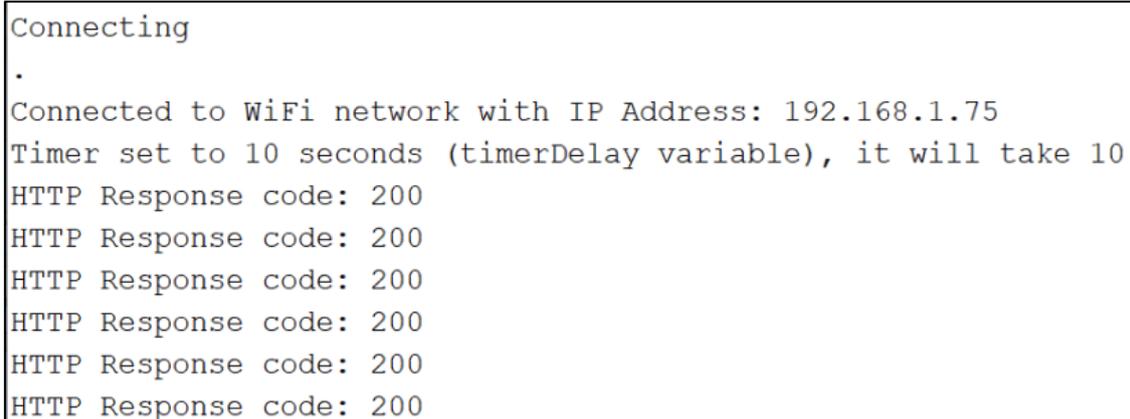
The ESP32 makes a new request in the following URL to update the sensor `field1` with a new value (30).

```
http://api.thingspeak.com/update?api_key=REPLACE_WITH_YOUR_API_KEY&f
ield1=30
```

Then, the following lines of code save the HTTP response from the server.

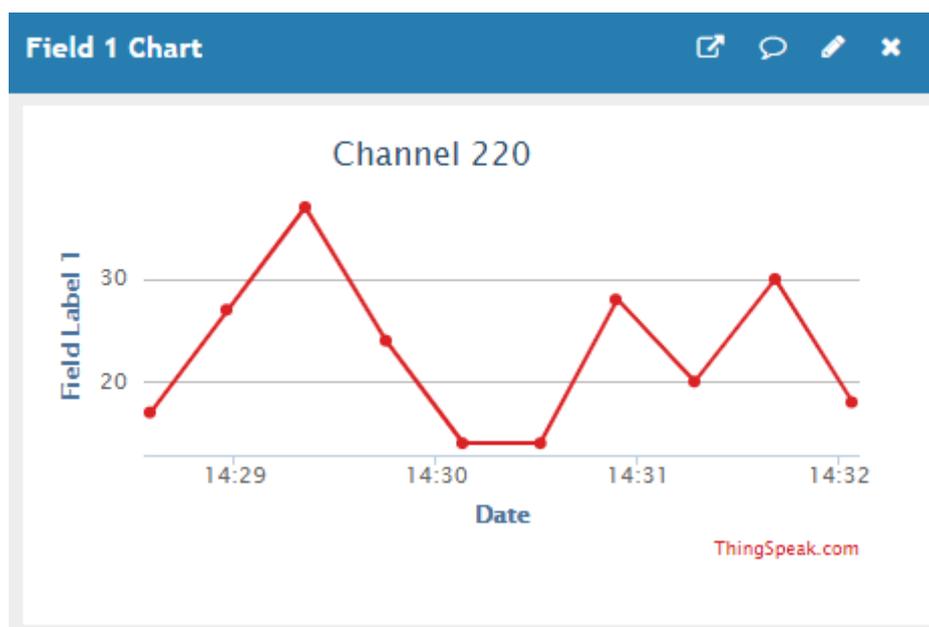
```
if (httpResponseCode>0) {
    Serial.print("HTTP Response code: ");
    Serial.println(httpResponseCode);
    String payload = http.getString();
    Serial.println(payload);
}
else {
    Serial.print("Error code: ");
    Serial.println(httpResponseCode);
}
```

In the Arduino IDE serial monitor, you should see an HTTP response code of 200 (this means that the request has succeeded).



```
Connecting
.
Connected to WiFi network with IP Address: 192.168.1.75
Timer set to 10 seconds (timerDelay variable), it will take 10
HTTP Response code: 200
```

Your ThingSpeak Dashboard should be receiving new readings every 10 seconds.



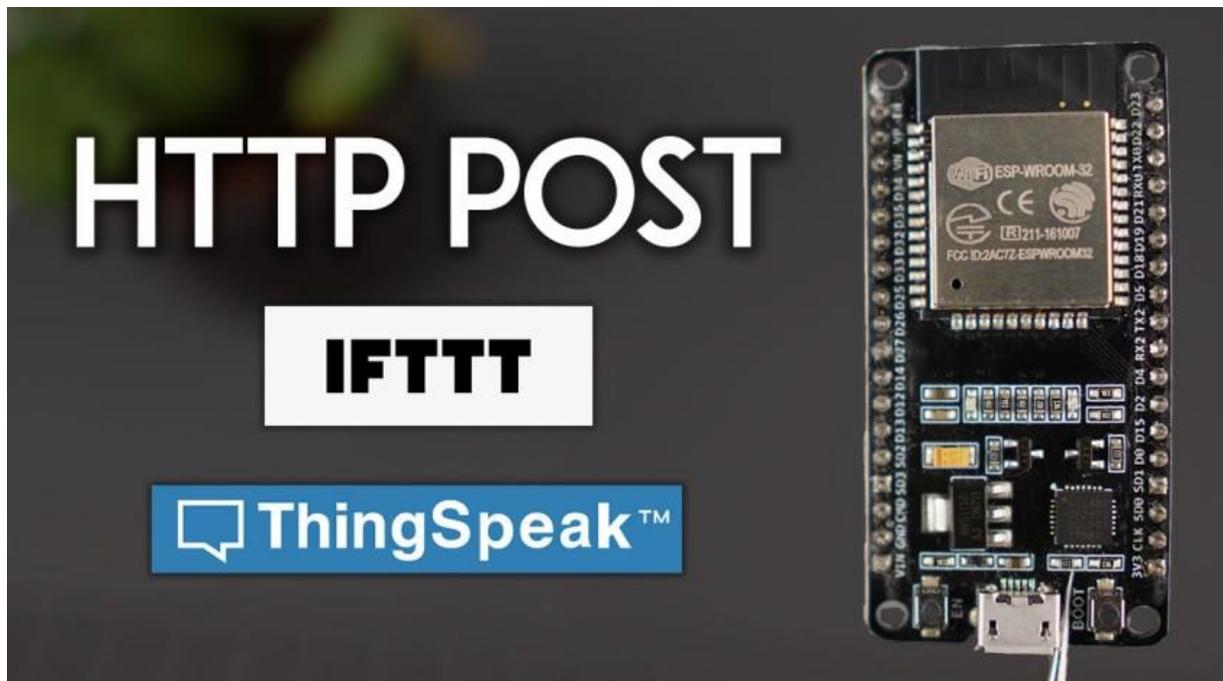
For a final application, you might need to increase the timer or check the API call limits per hour/minute to avoid getting blocked/banned.

Wrapping Up

In this tutorial you've learned how to integrate your ESP32 with web services using HTTP GET requests. You can also make HTTP POST requests with the ESP32 (see the next Unit).

ESP32 HTTP POST

(ThingSpeak and IFTTT.com)



In this Unit, you'll learn how to make HTTP POST requests using the ESP32 board with Arduino IDE. We'll demonstrate how to post JSON data or URL encoded values to two web APIs (ThingSpeak and IFTTT.com).

HTTP POST Request Method

The Hypertext Transfer Protocol (HTTP) works as a request-response protocol between a client and server. Here's an example:

- 1) The ESP32 (client) submits an HTTP request to a Server (for example: ThingSpeak or IFTTT.com);
- 2) The server returns a response to the ESP32 (client);
- 3) Finally, the response contains status information about the request and may also contain the requested content.

HTTP POST

POST is used to send data to a server to create/update a resource. For example, publish sensor readings to a server.

The data sent to the server with POST is stored in the request body of the HTTP request:

```
POST /update HTTP/1.1
Host: example.com
api_key=api&field1=value1
Content-Type: application/x-www-form-urlencoded
```

In the body request, you can also send a JSON object:

```
POST /update HTTP/1.1
Host: example.com
{api_key: "api", field1: value1}
Content-Type: application/json
```

(With HTTP POST, data is not visible in the URL request. However, if it's not encrypted, it's still visible in the request body.)

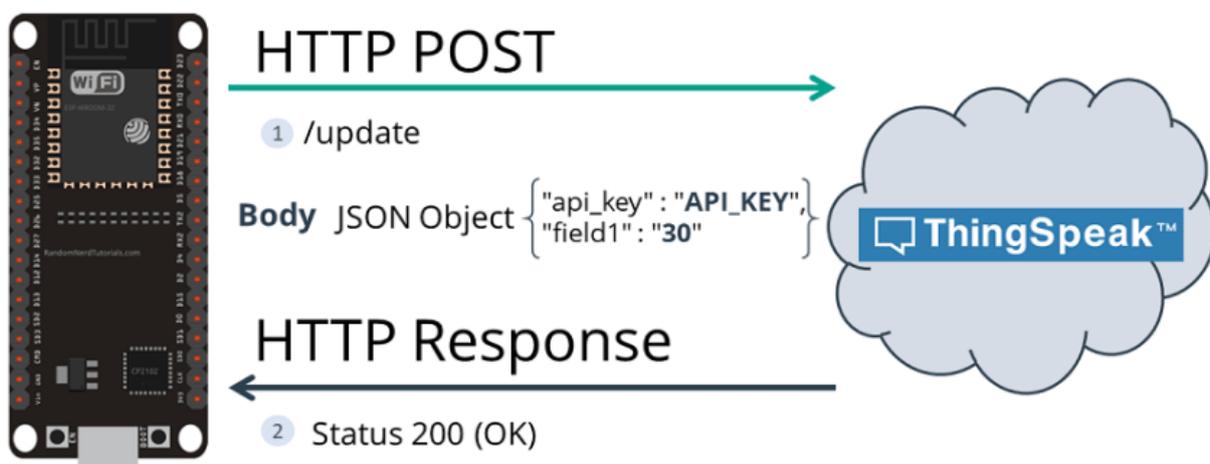
Other Web Services or APIs

In this guide, you'll learn how to setup your ESP32 board to perform HTTP requests to ThingSpeak and IFTTT.com. All examples presented in this guide also work with other APIs.

In summary, to make this guide compatible with any service, you need to search for the service API documentation. Then, you need the server name (URL or IP address), and parameters to send in the request (URL path or request body). Finally, modify our examples to integrate with any API you want to use.

1. ESP32 HTTP POST Data (ThingSpeak)

In this example, the ESP32 makes an HTTP POST request to send a new value to ThingSpeak.

