LROBRUYA

ESP32 Basic Starter Kit with expansion board

LROBRUYA


Wireless Control
ESP32 DHT Server
Temperature 17.00
Humidity 72.00

LROBRUYA

Content

Preface

Company Profile

Founded in 2014, Shenzhen Lonten Technology Co., Ltd. focuses on the design, research production of Electronics Module for robotics related products. Consisting of professional researchers and skilled engineers, our R&D team constantly strives for creative function and excellent user experience. The company's R&D investments on arduino kits raspberry pi kits, as well as 3D printer and robots that back up STEAM education.

Customer Service

Our self-owned factory is certificated with BSCI and SO, covering an area of 5,000 square meters, and achieving an annual production capacity of over 10,000 units. Our products are all certified to CE, FCC, and ROHS standards, have exported to more than 100 countries including, but not limited to France, the United States of America, Australia, Russia, the United Kingdom, Germany, Singapore, Egypt, and India, bringing technological innovation to all walks of life.

Tutorial

This tutorial include codes, libraries and detailed user documentation. It is designed for beginners. You will learn all the basic knowledge about how to use ESP32 board, sensors and components.

LROBRUYA

ESP32 Introduction

New to ESP32? Start here! The ESP32 is a series of low-cost and low-power.

System on a Chip (SoC) microcontrollers developed by Espress if that include Wi-Fi and Bluetooth wireless capabilities and dual-core processor.

If you're familiar with the ESP8266, the ESP32 is its successor, loaded with lots of new features.

ESP32 Specifications

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

- Low cost: You can purchase ESP 32 from $6, which makes it easy for

the general public to obtain it;

- Low power consumption: Compared with other microcontrollers, ESP 32 has extremely low power consumption and supports low-power mode states such as deep sleep to save electricity;
- WiFi function: ESP 32 can easily connect to a WiFi network to connect to the internet (station mode), or create its own WiFi wireless network (access point mode) so that other devices can connect to it - this is crucial for IoT and home automation projects - you can have multiple devices communicate with each other using their WiFi function; Bluetooth: ESP 32 supports classic Bluetooth and low-power Bluetooth (BLE) - suitable for various IoT applications;
- Dual Core: Most ESP 32s are dual core - they come with two Xtensa 32-bit LX6 microprocessors: core O And Core 1
- Rich peripheral input! Output interface - ESP 32 supports various inputs (reading data from the outside) and outputs (sending commands/signals to the outside) such as capacitive touch, ADC, DAC, UART, SPI, I2C, and peripheral devices, pulse width modulation, etc.
- Compatible with the Arduino "programming language": For those who are already familiar with Arduino board programming, you will

be happy to know that they can program ESP 32 using the Arduino style.

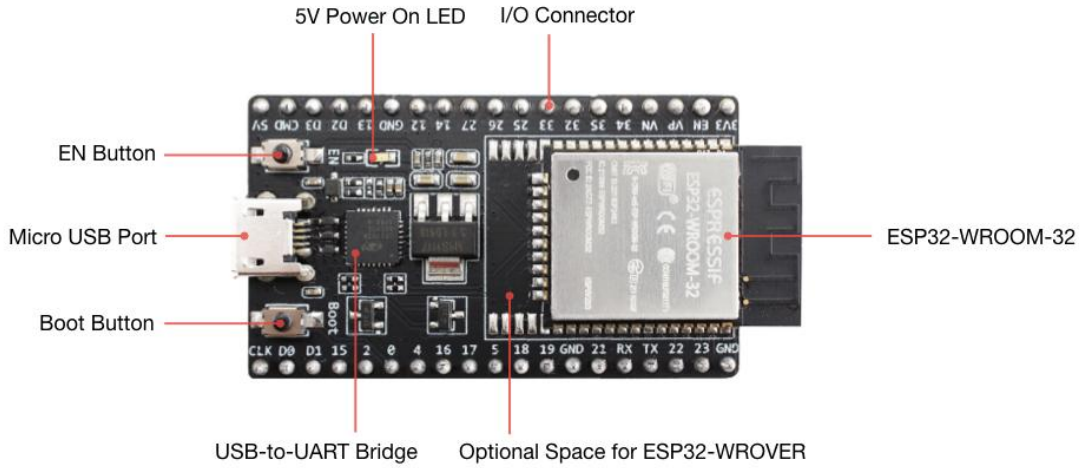
ESP32 Development Boards

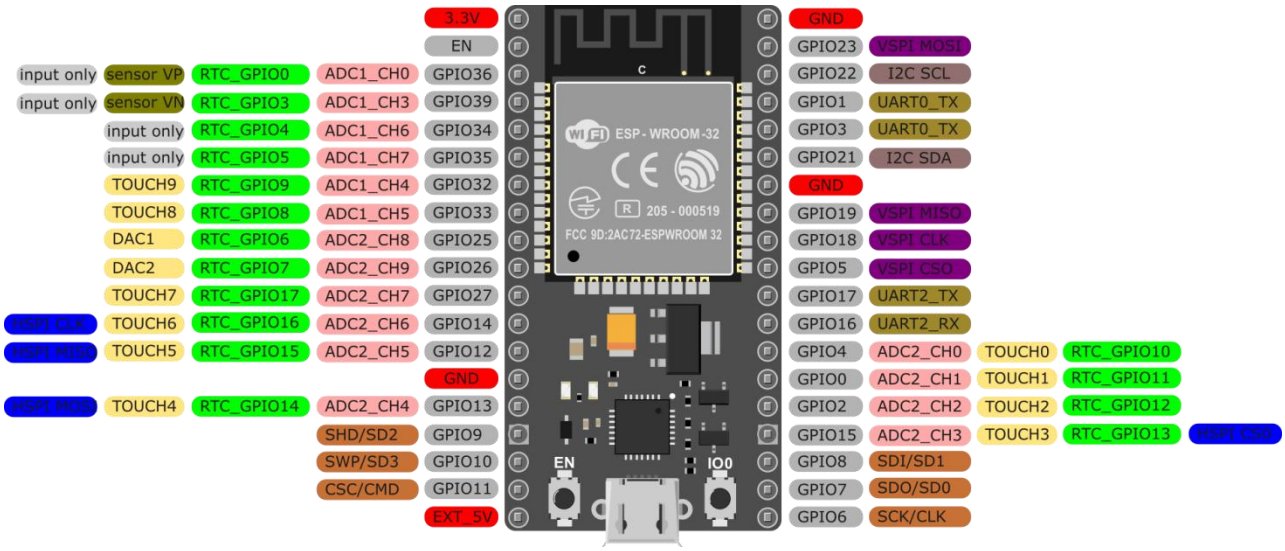
ESP32 refers to the bare ESP32 chip. However, the “ESP32” term is also used to refer to ESP32 development boards. Using ESP32 bare chips is not easy or practical, especially when learning, testing, and prototyping. Most of the time, you’ll want to use an ESP32 development board.

We’ll be using the ESP32-DevKitC V4 board as a reference.The picture below shows the ESP32-DevKitC V4 board, version with 38 GPIO pins.

Function Description

The main components, interfaces, and control methods of the ESP32 DevKitC V4 development board are shown below:





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

GPIO	Pad Name	Function 1	Function 2	Function 3	Function 4	Function 5	Function 6	Reset	Notes
0	GPIO0	GPIO0	CLK_OUT1	GPIO0	-	-	EMAC_TX_CLK	3	R
1	U0TXD	U0TXD	CLK_OUT3	GPIO1	-	-	EMAC_RXD2	3	-
2	GPIO2	GPIO2	HSPIWP	GPIO2	HS2_DATA0	SD_DATA0	-	2	R
3	U0RXD	U0RXD	CLK_OUT2	GPIO3	-	-	-	3	-
4	GPIO4	GPIO4	HSPIHD	GPIO4	HS2_DATA1	SD_DATA1	EMAC_TX_ER	2	R
5	GPIO5	GPIO5	VSPICS0	GPIO5	HS1_DATA6	-	EMAC_RX_CLK	3	-
6	SD_CLK	SD_CLK	SPICLK	GPIO6	HS1_CLK	U1CTS	-	3	-
7	SD_DATA_0	SD_DATA0	SPIQ	GPIO7	HS1_DATA0	U2RTS	-	3	-
8	SD_DATA_1	SD_DATA1	SPID	GPIO8	HS1_DATA1	U2CTS	-	3	-
9	SD_DATA_2	SD_DATA2	SPIHD	GPIO9	HS1_DATA2	U1RXD	-	3	-
10	SD_DATA_3	SD_DATA3	SPIWP	GPIO10	HS1_DATA3	U1TXD	-	3	-
11	SD_CMD	SD_CMD	SPICS0	GPIO11	HS1_CMD	U1RTS	-	3	-
12	MTDI	MTDI	HSPIQ	GPIO12	HS2_DATA2	SD_DATA2	EMAC_TXD3	2	R
13	MTCK	MTCK	HSPID	GPIO13	HS2_DATA3	SD_DATA3	EMAC_RX_ER	1	R
14	MTMS	MTMS	HSPICLK	GPIO14	HS2_CLK	SD_CLK	EMAC_TXD2	1	R
15	MTDO	MTDO	HSPICS0	GPIO15	HS2_CMD	SD_CMD	EMAC_RXD3	3	R
16	GPIO16	GPIO16	-	GPIO16	HS1_DATA4	U2RXD	EMAC_CLK_OUT	1	-
17	GPIO17	GPIO17	-	GPIO17	HS1_DATA5	U2TXD	EMAC_CLK_180	1	-
18	GPIO18	GPIO18	VSPICLK	GPIO18	HS1_DATA7	-	-	1	-
19	GPIO19	GPIO19	VSPIQ	GPIO19	U0CTS	-	EMAC_TXD0	1	-
21	GPIO21	GPIO21	VSPIHD	GPIO21	-	-	EMAC_TX_EN	1	-
22	GPIO22	GPIO22	VSPIWP	GPIO22	U0RTS	-	EMAC_TXD1	1	-
23	GPIO23	GPIO23	VSPID	GPIO23	HS1_STROBE	-	-	1	-
25	GPIO25	GPIO25	-	GPIO25	-	-	EMAC_RXD0	0	R
26	GPIO26	GPIO26	-	GPIO26	-	-	EMAC_RXD1	0	R
27	GPIO27	GPIO27	-	GPIO27	-	-	EMAC_RX_DV	1	R
32	32K_XP	GPIO32	-	GPIO32	-	-	-	0	R
33	32K_XN	GPIO33	-	GPIO33	-	-	-	0	R
34	VDET_1	GPIO34	-	GPIO34	-	-	-	0	R, I
35	VDET_2	GPIO35	-	GPIO35	-	-	-	0	R, I
36	SENSOR_VP	GPIO36	-	GPIO36	-	-	-	0	R, I
37	SENSOR_CAPP	GPIO37	-	GPIO37	-	-	-	0	R, I
38	SENSOR_CAPN	GPIO38	-	GPIO38	-	-	-	0	R, I
39	SENSOR_VN	GPIO39	-	GPIO39	-	-	-	0	R, I