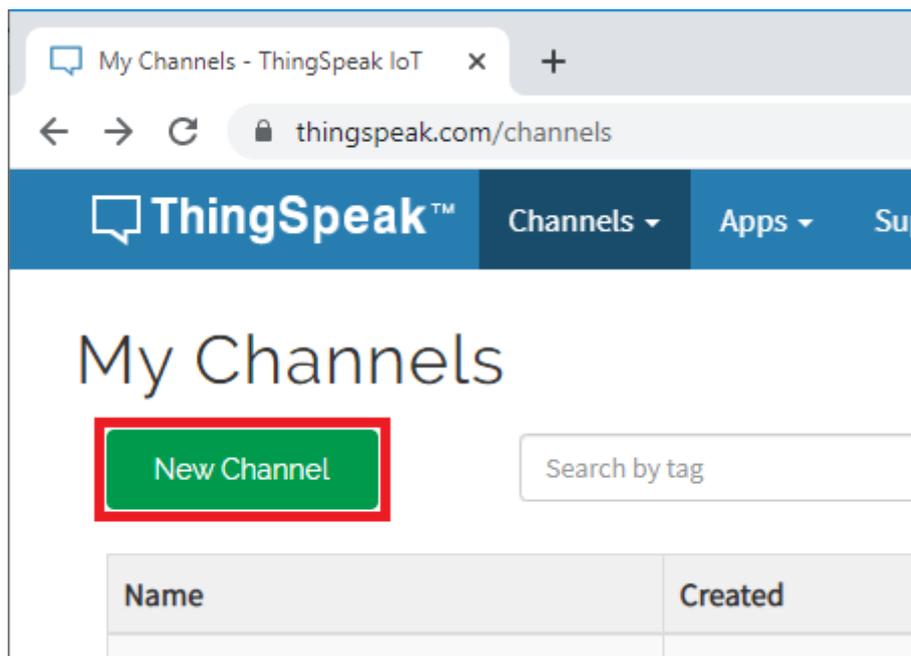


Using ThingSpeak API

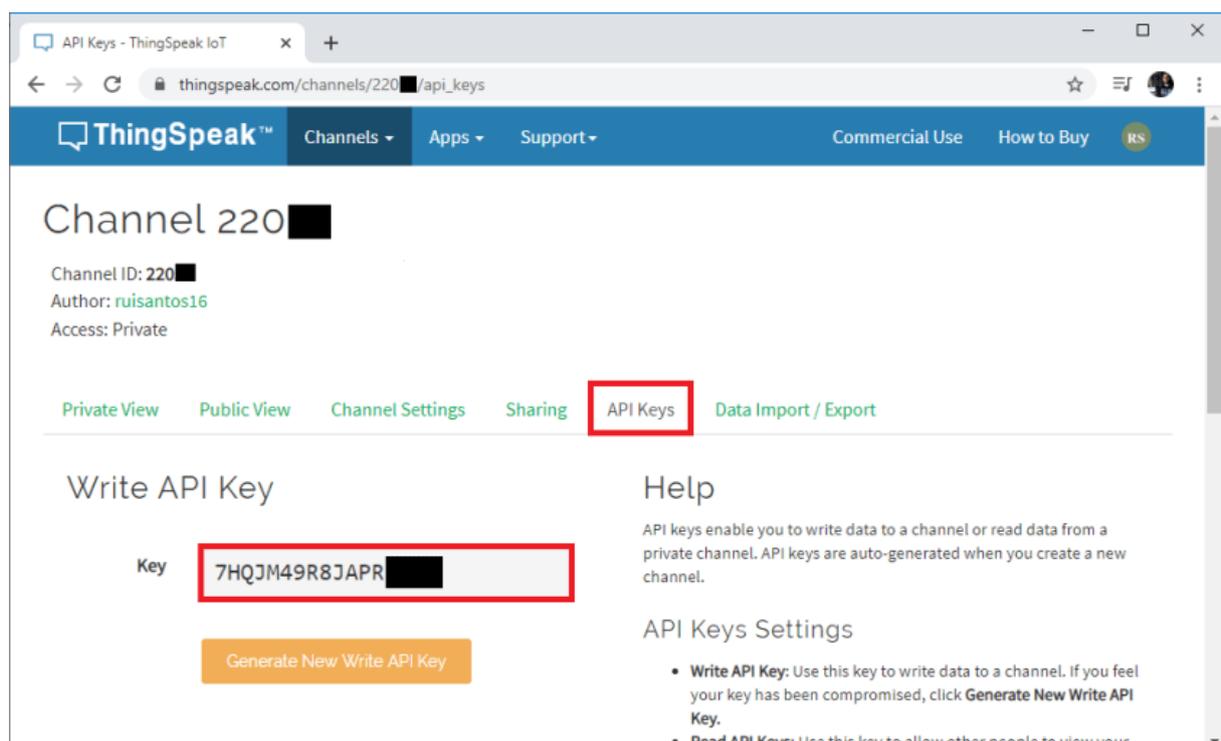
ThingSpeak has a free API that allows you to store and retrieve data using HTTP. In this tutorial, you'll use the ThingSpeak API to publish and visualize data in charts from anywhere. As an example, we'll publish random values, but in a real application you would use real sensor readings.

To use ThingSpeak API, you need an API key. Follow the next steps:

- 1) Go to [ThingSpeak.com](https://thingspeak.com) and create a free account.
- 2) Then, open the [Channels](#) tab.
- 3) Create a New Channel.



- 4) Open your newly created channel and select the API Keys tab to copy your API Key.



Code ESP32 HTTP POST ThingSpeak

Copy the next sketch to your Arduino IDE:

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_HTTP_GET_POST/ESP32_HTTP_POST_ThingSpeak/ESP32_HTTP_POST_ThingSpeak.ino

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

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

// Domain Name with full URL Path for HTTP POST Request
const char* serverName = "http://api.thingspeak.com/update";
// Service API Key
String apiKey = "REPLACE_WITH_YOUR_API_KEY";

// 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;
// Set timer to 10 minutes (600000)
//unsigned long timerDelay = 600000;
// Timer set 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 POST request every 10 seconds
  if ((millis() - lastTime) > timerDelay) {
    //Check WiFi connection status
    if(WiFi.status() == WL_CONNECTED) {
      HTTPClient http;

      // Your Domain name with URL path or IP address with path
```

```

    http.begin(serverName);

    // Specify content-type header
    http.addHeader("Content-Type", "application/x-www-form-
urlencoded");
    // Data to send with HTTP POST
    String httpRequestData = "api_key=" + apiKey + "&field1=" +
String(random(40));
    // Send HTTP POST request
    int httpResponseCode = http.POST(httpRequestData);

    /*
    // If you need an HTTP request with a content type:
application/json, use the following:
    http.addHeader("Content-Type", "application/json");
    // JSON data to send with HTTP POST
    String httpRequestData = "{\"api_key\": \"" + apiKey +
"\", \"field1\": \"" + String(random(40)) + "\"}";
    // Send HTTP POST request
    int httpResponseCode = http.POST(httpRequestData);*/

    Serial.print("HTTP Response 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 API Key

Modify the `apiKey` variable to include your ThingSpeak API key.

```

String apiKey = "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 POST request.

HTTP POST Request

In the `loop()` is where you make the HTTP POST request with URL encoded data every 10 seconds with random data:

```

// Specify content-type header
http.addHeader("Content-Type", "application/x-www-form-urlencoded");
// Data to send with HTTP POST

```

```
String httpRequestData = "api_key=" + apiKey + "&field1=" +
String(random(40));

// Send HTTP POST request
int httpResponseCode = http.POST(httpRequestData);
```

For example, the ESP32 makes a URL encoded request to publish a new value (30) to field1.

```
POST /update HTTP/1.1
Host: api.thingspeak.com
api_key=api&field1=30
Content-Type: application/x-www-form-urlencoded
```

Or you can uncomment these next lines to make a request with JSON data (instead of URL-encoded request):

```
// If you need an HTTP request with a content type: application/json, use
the following:
http.addHeader("Content-Type", "application/json");
// JSON data to send with HTTP POST
String httpRequestData = "{\"api_key\": \"" + apiKey + "\", \"field1\": \"" +
String(random(40)) + "\"}";
// Send HTTP POST request
int httpResponseCode = http.POST(httpRequestData);*/
```

Here's a sample HTTP POST request with JSON data:

```
POST /update HTTP/1.1
Host: api.thingspeak.com
{api_key: "api", field1: 30}
Content-Type: application/json
```

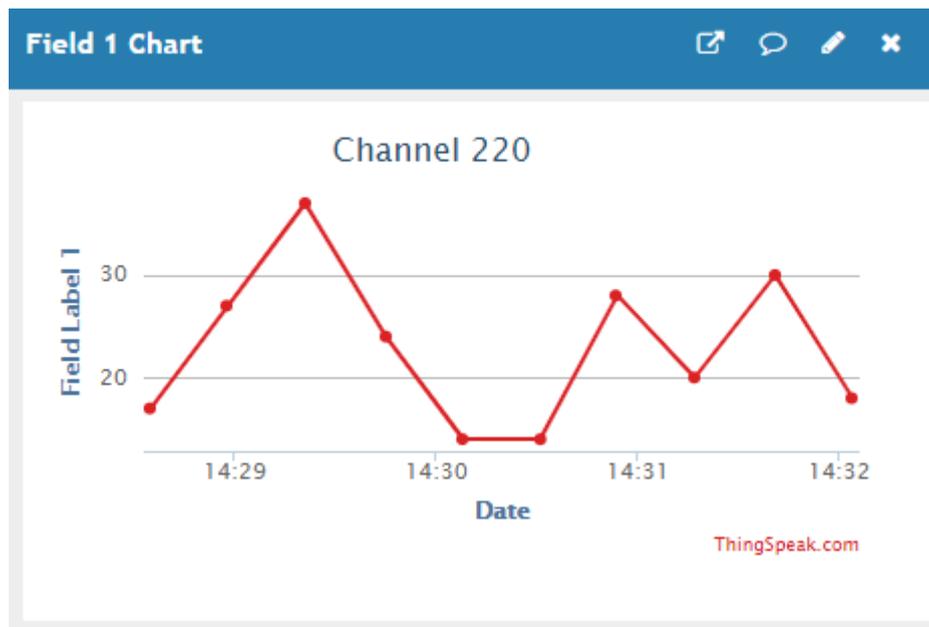
Then, the following lines print the server response code.