RTC GPIO Num	GPIO Num	Pad Name	Analog Function			RTC Function	
			1	2	3	Function 1 (FUN_SEL = 0)	Function 2 (FUN_SEL = 3)
0	36	SENSOR_VP	ADC_H	ADC1_CH0	-	RTC_GPIO0	-
1	37	SENSOR_CAPP	ADC_H	ADC1_CH1	-	RTC_GPIO1	-
2	38	SENSOR_CAPN	ADC_H	ADC1_CH2	-	RTC_GPIO2	-
3	39	SENSOR_VN	ADC_H	ADC1_CH3	-	RTC_GPIO3	-
4	34	VDET_1	-	ADC1_CH6	-	RTC_GPIO4	-
5	35	VDET_2	-	ADC1_CH7	-	RTC_GPIO5	-
6	25	GPIO25	DAC_1	ADC2_CH8	-	RTC_GPIO6	-
7	26	GPIO26	DAC_2	ADC2_CH9	-	RTC_GPIO7	-
8	33	32K_XN	XTAL_32K_N	ADC1_CH5	TOUCH8	RTC_GPIO8	-
9	32	32K_XP	XTAL_32K_P	ADC1_CH4	TOUCH9	RTC_GPIO9	-
10	4	GPIO4	-	ADC2_CH0	TOUCH0	RTC_GPIO10	I2C_SCL
11	0	GPIO0	-	ADC2_CH1	TOUCH1	RTC_GPIO11	I2C_SDA
12	2	GPIO2	-	ADC2_CH2	TOUCH2	RTC_GPIO12	I2C_SCL
13	15	MTDO	-	ADC2_CH3	TOUCH3	RTC_GPIO13	I2C_SDA
14	13	MTCK	-	ADC2_CH4	TOUCH4	RTC_GPIO14	-
15	12	MTDI	-	ADC2_CH5	TOUCH5	RTC_GPIO15	-
16	14	MTMS	-	ADC2_CH6	TOUCH6	RTC_GPIO16	-
17	27	GPIO27	-	ADC2_CH7	TOUCH7	RTC_GPIO17	-

LROBRUYA

ESP32 Arduino IDE

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

Prerequisites: Arduino IDE Installed

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

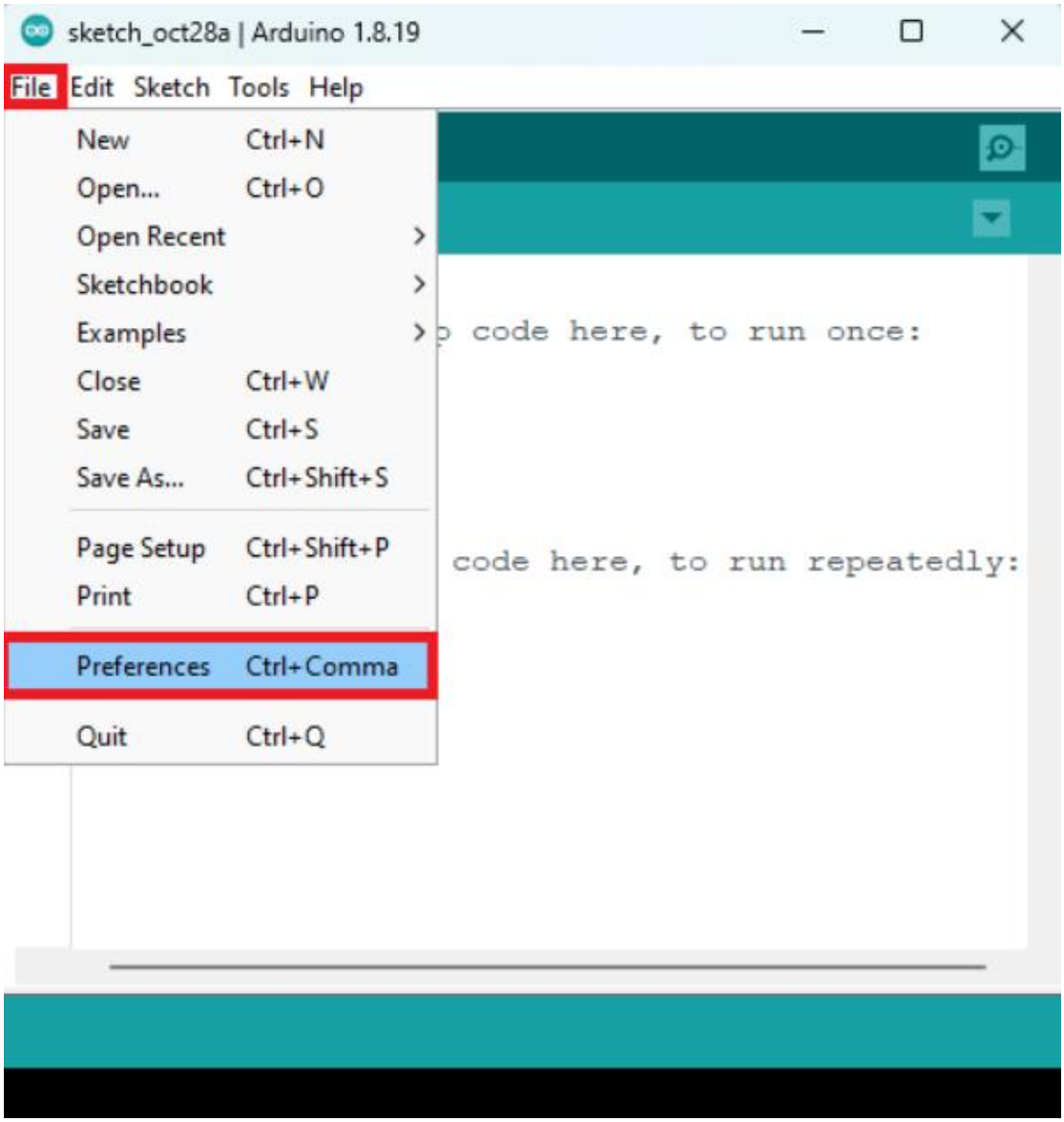
You can download and install Arduino IDE by clicking on the following link: arduino.cc/en/Main/Software

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

Installing ESP32 Add-on in Arduino IDE

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

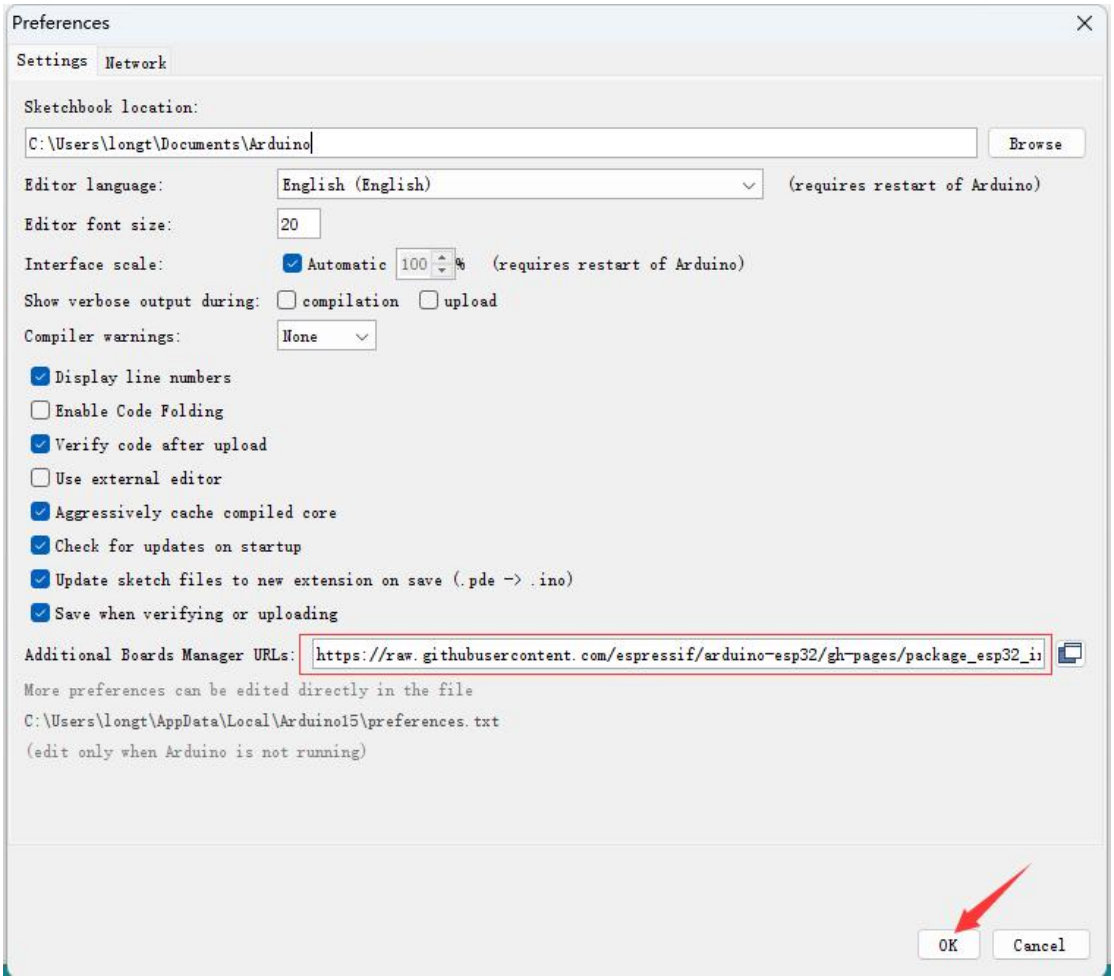
1.In your Arduino IDE, go to File> Preferences



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

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

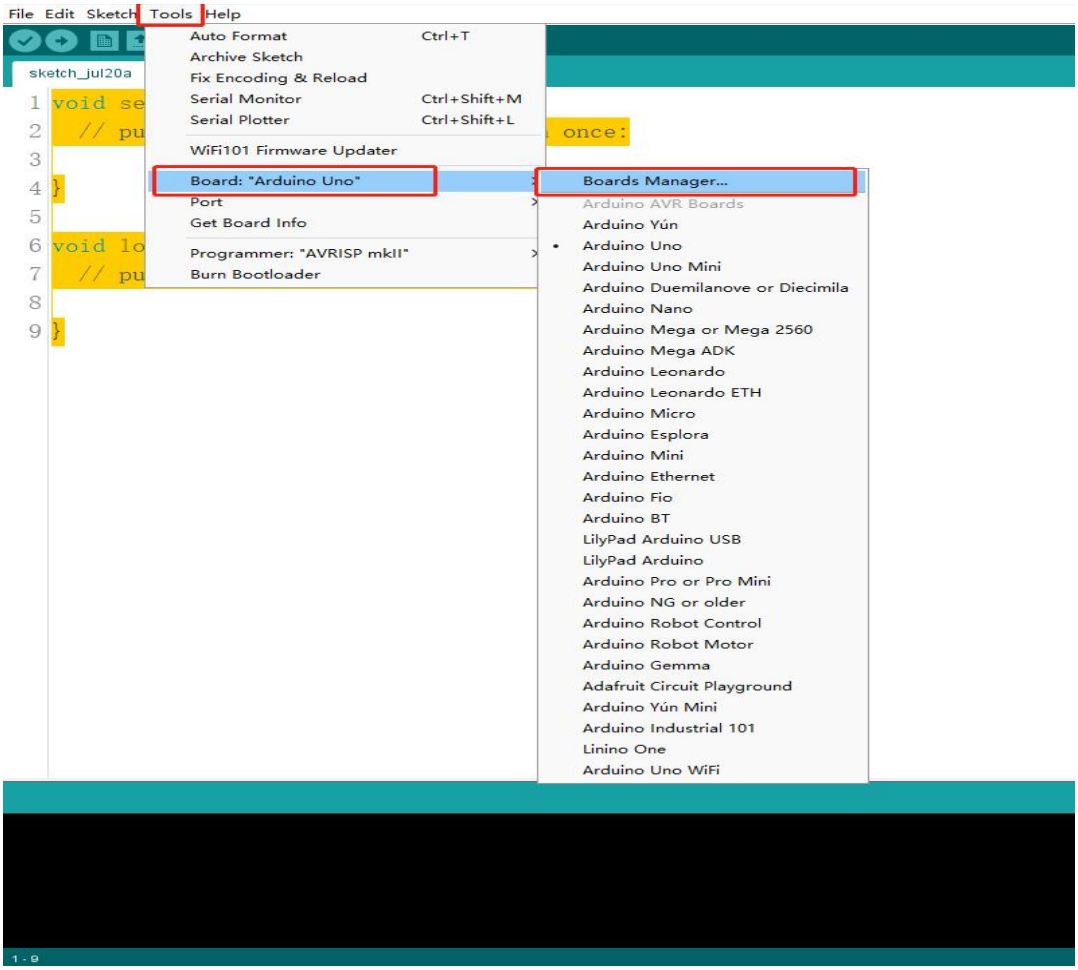
Then, click the “OK” button:



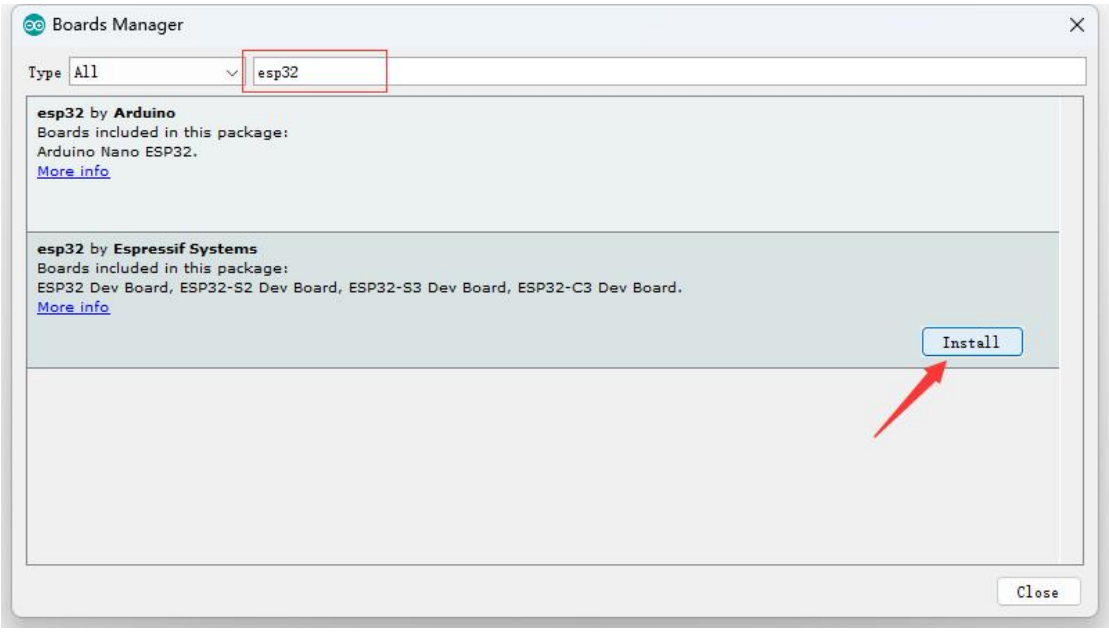
Note: if you already have the ESP8266 boards URL, you can separate the URLs with a comma as follows:

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

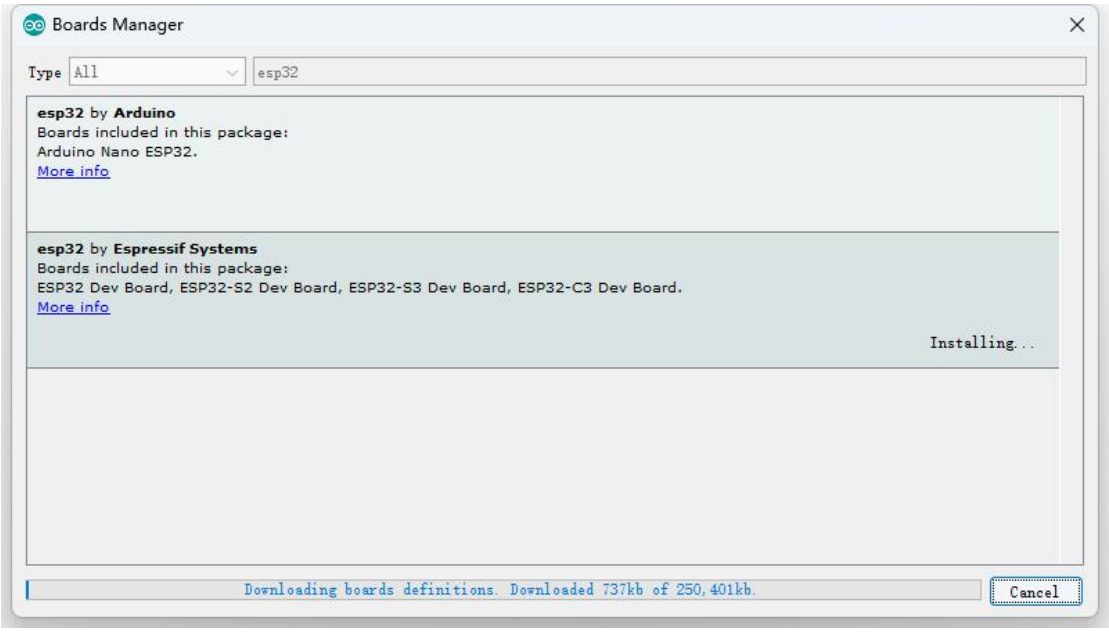
3.Open the Boards Manager. Go to **Tools** > **Board** > **Boards Manager…**



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



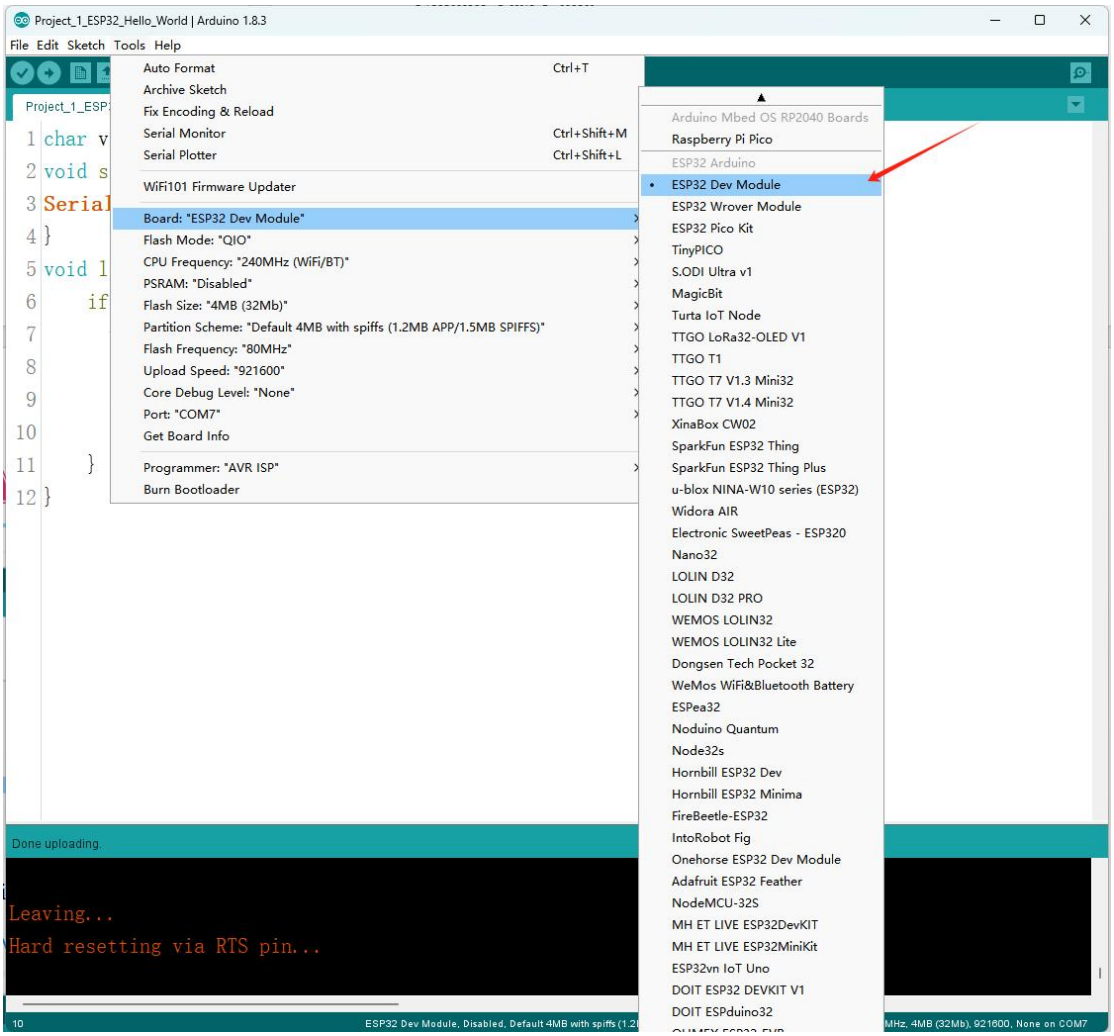
5.That's it. It should be installed after a few seconds.