```
Serial.print("HTTP Response 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).

```
COM3
Connecting
.
Connected to WiFi network with IP Address: 192.168.1.75
Timer set to 10 seconds (timerDelay variable), it will take 10
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
HTTP Response code: 200
```

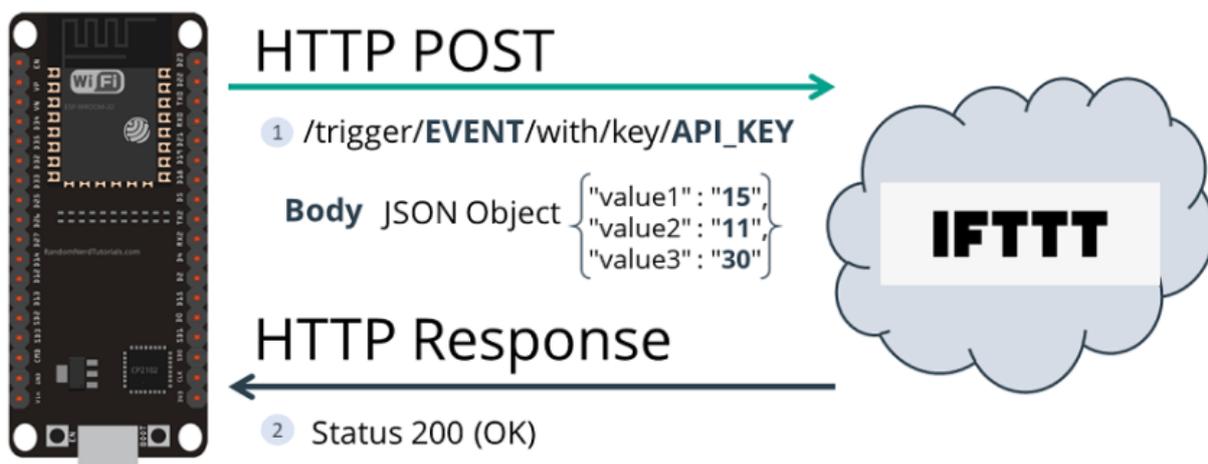
Your ThingSpeak Dashboard should be receiving new random readings every 10 seconds.



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.

2. ESP32 HTTP POST (IFTTT.com)

In this example you'll learn how to trigger a web API to send email notifications. As an example, we'll use the IFTTT.com API. IFTTT has a free plan with lots of useful automations.



Using IFTTT.com Webhooks API

IFTTT stands for "If This Than That", and it is a free web-based service to create chains of simple conditional statements called applets.

IFTTT

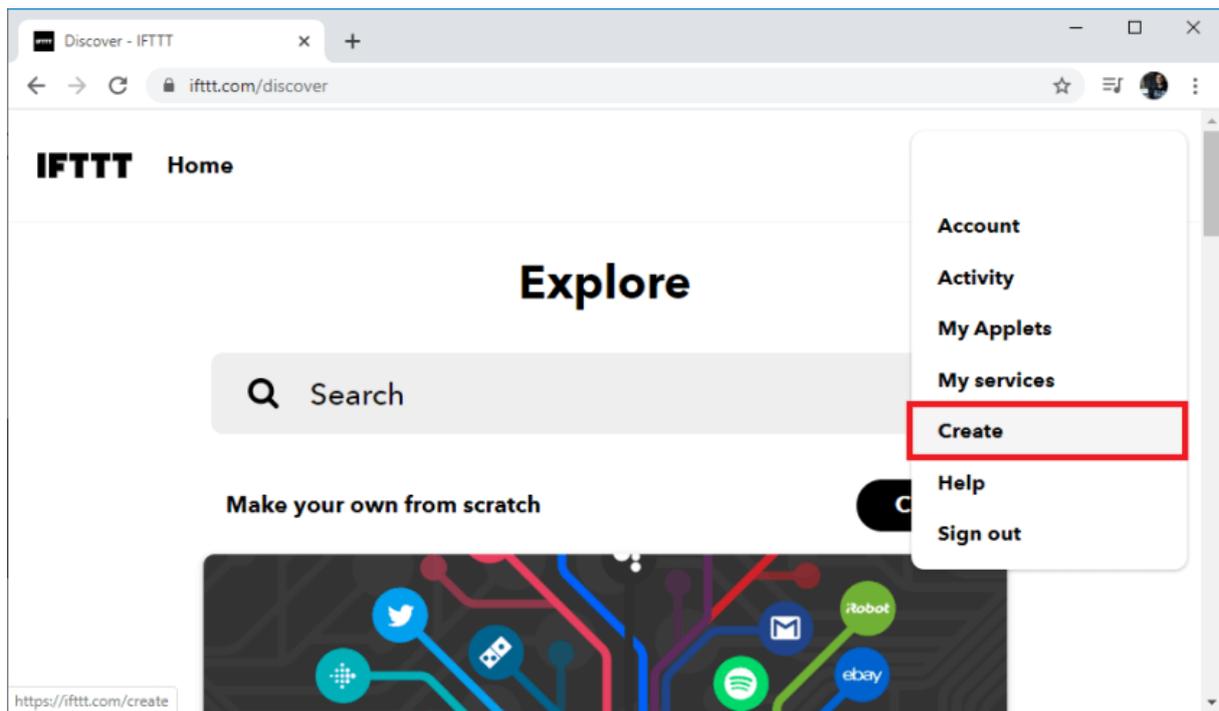
This means you can trigger an event when something happens. In this example, the applet sends three random values to your email when the ESP32 makes a request. You can replace those random values with useful sensor readings.

Creating an IFTTT Account

If you don't have an IFTTT account, go the IFTTT website: ifttt.com and enter your email to create an account and get started. Creating an account on IFTTT is free!

Next, you need to create a new applet. Follow the next steps to create a new applet:

- 1) Open the left menu and click the **"Create"** button.



- 2) Click on the **"this"** word. Search for the "Webhooks" service and select the Webhooks icon.

Choose a service

Step 1 of 6



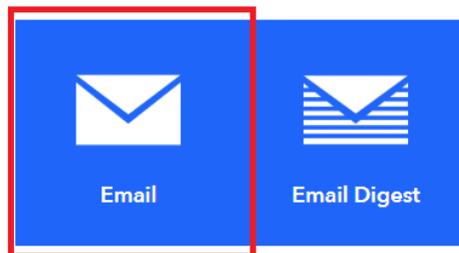
3) Choose the **“Receive a web request”** trigger and give a name to the event. In this case, I’ve typed **“test_event”**. Then, click the **“Create trigger”** button.

4) Click the **“that”** word to proceed. Now, define what happens when the event you’ve defined is triggered. Search for the **“Email”** service and select it. You can leave the default options.

Choose action service

Step 3 of 6

Q email

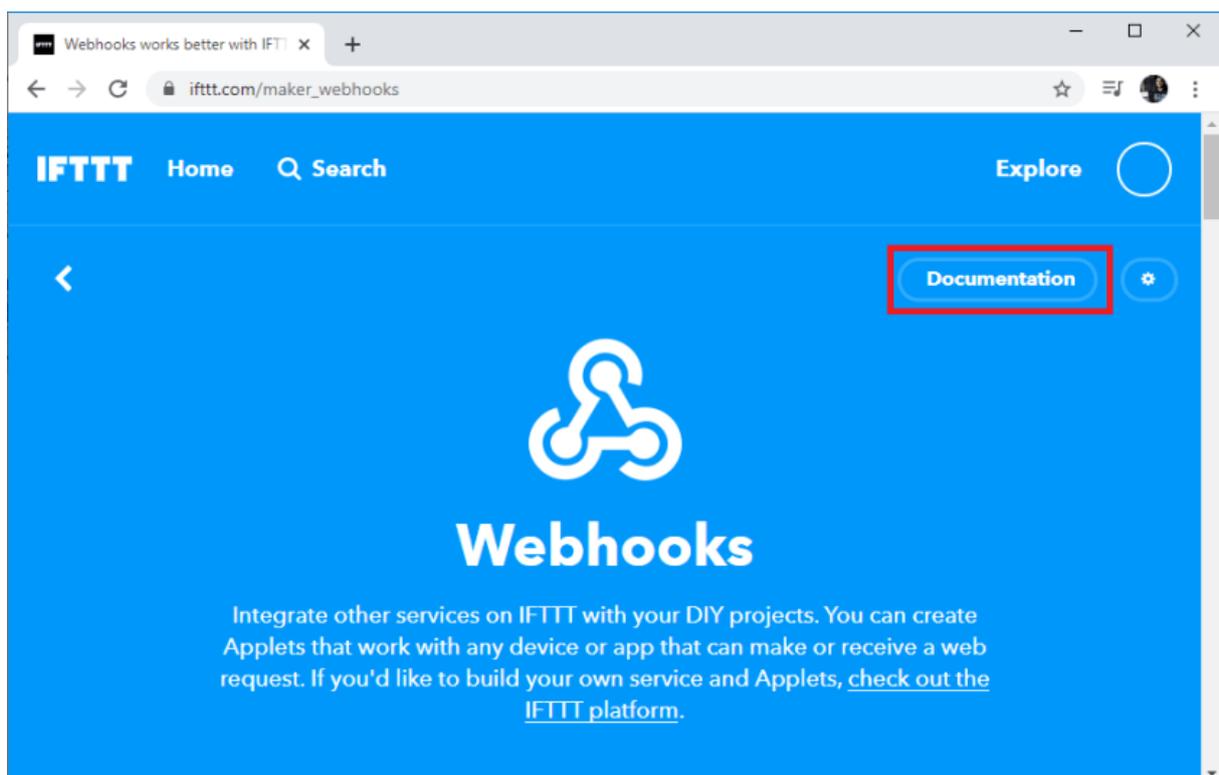


5) Press the **“Finish”** button to create your Applet.

Testing Your Applet

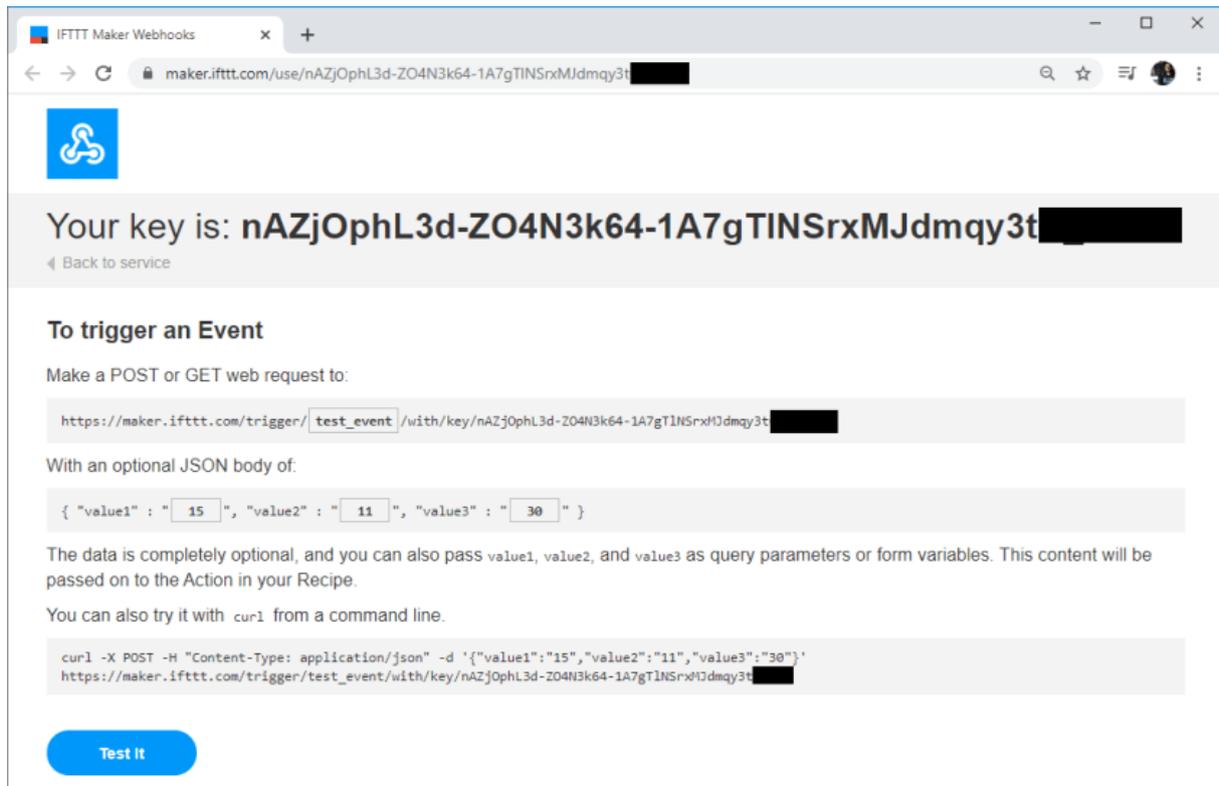
Before proceeding with the project, it’s important to test your Applet first. Follow the next steps to test it:

- 1) Search for Webhooks service or open this link: https://ifttt.com/maker_webhooks
- 2) Click the **“Documentation”** button.



A page as shown in the following figure shows up. The page shows your unique API key. You should not share your unique API key with anyone.

3) Fill the **“To trigger an Event”** section. Then, click the **“Test it”** button.



4) The event should be successfully triggered, and you'll get a green message saying **“Event has been triggered”**.

5) Go to your Email account. You should have a new email in your inbox from the IFTTT service with the values you've defined in the previous step.

If you've received an email with the data entered in the test request, it means your Applet is working as expected. Now, we need to program the ESP32 to send an HTTP POST request to the IFTTT service with the sensor readings.

Code ESP32 HTTP POST Webhooks IFTTT.com

Copy the following code to your Arduino IDE, but don't upload it yet. You need to make some changes to make it work for you.

SOURCE CODE

https://github.com/RuiSantosdotme/ESP32-Course/blob/master/code/ESP32_HTTP_GET_POST/ESP32_HTTP_POST_IFTTT/ESP32_HTTP_POST_IFTTT.ino

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

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

```

// Domain Name with full URL Path for HTTP POST Request
// REPLACE WITH YOUR EVENT NAME AND API KEY - open the documentation:
https://ifttt.com/maker webhooks
const          char*          serverName          =
"http://maker.ifttt.com/trigger/REPLACE_WITH_YOUR_EVENT/with/key/REPLACE_WI
TH_YOUR_API_KEY";
// Example:
//const          char*          serverName          =
"http://maker.ifttt.com/trigger/test event/with/key/nAZjOphL3d-ZO4N3k64-
1A7gTlNSrxMJdmqy3tC";

// 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;
// Set timer to 10 minutes (600000)
//unsigned long timerDelay = 600000;
// Timer set 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 POST request every 10 seconds
  if ((millis() - lastTime) > timerDelay) {
    //Check WiFi connection status
    if(WiFi.status()== WL_CONNECTED){
      HTTPClient http;

      // Your Domain name with URL path or IP address with path
      http.begin(serverName);

      // Specify content-type header
      http.addHeader("Content-Type", "application/x-www-form-urlencoded");
      // Data to send with HTTP POST
      String httpRequestData = "value1=" + String(random(40)) + "&value2=" +
String(random(40))+ "&value3=" + String(random(40));
      // Send HTTP POST request
      int httpResponseCode = http.POST(httpRequestData);

      /*
      // If you need an HTTP request with a content type: application/json,
use the following:
      http.addHeader("Content-Type", "application/json");
      // JSON data to send with HTTP POST

```

```

    String httpRequestData = "{\"value1\": \"" + String(random(40)) +
    "\", \"value2\": \"" + String(random(40)) + "\", \"value3\": \"" +
    String(random(40)) + "\"}";
    // Send HTTP POST request
    int httpResponseCode = http.POST(httpRequestData);
    /*
    Serial.print("HTTP Response 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. The code is well commented on where you should make the changes.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

Setting your IFTTT.com API Key

Insert your event name and API key in the following line:

```

const char* serverName = "http://maker.ifttt.com/trigger/REPLACE_WITH_YOUR_EVENT/with/key/REPLACE_WITH_YOUR_API_KEY";

```

Example URL:

```

http://maker.ifttt.com/trigger/test_event/with/key/nAZjOphL3d-ZO4N3k64-1A7gTlNSrxMJdmqy3tC

```

HTTP POST Request

In the `loop()` is where you make the HTTP POST request every 10 seconds with sample data:

```

// Specify content-type header
http.addHeader("Content-Type", "application/x-www-form-urlencoded");
// Data to send with HTTP POST
String httpRequestData = "value1=" + String(random(40)) + "&value2=" +
String(random(40)) + "&value3=" + String(random(40));
// Send HTTP POST request
int httpResponseCode = http.POST(httpRequestData);

```

The ESP32 makes a new URL encoded request to publish some random values in the `value1`, `value2` and `value3` fields. For example:

```

POST /trigger/test_event/with/key/nAZjOphL3d-ZO4N3k64-1A7gTlNSrxMJdmqy3tC HTTP/1.1
Host: maker.ifttt.com
value1=15&value2=11&value3=30
Content-Type: application/x-www-form-urlencoded

```

Alternatively, you can uncomment these next lines to make a request with JSON data:

```
/*  
// If you need an HTTP request with a content type: application/json, use  
the following:  
http.addHeader("Content-Type", "application/json");  
// JSON data to send with HTTP POST  
String httpRequestData = "{\"value1\":\":" + String(random(40)) +  
"\",\"value2\":\":" + String(random(40)) + "\",\"value3\":\":" +  
String(random(40)) + "\"}";  
// Send HTTP POST request  
int httpResponseCode = http.POST(httpRequestData);  
*/
```

Here's an example of HTTP POST request with a JSON data object.

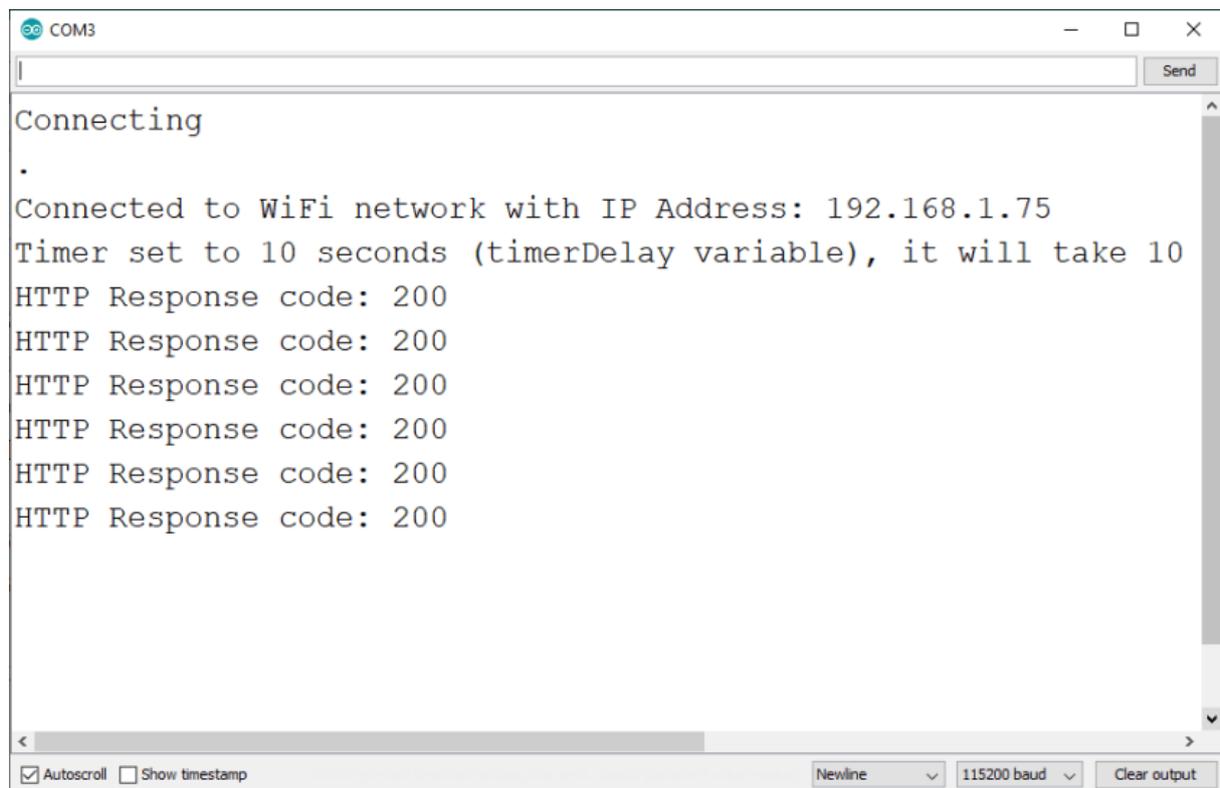
```
POST /trigger/test_event/with/key/nAZjOphL3d-ZO4N3k64-1A7gT1NSrxMJdmqy3tC HTTP/1.1  
Host: maker.ifttt.com  
{value1: 15, value2: 11, value3: 30}  
Content-Type: application/json
```

Then, the following lines of code print the HTTP response from the server.

```
Serial.print("HTTP Response code: ");  
Serial.println(httpResponseCode);
```

HTTP POST Demonstration

After uploading the code, open the Serial Monitor and you'll see a message printing the HTTP response code 200 indicating that the request has succeeded.