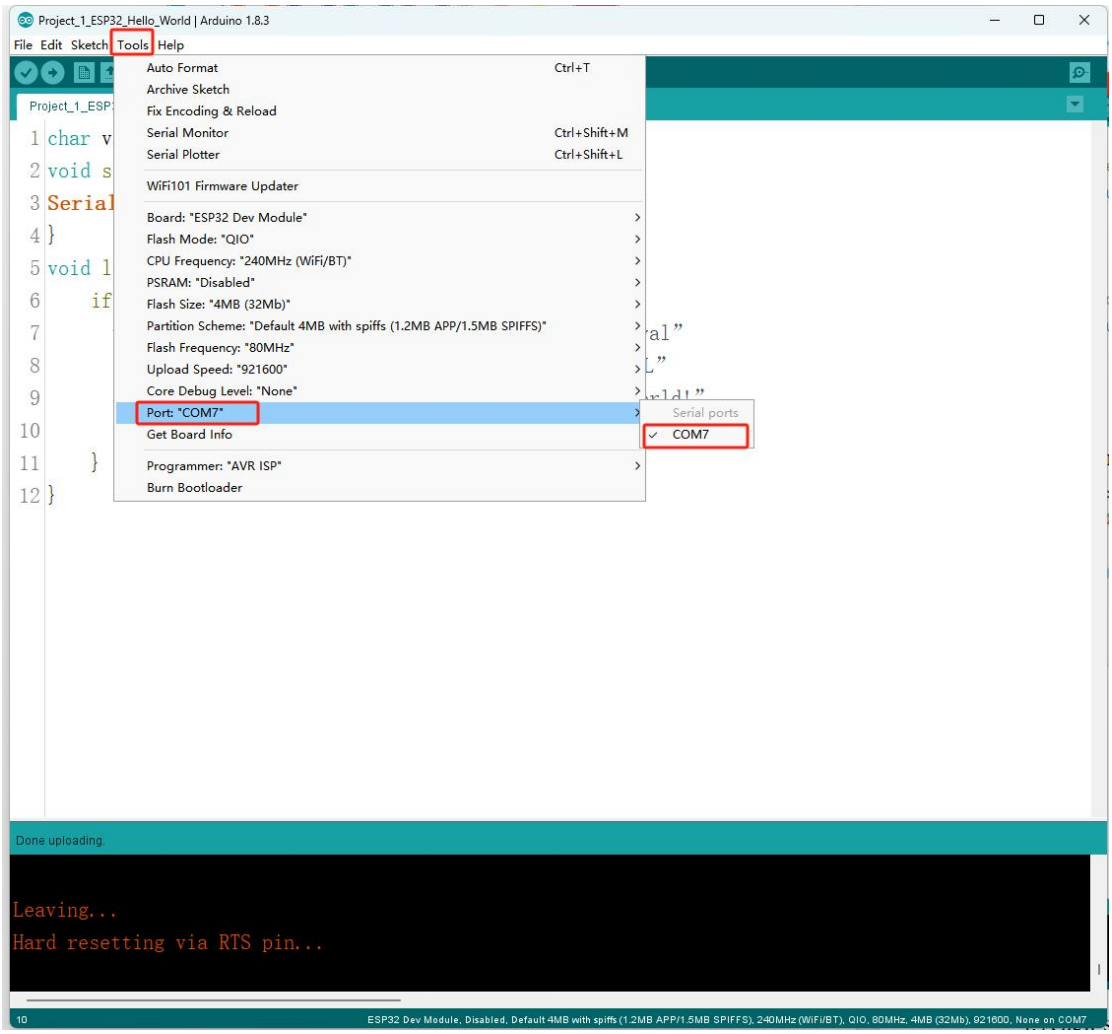
Upload Test Code

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

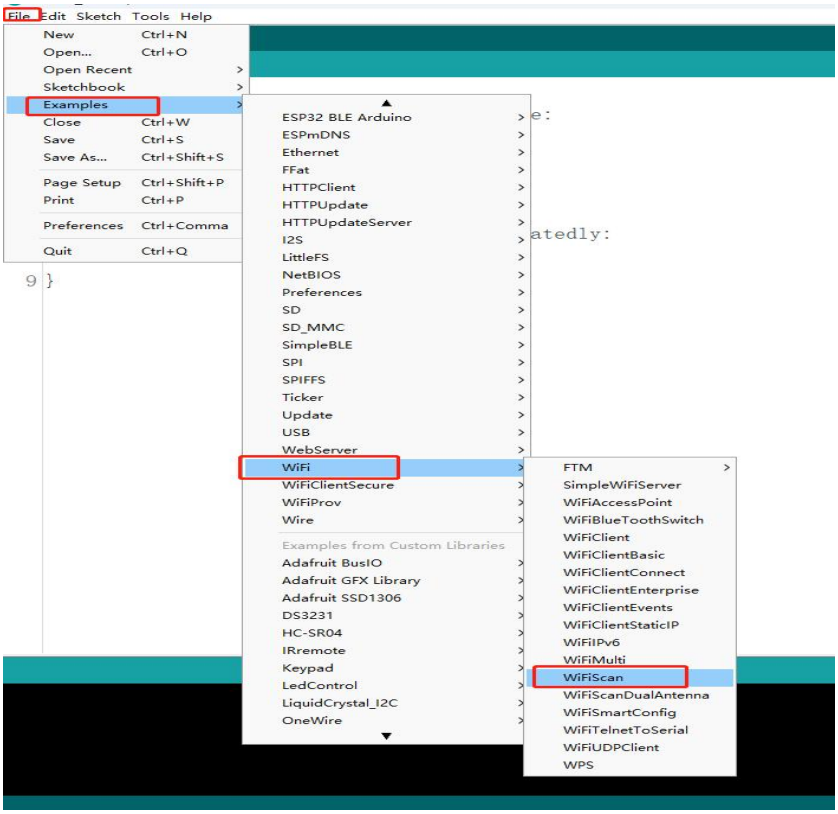
1. Select your Board in **Tools > Board** menu (in my case it's the **ESP32 Dev Module**)



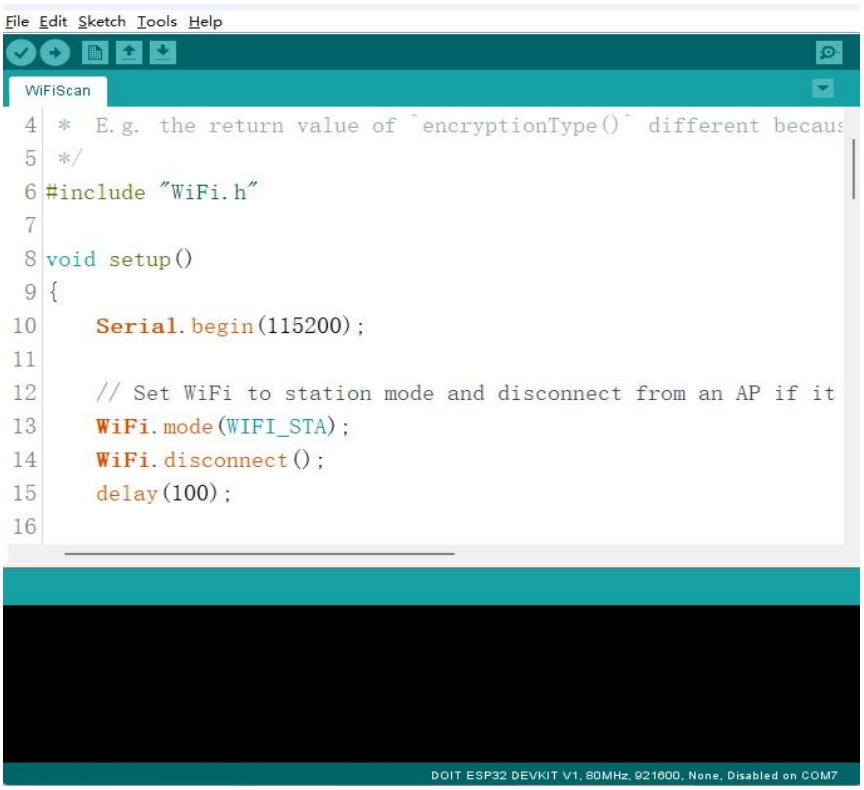
2. Select the Port (if you don’t see the COM Port in your Arduino IDE, you need to install the CP210x USB to UART Bridge VCP Drivers):



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

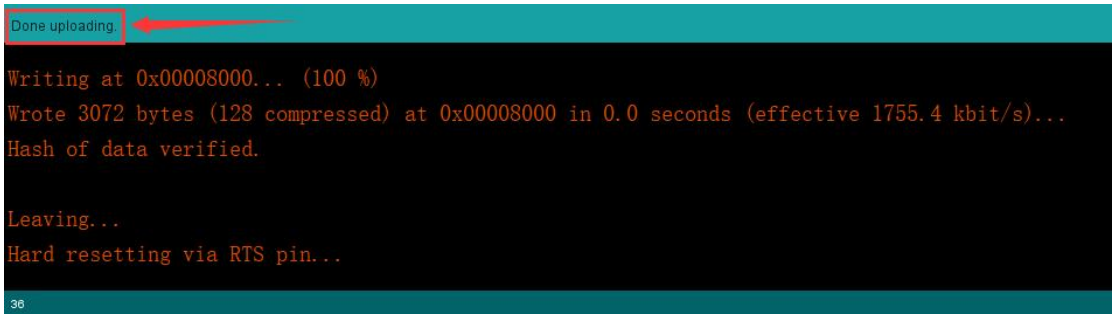


4. A new sketch opens in your Arduino IDE:



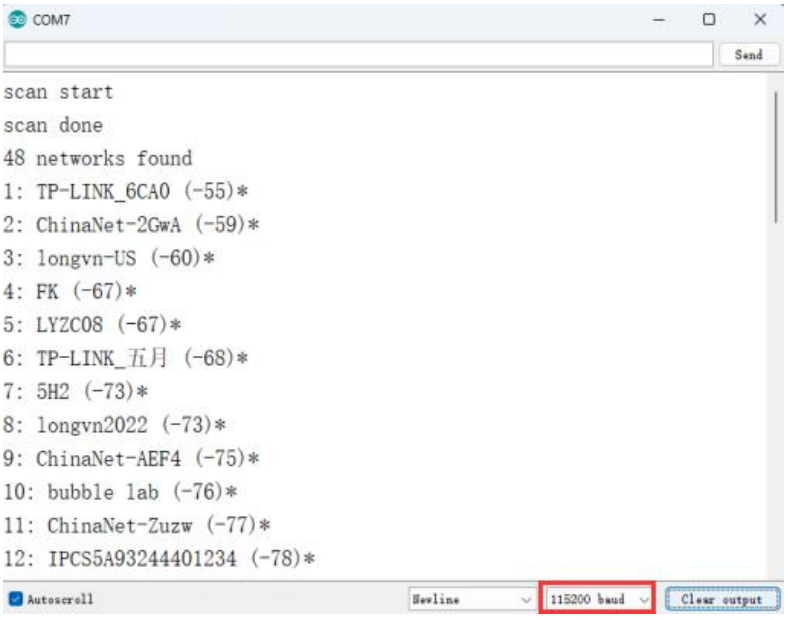
5. Press the **Upload** button in the Arduino IDE. Wait a few seconds while the code compiles and uploads to your board.

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



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

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



Troubleshooting

If you try to upload a new sketch to your ESP32 and you get this error message "A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header". It means that your ESP32 is not in flashing/uploading mode.

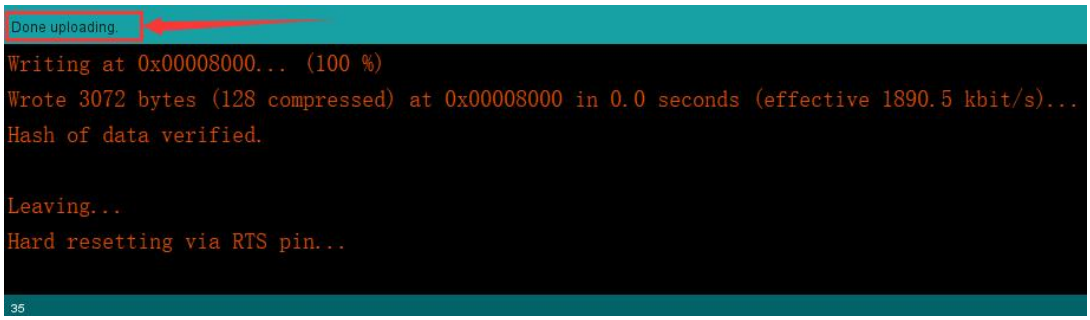
Having the right board name and COM port selected, follow these steps:

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

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

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

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



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

Note:You'll also have to repeat that button sequence every time you want to upload a new sketch.