```
COM3  
Connecting  
.  
Connected to WiFi network with IP Address: 192.168.1.75  
Timer set to 10 seconds (timerDelay variable), it will take 10  
HTTP Response code: 200  
Autoscroll Show timestamp Newline 115200 baud Clear output
```

Go to your email account, and you should get a new email from IFTTT with three random values. In this case: 38, 20 and 13.

The event named "test_event" occurred on the Maker Webhooks service ➤



Webhooks via IFTTT <action@ifttt.com>
to me ▾

14:34 (1 minute ago)



What: test_event

When: May 11 at 02:34PM

Extra Data: 38, 20, 13



If Maker Event "test_event", then Send me an email at



For demonstration purposes, we're publishing new data every 10 seconds. However, for a long term project you should increase the timer or check the API call limits per hour/minute to avoid getting blocked/banned.

Wrapping Up

In this tutorial you've learned how to integrate your ESP32 with web services using HTTP POST requests. You can also make HTTP GET requests with the ESP32 (see the previous unit).

ESP32 Over-the-air (OTA) Programming – Web Updater

This Unit is a quick guide that shows how to do over-the-air (OTA) programming with the ESP32 using the OTA Web Updater in Arduino IDE.



Introduction

The OTA Web Updater allows you to update/upload new code to your ESP32 using a browser, without the need to make a serial connection between the ESP32 and your computer.

OTA programming is useful when you need to update code to ESP32 boards that are not easily accessible. The example we'll show here works when the ESP32 and your browser are on your local network.

The only disadvantage of the OTA Web Updater is that you have to add the code for OTA in every sketch you upload, so that you're able to use OTA in the future.

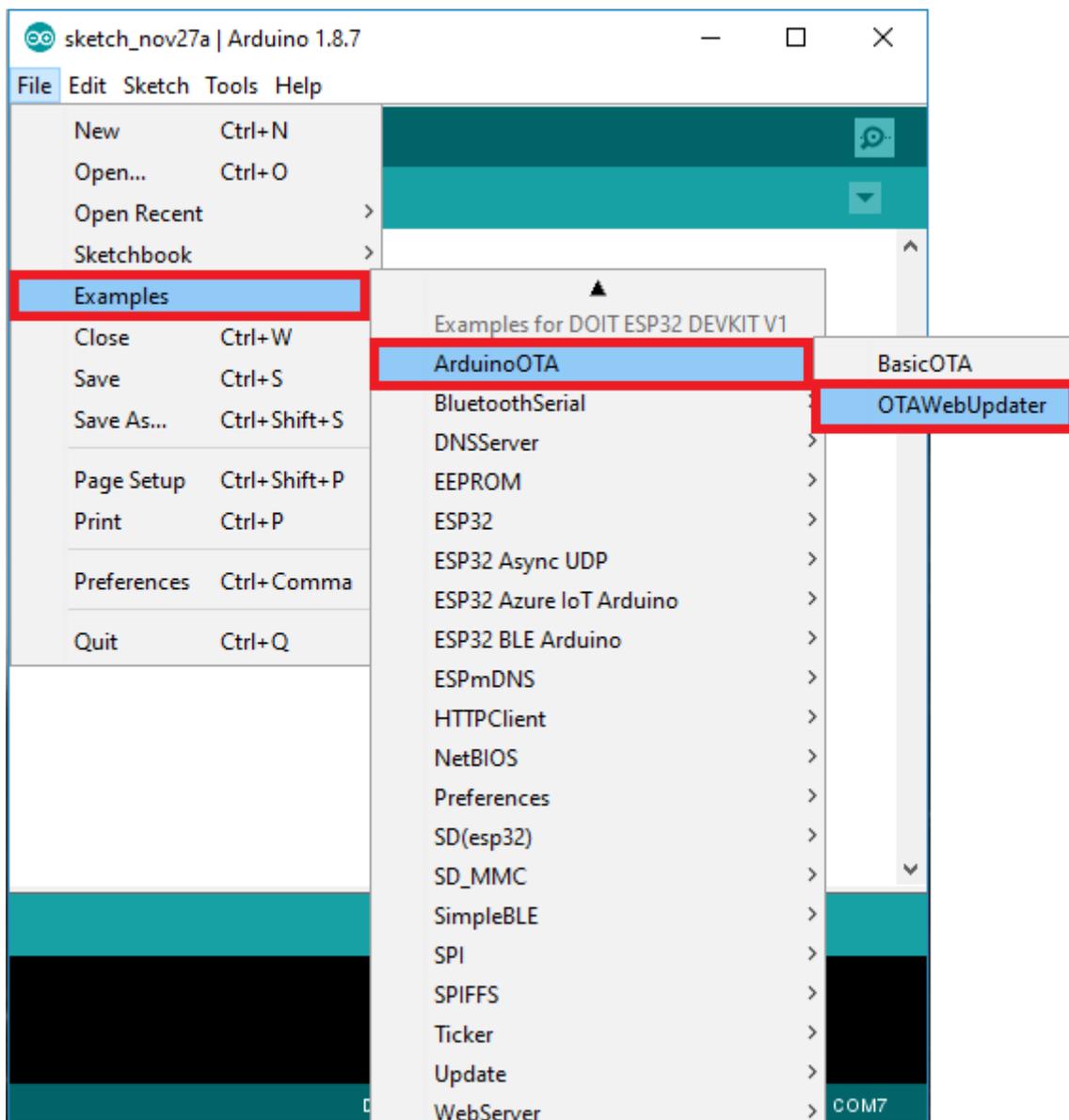
How does OTA Web Updater Works?

- The first sketch should be uploaded via serial port. This sketch should contain the code to create the OTA Web Updater, so that you are able to upload code later using your browser.
- The OTA Web Updater sketch creates a web server you can access to upload a new sketch via web browser.

- Then, you need to implement OTA routines in every sketch you upload, so that you're able to do the next updates/uploads over-the-air.
- If you upload a code without a OTA routine you'll no longer be able to access the web server and upload a new sketch over-the-air.

ESP32 OTA Web Updater

When you install the ESP32 add-on for the Arduino IDE, it will automatically install the ArduinoOTA library. Go to **File** ▶ **Examples** ▶ **ArduinoOTA** ▶ **OTAWebUpdater**.



The following code should load.

SOURCE CODE

https://github.com/RuiSantosdotme/Random-Nerd-Tutorials/blob/master/Projects/ESP32/OTA_Web_Updater.ino

```

#include <WiFi.h>
#include <WiFiClient.h>
#include <WebServer.h>
#include <ESPmDNS.h>
#include <Update.h>
const char* host = "esp32";
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

WebServer server(80);

/*
 * Login page
 */
const char* loginIndex =
"<form name='loginForm'>"
  "<table width='20%' bgcolor='A09F9F' align='center'>"
    "<tr>"
      "<td colspan=2>"
        "<center><font                size=4><b>ESP32                Login"
Page</b></font></center>"
        "<br>"
      "</td>"
      "<br>"
      "<br>"
    "</tr>"
    "<td>Username:</td>"
    "<td><input type='text' size=25 name='userid'><br></td>"
    "</tr>"
    "<br>"
    "<br>"
    "<tr>"
      "<td>Password:</td>"
      "<td><input type='Password' size=25 name='pwd'><br></td>"
      "<br>"
      "<br>"
    "</tr>"
    "<tr>"
      "<td><input    type='submit'    onclick='check(this.form)'"
value='Login'></td>"
      "</tr>"
    "</table>"
"</form>"
"<script>"
  "function check(form)"
  "{"
  "if(form.userid.value=='admin' && form.pwd.value=='admin')"

```

```

const char* serverIndex =
"<script
src='https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js'></script>"
"<form method='POST' action='#' enctype='multipart/form-data' id='upload_form'>"
  "<input type='file' name='update'>"
  "<input type='submit' value='Update'>"
"</form>"
"<div id='prg'>progress: 0%</div>"
"<script>"
  "$('form').submit(function(e){"
  "e.preventDefault();"
  "var form = $('#upload_form')[0];"
  "var data = new FormData(form);"
  " $.ajax({"
  "url: '/update',"
  "type: 'POST',"
  "data: data,"
  "contentType: false,"
  "processData:false,"
  "xhr: function() {"
  "var xhr = new window.XMLHttpRequest();"
  "xhr.upload.addEventListener('progress', function(evt) {"
  "if (evt.lengthComputable) {"
  "var per = evt.loaded / evt.total;"
  "$('#prg').html('progress: ' + Math.round(per*100) + '%');"
  }}"
  "}, false);"
  "return xhr;"
  "},"
  "success:function(d, s) {"
  "console.log('success!)"
  "},"
  "error: function (a, b, c) {"
  "}"
  "});"
  "});"
"</script>";

/*
 * setup function
 */
void setup(void) {
  Serial.begin(115200);

  // Connect to WiFi network
  WiFi.begin(ssid, password);
  Serial.println("");

  // Wait for connection
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");

```

```

Serial.println(WiFi.localIP());

/*use mdns for host name resolution*/
if (!MDNS.begin(host)) { //http://esp32.local
  Serial.println("Error setting up MDNS responder!");
  while (1) {
    delay(1000);
  }
}
Serial.println("mDNS responder started");
/*return index page which is stored in serverIndex */
server.on("/", HTTP_GET, []() {
  server.sendHeader("Connection", "close");
  server.send(200, "text/html", loginIndex);
});
server.on("/serverIndex", HTTP_GET, []() {
  server.sendHeader("Connection", "close");
  server.send(200, "text/html", serverIndex);
});
/*handling uploading firmware file */
server.on("/update", HTTP_POST, []() {
  server.sendHeader("Connection", "close");
  server.send(200, "text/plain", (Update.hasError()) ? "FAIL" :
"OK");
  ESP.restart();
}, []() {
  HTTPUpload& upload = server.upload();
  if (upload.status == UPLOAD_FILE_START) {
    Serial.printf("Update: %s\n", upload.filename.c_str());
    if (!Update.begin(UPDATE_SIZE_UNKNOWN)) { //start with max
available size
      Update.printError(Serial);
    }
  } else if (upload.status == UPLOAD_FILE_WRITE) {
    /* flashing firmware to ESP*/
    if (Update.write(upload.buf, upload.currentSize) !=
upload.currentSize) {
      Update.printError(Serial);
    }
  } else if (upload.status == UPLOAD_FILE_END) {
    if (Update.end(true)) { //true to set the size to the current
progress
      Serial.printf("Update Success: %u\nRebooting...\n",
upload.totalSize);
    } else {
      Update.printError(Serial);
    }
  }
});
server.begin();
}
void loop(void) {
  server.handleClient();
  delay(1);
}

```

The OTAWebUpdater example for the ESP32 creates an asynchronous web server where you can upload new code to your board without the need for a serial connection.

Upload the previous code to your ESP32 board. Don't forget to enter your network credentials and select the right board and serial port.



After uploading the code, open the Serial Monitor at a baud rate of 115200, press the ESP32 enable button, and you should get the ESP32 IP address:

```
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:0x3fff0018,len:4
load:0x3fff001c,len:808
load:0x40078000,len:6084
load:0x40080000,len:6696
entry 0x400802e4

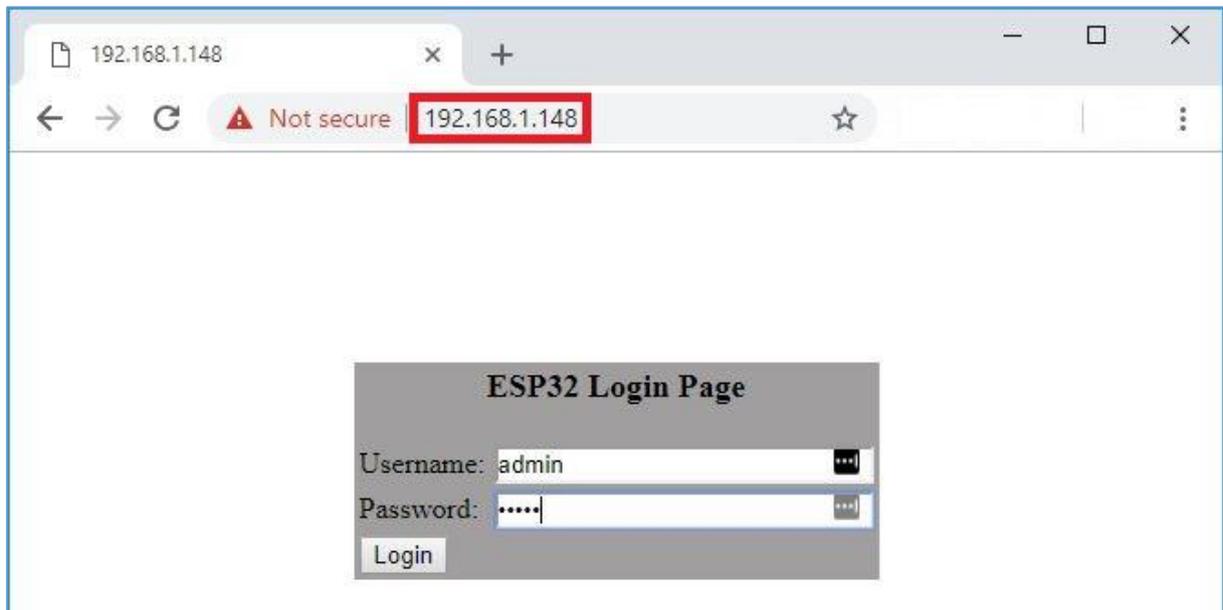
..
Connected to MEO-620B4B
IP address: 192.168.1.148
mDNS responder started
```

Now, you can upload code to your ESP32 over-the-air using a browser on your local network.

To test the OTA Web Updater you can disconnect the ESP32 from your computer and power it using a power bank, for example (this is optional, we're suggesting this to mimic a situation in which the ESP32 is not connected to your computer).

Update New Code using OTA Web Updater

Open a browser in your network and enter the ESP32 IP address. You should get the following:



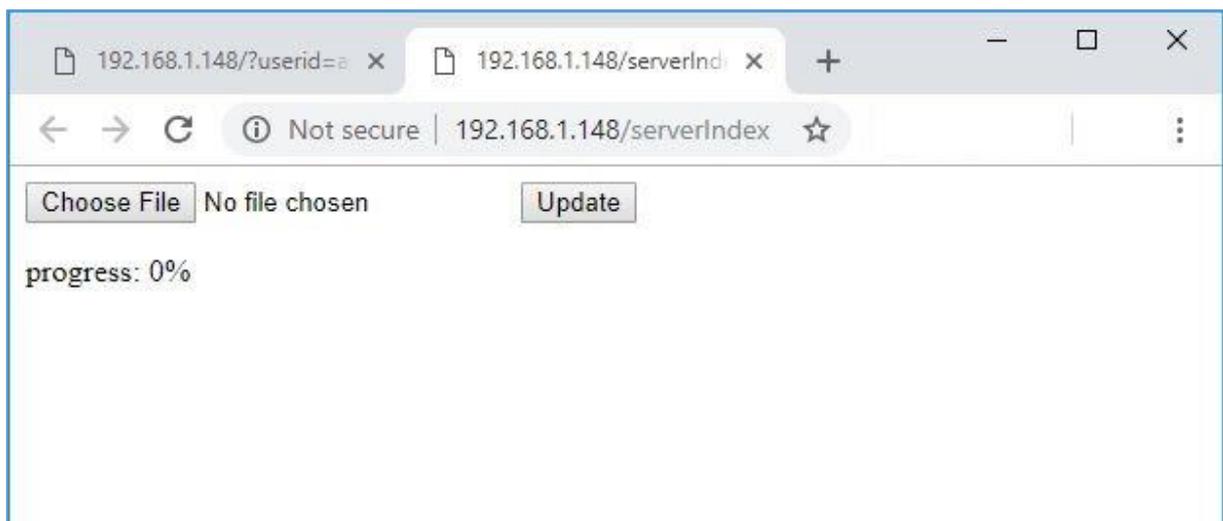
Enter the username and the password:

- **Username:** admin
- **Password:** admin

You can change the username and password on the code.

Note: After you enter the username and password, you are redirected to the `/serverIndex` URL. You don't need to enter the username and password to access the `/serverIndex` URL. So, if someone knows the URL to upload new code, the username and password don't protect the web page from being accessible from others.

A new tab should open on the `/serverIndex` URL. This page allows you to upload a new code to your ESP32. You should upload `.bin` files (we'll see how to do that in a moment).



Preparing the New Sketch

When uploading a new sketch over-the-air, you need to keep in mind that you need to add code for OTA in your new sketch, so that you can always overwrite any sketch with a

new one in the future. So, we recommend that you modify the OTAWebUpdater sketch to include your own code.

For learning purposes let's upload a new code that blinks an LED (without delay). Copy the following code to your Arduino IDE.

SOURCE CODE

https://github.com/RuiSantosdotme/Random-Nerd-Tutorials/blob/master/Projects/ESP32/OTA_Web_Updater_LED.ino

```
#include <WiFi.h>
#include <WiFiClient.h>
#include <WebServer.h>
#include <ESPmDNS.h>
#include <Update.h>

const char* host = "esp32";
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

//variables to blink without delay:
const int led = 2;
unsigned long previousMillis = 0;          // will store last time LED
was updated
const long interval = 1000;              // interval at which to blink
(millisecons)
int ledState = LOW;                      // ledState used to set the LED

WebServer server(80);

/*
 * Login page
 */

const char* loginIndex =
  "<form name='loginForm'>"
  "  <table width='20%' bgcolor='A09F9F' align='center'>"
  "    <tr>"
  "      <td colspan=2>"
  "        <center><font                size=4><b>ESP32                Login
Page</b></font></center>"
  "        <br>"
  "      </td>"
  "    <br>"
  "    <br>"
  "  </tr>"
  "  <td>Username:</td>"
  "  <td><input type='text' size=25 name='userid'><br></td>"
  "</tr>"
  "<br>"
  "<br>"
  "<tr>"
  "  <td>Password:</td>"
  "  <td><input type='Password' size=25 name='pwd'><br></td>"
  "  <br>"
  "  <br>"
  "</tr>"
```

```

        "<tr>"
            "<td><input    type='submit'    onclick='check(this.form) '
value='Login'></td>"
        "</tr>"
    "</table>"
"</form>"
"<script>"
    "function check(form)"
    "{"
    "if(form.userid.value=='admin' && form.pwd.value=='admin')"

```

```

});"
});"
"</script>";

/*
 * setup function
 */
void setup(void) {
  pinMode(led, OUTPUT);

  Serial.begin(115200);

  // Connect to WiFi network
  WiFi.begin(ssid, password);
  Serial.println("");

  // Wait for connection
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  /*use mdns for host name resolution*/
  if (!MDNS.begin(host)) { //http://esp32.local
    Serial.println("Error setting up MDNS responder!");
    while (1) {
      delay(1000);
    }
  }
  Serial.println("mDNS responder started");
  /*return index page which is stored in serverIndex */
  server.on("/", HTTP_GET, []() {
    server.sendHeader("Connection", "close");
    server.send(200, "text/html", loginIndex);
  });
  server.on("/serverIndex", HTTP_GET, []() {
    server.sendHeader("Connection", "close");
    server.send(200, "text/html", serverIndex);
  });
  /*handling uploading firmware file */
  server.on("/update", HTTP_POST, []() {
    server.sendHeader("Connection", "close");
    server.send(200, "text/plain", (Update.hasError()) ? "FAIL" :
"OK");
    ESP.restart();
  }, []() {
    HTTPUpload& upload = server.upload();
    if (upload.status == UPLOAD_FILE_START) {
      Serial.printf("Update: %s\n", upload.filename.c_str());
      if (!Update.begin(UPDATE_SIZE_UNKNOWN)) { //start with max
available size
        Update.printError(Serial);
      }
    } else if (upload.status == UPLOAD_FILE_WRITE) {
      /* flashing firmware to ESP*/

```