Project 1 ESP32 Hello World

For ESP32 beginners, we'll start with some simple things. In this project, you just need an ESP32 board, a USB cable and a computer to complete "Hello World!" Project. It is not only a communication test for ESP32 mainboard and computer, but also a primary project for ESP32.

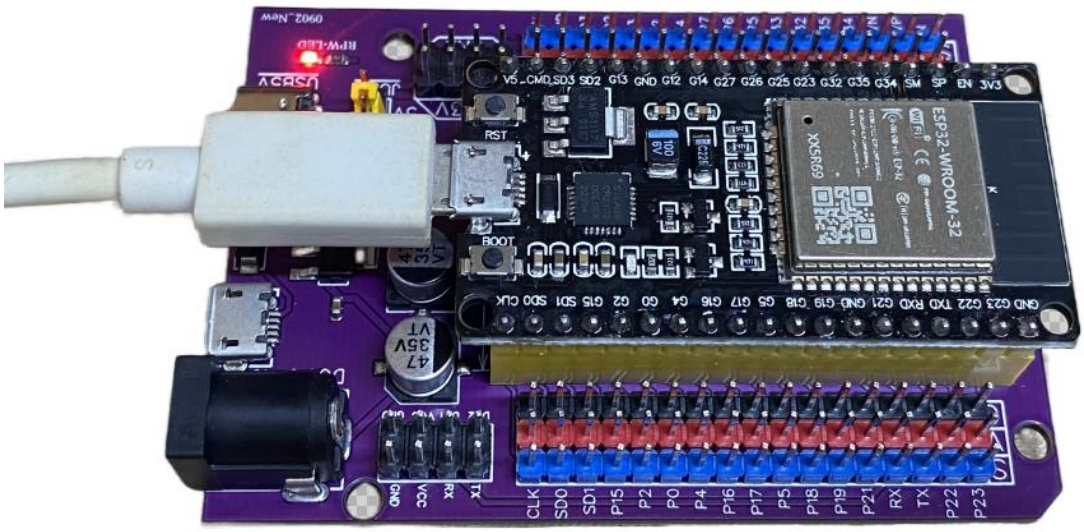
Prerequisites

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

Installing ESP32 Add-on in Arduino IDE

Project Example

In this project, we use a USB cable to connect the ESP32 to the computer.



Parts Required

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

- ESP32-DevKitC V4
- 38P Expansion board
- USB cable

Code

Open the code Project_1_ESP32_Hello_World.ino in arduino IDE

```
char val; // defines variable “val”
void setup() {
Serial.begin(115200); //sets baudrate to 115200
}
void loop() {
if (Serial.available() > 0) {
val=Serial.read(); //reads symbols assigns to “val”
if (val == ‘L’) { //check input for the letter “L”
Serial.println(“Hello World!”); //shows “Hello World!”
}
}
}
```

Uploading the Code

Before clicking the upload button, go to **Tools** > **Board**, and select the board :ESP32 Dev Module board.

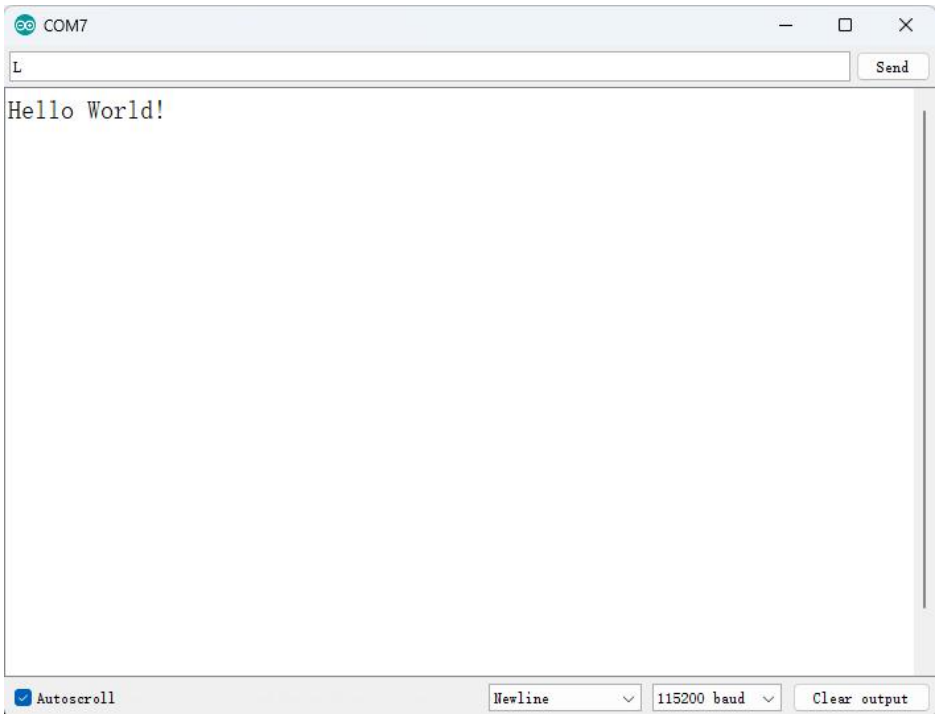
Go to **Tools** > **Port** and select the COM port the ESP32 is connected to.

Then, press the upload button and wait for the "**Done uploading**" message.

Note:If you see a lot of dots (connecting…__…__) on the debugging window and the "Failed to connect to ESP32: Timed out waiting for packet header" message, that means you need to press the ESP32 on-board BOOT button after the dots start appearing.

Demonstration

Click to open serial port monitor, input an " L", PC will receive the information from Arduino: Hello World!



LROBRUYA

Project 2 ESP32 Inputs Outputs

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

Prerequisites

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

Installing ESP32 Add-on in Arduino IDE

ESP32 Control Digital Outputs

First, you need set the GPIO you want to control as an OUTPUT.

Use the pinMode() function as follows:

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

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

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

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

Learn more about the ESP32 GPIOs: ESP32 GPIO Reference Guide

ESP32 Read Digital Inputs

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

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

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

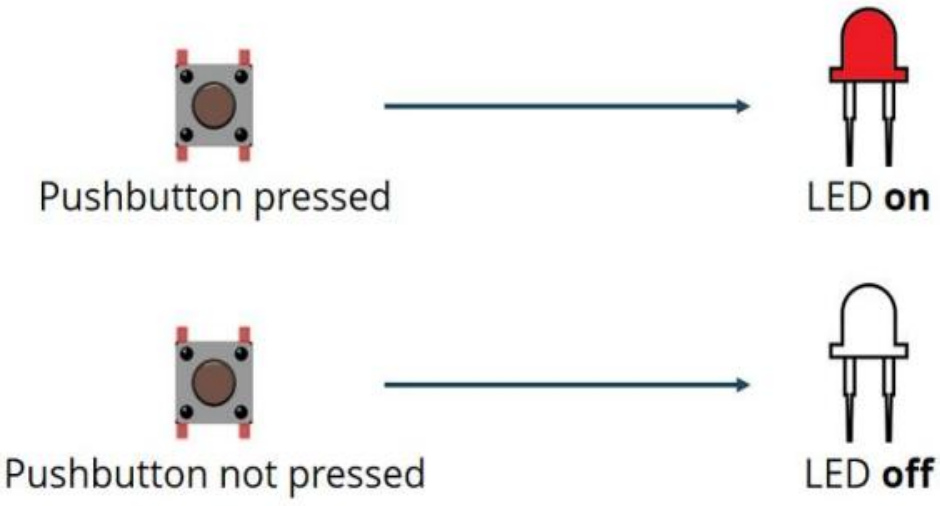
```
digitalRead(GPIO);
```

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

Learn more about the ESP32 GPIOs: ESP32 GPIO Reference Guide

Project Example

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



Parts Required

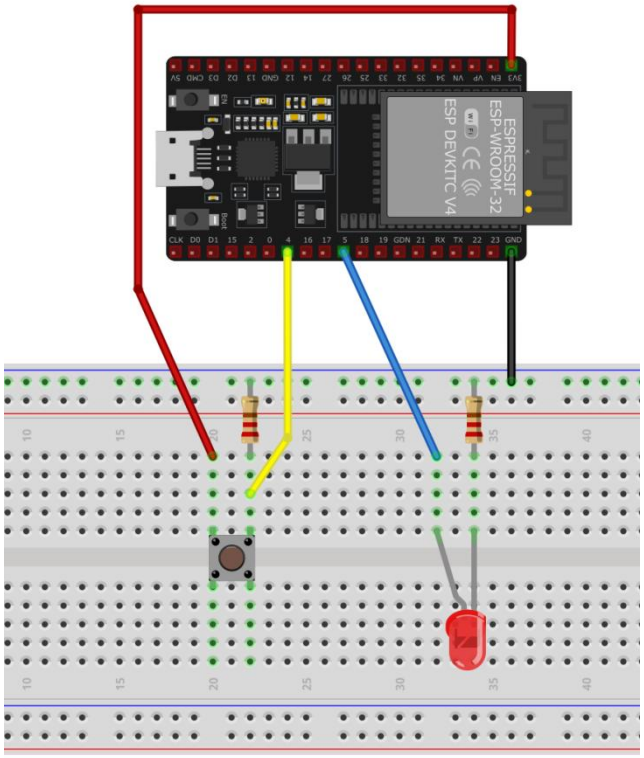
Here's a list of the parts to you need to build the circuit:

- ESP32-DevKitC V4
- 38P Expansion board
- 5 mm LED
- 220 Ohm resistor
- Push button
- 10k Ohm resistor
- Breadboard
- Jumper wires

Schematic Diagram

Before proceeding, you need to assemble a circuit with an LED and a push button.

We'll connect the LED to GPIO 5 and the push button to GPIO 4.



Code

Open the code Project_2_ESP32_Inputs_Outputs.ino in arduino IDE

```
// set pin numbers
const int buttonPin = 4; // the number of the pushbutton pin
const int ledPin = 5; // the number of the LED pin
// variable for storing the pushbutton status
int buttonState = 0;
void setup() {
Serial.begin(115200);
pinMode(buttonPin, INPUT); // initialize the pushbutton pin
as an input
pinMode(ledPin, OUTPUT); // initialize the LED pin as an output
}
void loop() {
// read the state of the pushbutton value
buttonState = digitalRead(buttonPin);
Serial.println(buttonState); // check if the pushbutton is
pressed.
// if it is, the buttonState is HIGH
if (buttonState == HIGH) {
// turn LED on
digitalWrite(ledPin, HIGH);
} else {
// turn LED off
digitalWrite(ledPin, LOW);
}
}
```

How the Code Works

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

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

The Button is connected to GPIO 4 and the LED is connected to GPIO 5.

When using the Arduino IDE with the ESP32, 4 corresponds to GPIO 4

and 5 corresponds to GPIO 5.

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

```
int buttonState = 0;
```

In the setup(), you initialize the button as an INPUT, and the LED as an OUTPUT.

For that, you use the pinMode() function that accepts the pin you are referring to, and the mode: INPUT or OUTPUT.

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

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

In the next line, you read the button state and save it in the buttonState variable.

As we've seen previously, you use the digitalRead() function.

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

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

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

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

```
else
```

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

Uploading the Code

Before clicking the upload button, go to **Tools** > **Board**, and select the board :ESP32 Dev Module board.

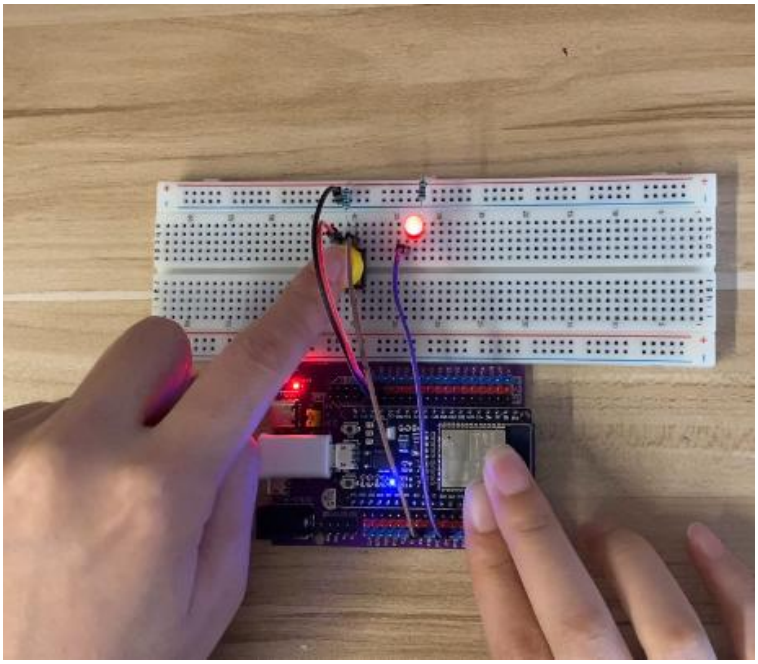
Go to **Tools** > **Port** and select the COM port the ESP32 is connected to.

Then, press the upload button and wait for the "**Done uploading**" message.

Note:If you see a lot of dots (connecting…__…__) on the debugging window and the "Failed to connect to ESP32: Timed out waiting for packet header" message, that means you need to press the ESP32 on-board BOOT button after the dots start appearing.

Demonstration

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



And turn off when you release it.

Project 3 ESP32 PWM(Analog Output)

In this project we'll show you how to generate PWM signals with the ESP32 using Arduino IDE. As an example we'll build a simple circuit that dims an LED using the LED PWM controller of the ESP32.

ESP32 LED PWM Controller

The ESP32 has a LED PWM controller with 16 independent channels that can be configured to generate PWM signals with different properties. Here's the steps you'll have to follow to dim an LED with PWM using the Arduino IDE:

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

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

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

4. Next, you need to specify to which GPIO or GPIOs the signal will appear upon.

For that you'll use the following function:

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

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

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

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

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

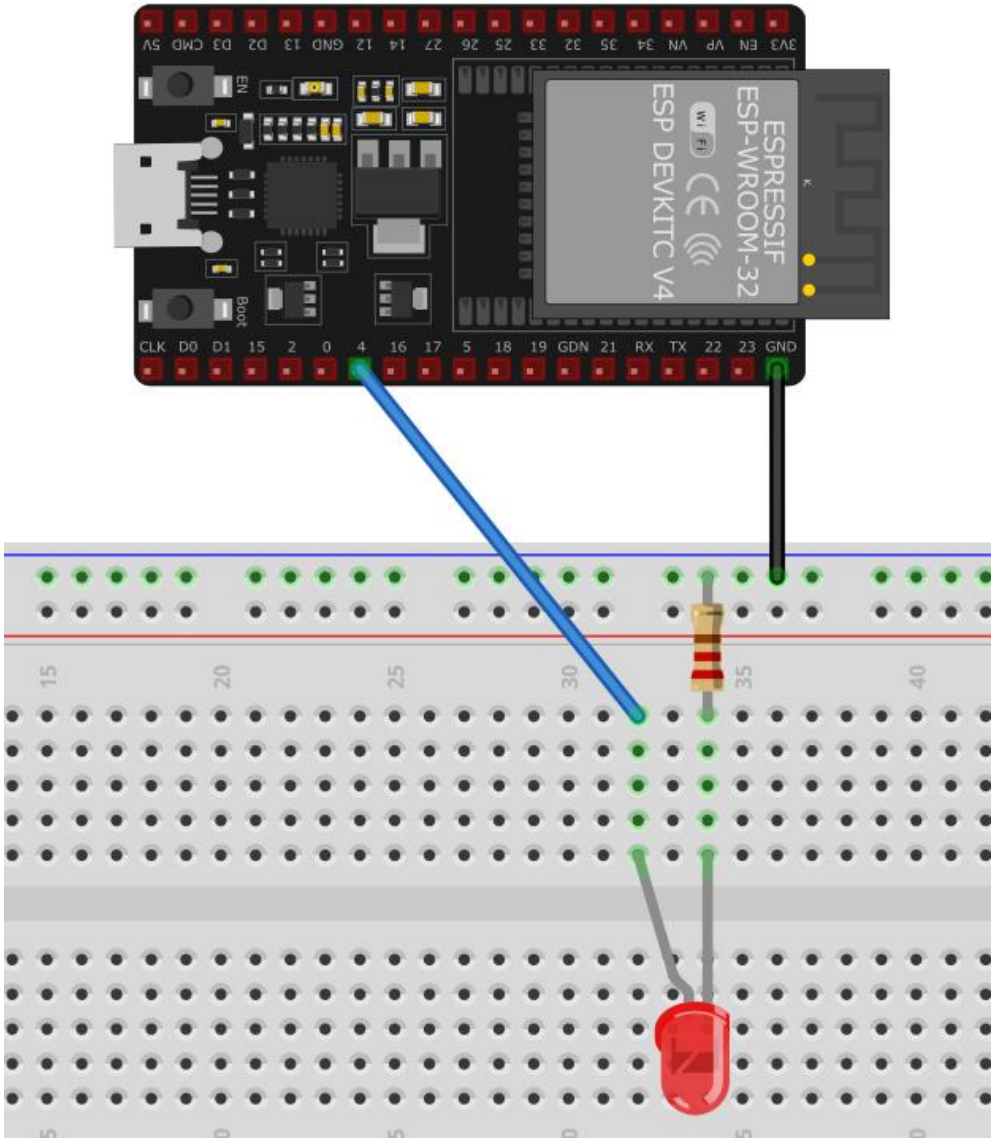
Parts Required

To follow this tutorial you need these parts:

- ESP32-DevKitC V4
- 38P Expansion board
- 220 Ohm resistor
- Breadboard
- Jumper wires

Schematic

Wire an LED to your ESP32 as in the following schematic diagram. The LED should be connected to GPIO 4.



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.

Code

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:(If you have already done this step, you can skip to the next step.)