```

        if (Update.write(upload.buf, upload.currentSize) !=
upload.currentSize) {
            Update.printError(Serial);
        }
    } else if (upload.status == UPLOAD_FILE_END) {
        if (Update.end(true)) { //true to set the size to the current
progress
            Serial.printf("Update Success: %u\nRebooting...\n",
upload.totalSize);
        } else {
            Update.printError(Serial);
        }
    }
}
});
server.begin();
}
void loop(void) {
    server.handleClient();
    delay(1);
    //loop to blink without delay
    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:
        ledState = not(ledState);
        // set the LED with the ledState of the variable:
        digitalWrite(led, ledState);
    }
}
}
}

```

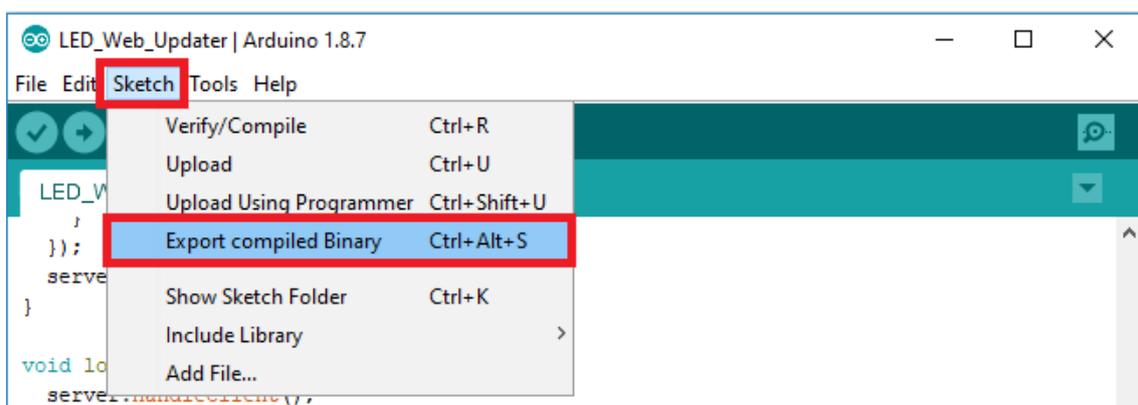
As you can see, we've added the "blink without delay" code to the OTAWebUpdater code, so that we're able to make updates later on.

After copying the code to your Arduino IDE, you should generate a *.bin* file.

Generate a *.bin* file in Arduino IDE

Save your sketch as *LED_Web_Updater*.

To generate a *.bin* file from your sketch, go to **Sketch ▶ Export compiled Binary**.



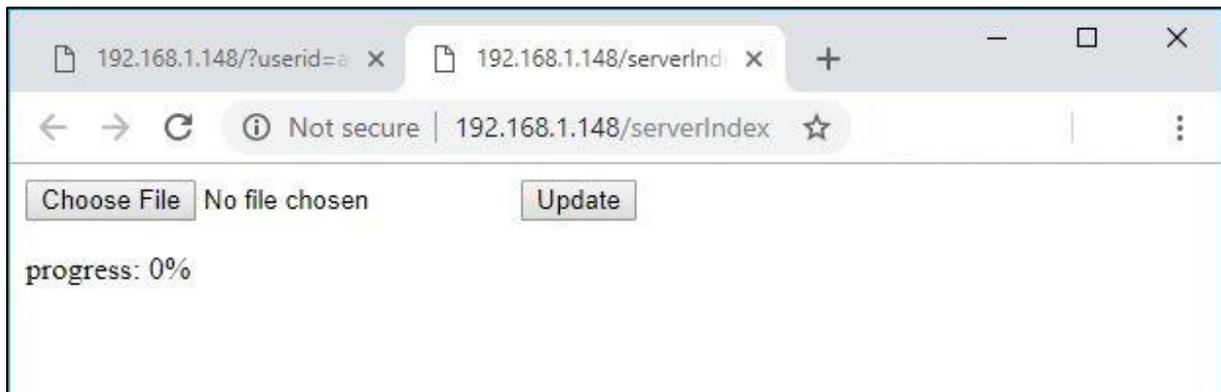
A new file on the folder sketch should be created. Go to **Sketch ▶ Show SketchFolder**. You should have two files in your Sketch folder: the *.ino* and the *.bin* file. You should upload the *.bin* file using the OTA Web Updater.



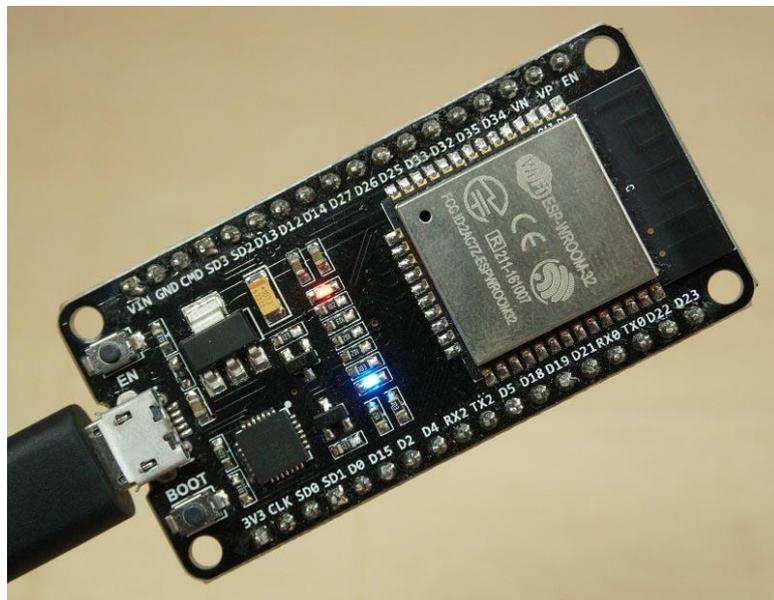
Upload a new sketch over-the-air to the ESP32

In your browser, on the ESP32 OTA Web Updater page, click the **Choose File** button. Select the *.bin* file generated previously, and then click **Update**.

After a few seconds, the code should be successfully uploaded.



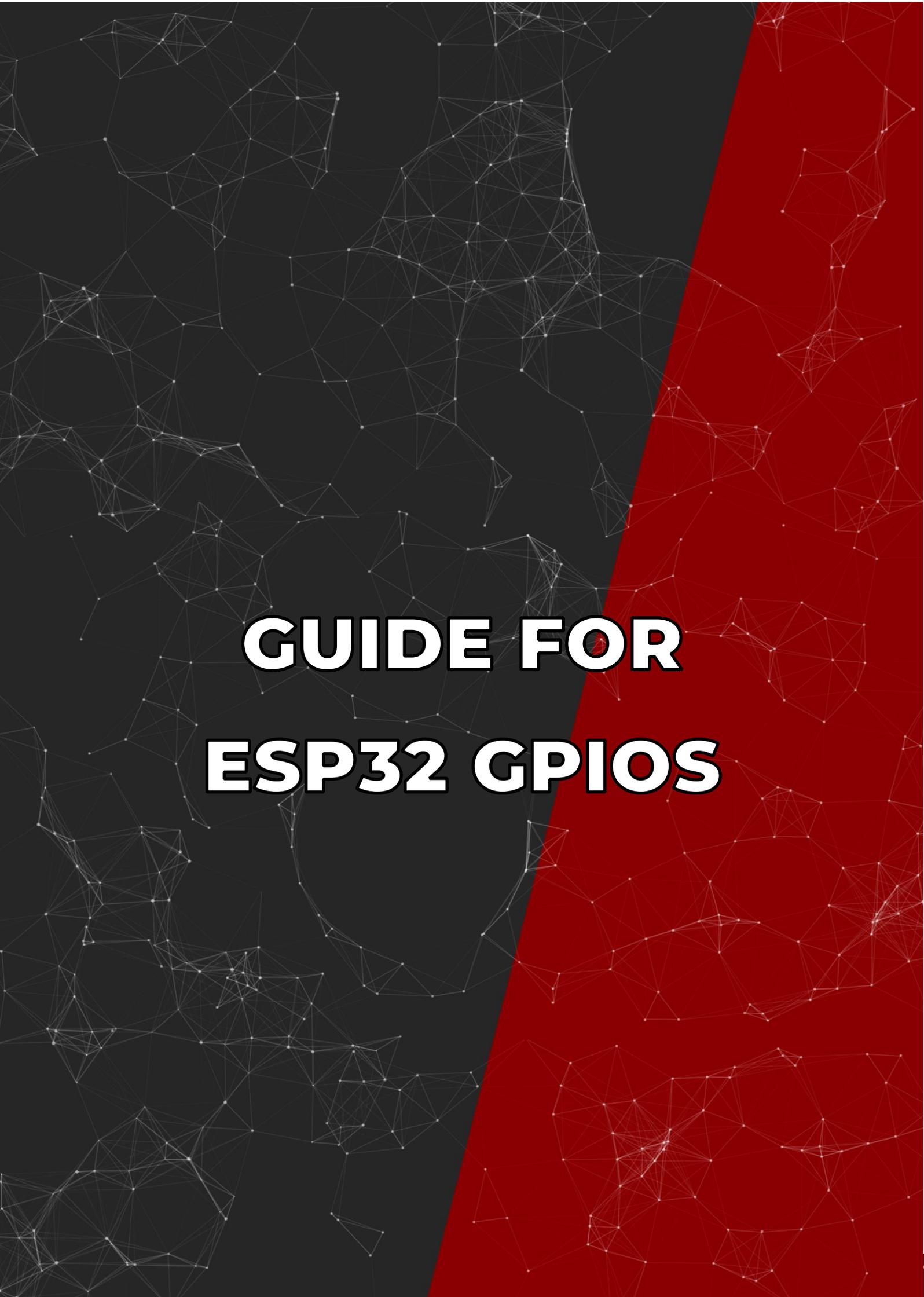
The ESP32 built-in LED should be blinking.



Congratulations! You've uploaded a new code to your ESP32 over-the-air.

Wrapping Up

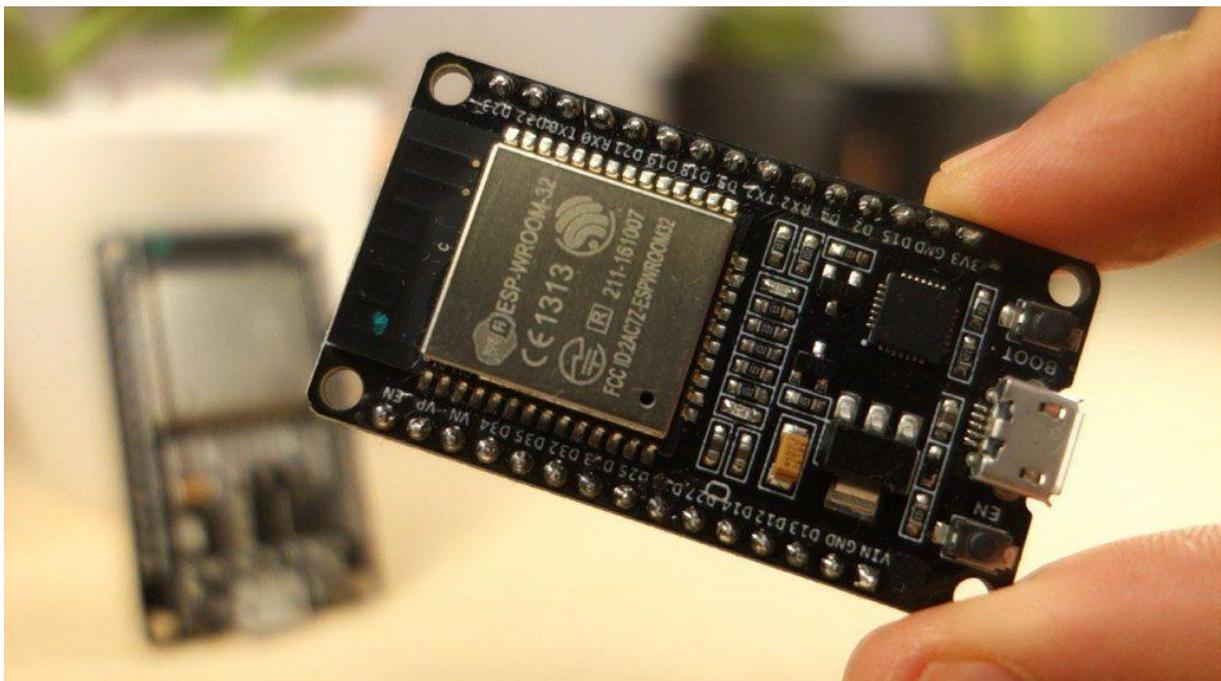
Over-the-air updates are useful to upload new code to your ESP32 board when it is not easily accessible. The OTA Web Updater code creates a web server that you can access to upload new code to your ESP32 board using a web browser on your local network.



GUIDE FOR ESP32 GPIOS

ESP32 Pinout Reference: Which GPIO pins should you use?

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



ESP32 Peripherals

The ESP32 peripherals include:

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

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

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
3	OK	RX pin	HIGH at boot
4	OK	OK	
5	OK	OK	outputs PWM signal at boot
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
13	OK	OK	
14	OK	OK	outputs PWM signal at boot
15	OK	OK	outputs PWM signal at boot
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-ups 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)

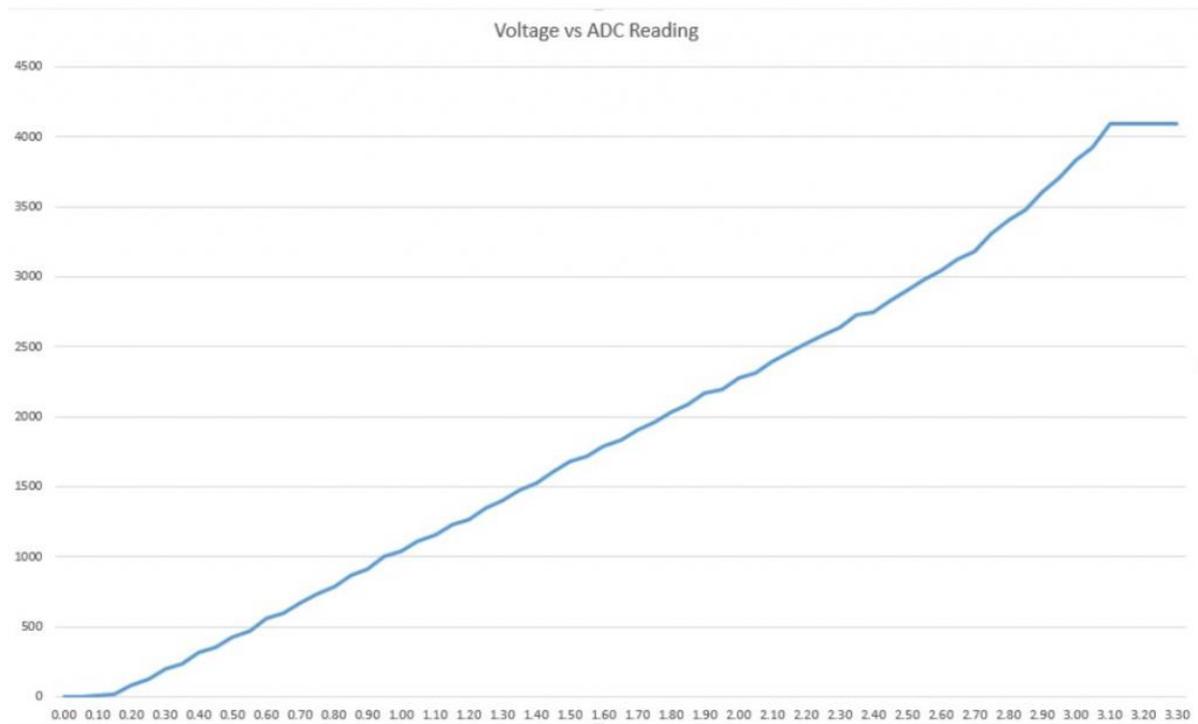
The ESP32 has 18 x 12 bits ADC input channels (while the ESP8266 only has 1x 10 bits ADC). These are the GPIOs that can be used as ADC and respective channels:

- ADC1_CH0 (GPIO 36)
- ADC1_CH1 (GPIO 37)
- ADC1_CH2 (GPIO 38)
- ADC1_CH3 (GPIO 39)
- ADC1_CH4 (GPIO 32)
- ADC1_CH5 (GPIO 33)
- ADC1_CH6 (GPIO 34)
- ADC1_CH7 (GPIO 35)
- ADC2_CH0 (GPIO 4)
- ADC2_CH1 (GPIO 0)
- ADC2_CH2 (GPIO 2)
- ADC2_CH3 (GPIO 15)
- ADC2_CH4 (GPIO 13)
- ADC2_CH5 (GPIO 12)
- ADC2_CH6 (GPIO 14)
- ADC2_CH7 (GPIO 27)
- ADC2_CH8 (GPIO 25)
- ADC2_CH9 (GPIO 26)

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

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

- DAC1 (GPIO25)
- DAC2 (GPIO26)

RTC GPIOs

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

- RTC_GPIO0 (GPIO36)
- RTC_GPIO3 (GPIO39)
- RTC_GPIO4 (GPIO34)
- RTC_GPIO5 (GPIO35)
- RTC_GPIO6 (GPIO25)
- RTC_GPIO7 (GPIO26)
- RTC_GPIO8 (GPIO33)
- RTC_GPIO9 (GPIO32)
- RTC_GPIO10 (GPIO4)
- RTC_GPIO11 (GPIO0)
- RTC_GPIO12 (GPIO2)

- RTC_GPIO13 (GPIO15)
- RTC_GPIO14 (GPIO13)
- RTC_GPIO15 (GPIO12)
- RTC_GPIO16 (GPIO14)
- RTC_GPIO17 (GPIO27)

PWM

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

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

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

I2C

When using the ESP32 with the Arduino IDE, you should use the ESP32 I2C default pins (supported by the Wire library):

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

SPI

By default, the pin mapping for SPI is:

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

Interrupts

All GPIOs can be configured as interrupts.

Strapping Pins

The ESP32 chip has the following strapping pins:

- GPIO 0
- GPIO 2
- GPIO 4

- GPIO 5
- GPIO 12
- GPIO 15

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

However, if you have peripherals connected to those pins, you may have trouble trying to upload new code, flashing the ESP32 with new firmware or resetting the board. If you have some peripherals connected to the strapping pins and you are getting trouble uploading code or flashing the ESP32, it may be because those peripherals are preventing the ESP32 to enter 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 its state to HIGH or output PWM signals at boot or reset. This means that if you have outputs connected to these GPIOs you may get unexpected results when the ESP32 resets or boots.

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

Enable (EN)

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

Final Thoughts

Congratulations for completing this course!

If you followed all the Modules presented in this course, now you should have mastered how to program the ESP32 with the Arduino IDE. 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 read sensors;
- 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 8

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 and see exactly what you want to build and get the components accordingly.

Important: all components and parts are linked to our website [MakerAdvisor.com/tools](https://makeradvisor.com/tools) where you can compare prices in more than 8 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 \(HC-SR501\)](#)
- [Relay module](#)
- 12V lamp
- 12V lamp holder
- [Male DC barrel jack 2.1mm](#)
- [12V power adaptor](#)
- [BME280 sensor module](#)
- [Micro Servo Motor – S0009](#) or [Servo Motor – S0003](#)
- [DHT11/DHT22](#) temperature and humidity sensor
- [4.7k Ohm resistor](#)
- [RGB LED Strip \(5V\)](#)
- 3x NPN transistors (see project description)
- 3x [1k ohm resistors](#)
- [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 – 16GB 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](#)

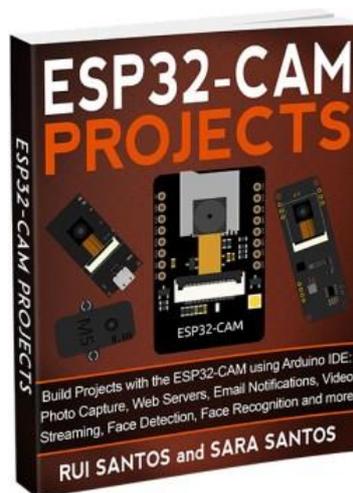
Download Other RNT Products

[Random Nerd Tutorials](#) is an online resource with electronics projects, tutorials and reviews. Creating and posting new projects takes a lot of time. At this moment, Random Nerd Tutorials has more than 200 free blog posts with complete tutorials using open-source hardware that anyone can read, remix and apply to their own projects.

To keep free tutorials coming, there's also paid content or as we like to call "premium content". To support Random Nerd Tutorials you can [download premium content here](#). If you enjoyed this eBook/course, make sure you [check all the others](#).

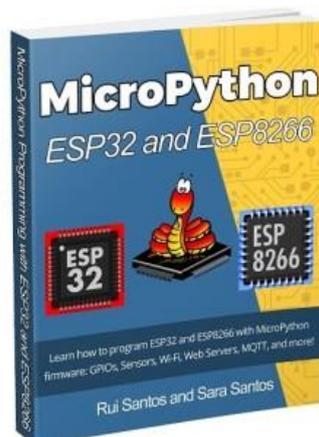
Build ESP32-CAM Projects using Arduino IDE eBook

Build 17 projects with the ESP32-CAM using Arduino IDE: photo capture, web servers, email notifications, video streaming, car robot, pan and tilt server, face detection, face recognition and much more. [Read product description](#).



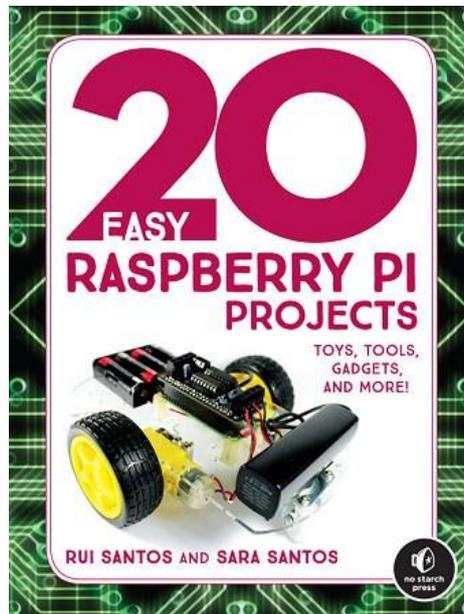
MicroPython Programming with ESP32 and ESP8266

Learn how to program the ESP32 and ESP8266 with MicroPython, a re-implementation of Python 3 programming language targeted for microcontrollers. This is one of the easiest ways to program your ESP32/ESP8266 boards! [Read product description](#).



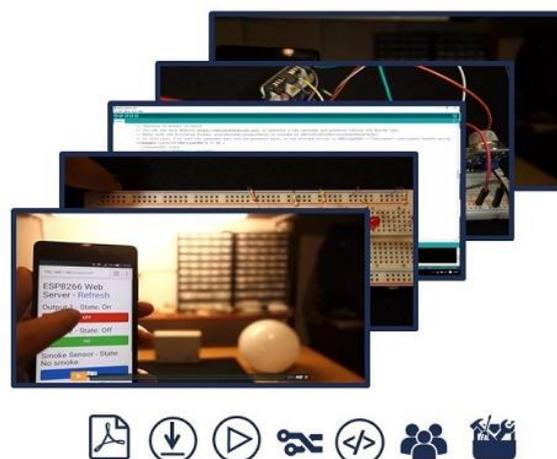
20 Easy Raspberry Pi Projects

20 Easy Raspberry Pi Projects book is a beginner-friendly collection of electronics projects using the Raspberry Pi. The Raspberry Pi is a tiny and affordable computer, for beginners looking to make cool things right away. Projects are explained with full-color visuals and simple step-by-step instructions. This book was a collaboration with the NoStarch Press Publisher and it is available in paperback format. [Read product description.](#)



Home Automation Using ESP8266 (4th Edition)

This course is a step-by-step guide designed to help you get started with this the ESP8266. If you're new to the world of ESP8266, this eBook is perfect for you. If you already used the ESP8266 before, I'm sure you'll also learn something new. [Read product description.](#)



Build a Home Automation System for \$100

Learn Raspberry Pi, ESP8266, Arduino and Node-RED. This is a premium step-by-step course to get you building a real world home automation system using open-source hardware and software. [Read product description.](#)



Arduino Step-by-step Projects

Our step-by-step course to get you building cool Arduino projects even with no prior experience! This Arduino Course is a compilation of 25 projects divided in 5 Modules that you can build by following clear step-by-step instructions with schematics and downloadable code. [Read product description.](#)



Android Apps for Arduino with MIT App Inventor 2

This eBook is our step-by-step guide designed to get you building cool Android applications for Arduino, even with no prior experience! Android Apps for Arduino with MIT App Inventor 2 is a practical course in which you're going to build 8 Android applications to interact with the Arduino. [Read product description.](#)