Installing ESP32 Add-on in Arduino IDE

Open the code Project_3_ESP32_PWM.ino in arduino IDE

```
// the number of the LED pin
#define ledPin 4; // 4 corresponds to GPIO4
// setting PWM properties

#define ledChannel 0; //PWM Channel 0

#define resolution 8; //8-bit resolution, we can control the
LED brightness using a value from 0 to 255
#define freq 5000; // PWM signal frequency 5000 Hz
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);
}}
```

How the Code Works

You start by defining the pin the LED is attached to. In this case the LED is attached to GPIO 4.

```
#define ledPin 4; // 4 corresponds to GPIO4
```

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

```
#define ledChannel 0;
#define resolution 8;
#define freq 5000;
```

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 4. The channel that generates the signal is the ledChannel, that corresponds to channel 0.

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

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

```
for(int dutyCycle = 0; dutyCycle <= 255; dutyCycle++){
```

```
// changing the LED brightness with PWM
ledcWrite(ledChannel, dutyCycle);
delay(15); }
```

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

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

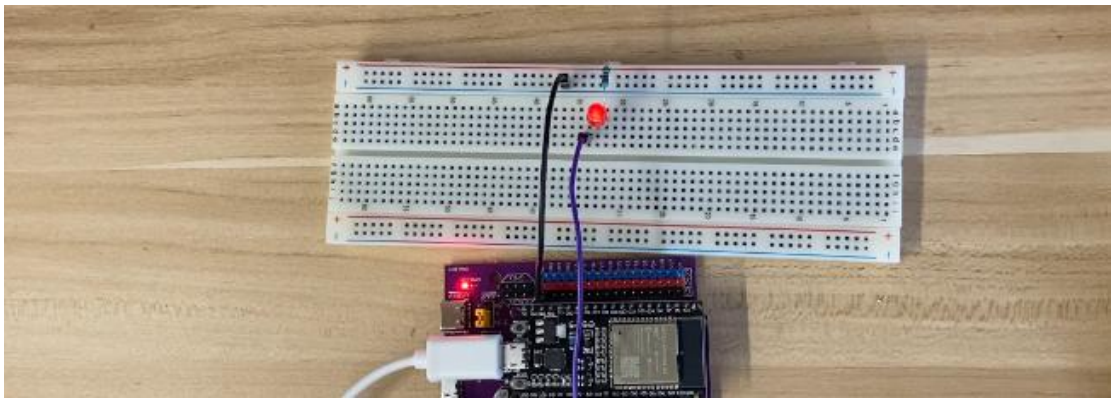
To set the brightness of the LED, you just need to use the ledcWrite() function that accepts as arguments the channel that is generating the signal, and the duty cycle.

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

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

Testing the Example

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

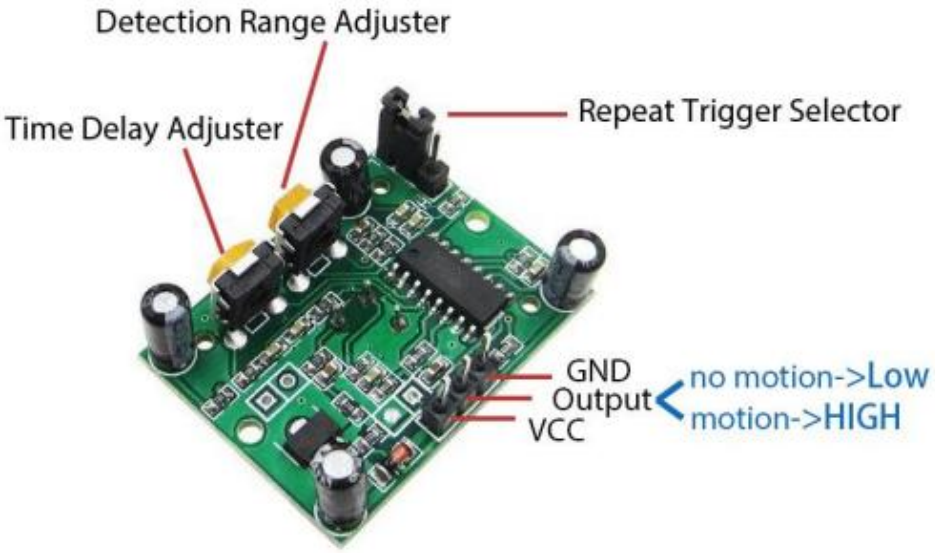


LROBRUYA

Project 4 ESP32 PIR Motion Sensor

This project shows how to detect motion with the ESP32 using a PIR motion sensor.The buzzer will sound an alarm when motion is detected, and stop the alarm when no motion is detected for a preset time (such as 4 seconds).

How HC-SR501 Motion Sensor Works



The working principle of HC-SR501 sensor is based on the change of the infrared radiation on the moving object.To be detected by the HC-SR501 sensor, the object must meet two requirements:

- The object is emitting the infrared way.

- The object is moving or shaking

So: If an object is emitting the infrared ray but not moving (e.g, a person stands still without moving), it is not detected by the sensor.

If an object is moving but not emitting the infrared ray (e.g, robot or vehicle), it is NOT detected by the sensor.

Introducing Timers

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

The delay() function

You should be familiar with the delay() function as it is widely used. This function is pretty straightforward to use. It accepts a single int number as an argument.

This number represents the time in milliseconds the program has to wait until moving on to the next line of code.

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

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

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

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

The millis() function

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

```
millis()
```

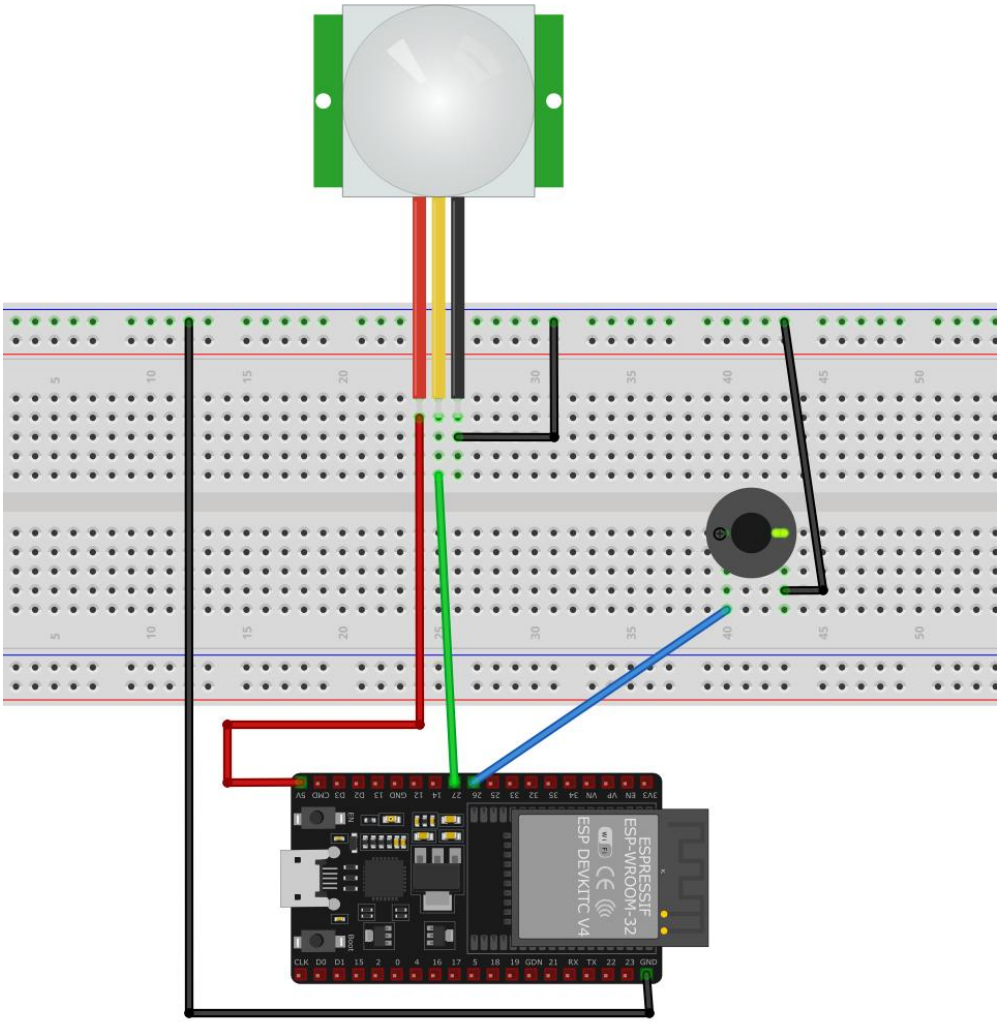
Why is that function useful? Because by using some math, you can easily verify how much time has passed without blocking your code.

Parts Required

To follow this tutorial you need the following parts

- ESP32-DevKitC V4
- 38P Expansion board
- PIR motion sensor (HC-SR501)
- Active Buzzer
- Jumper wires
- Breadboard

Schematic



Note:The working voltage of HC-SR501 is 5V.

Code

Before proceeding with this tutorial you should have the ESP32 add-on installed in your Arduino IDE. Follow one of the following tutorials to install the ESP32 on the Arduino IDE, if you haven't already.(If you havealready done this step, you can skip to the next step.)

Installing ESP32 Add-on in Arduino IDE

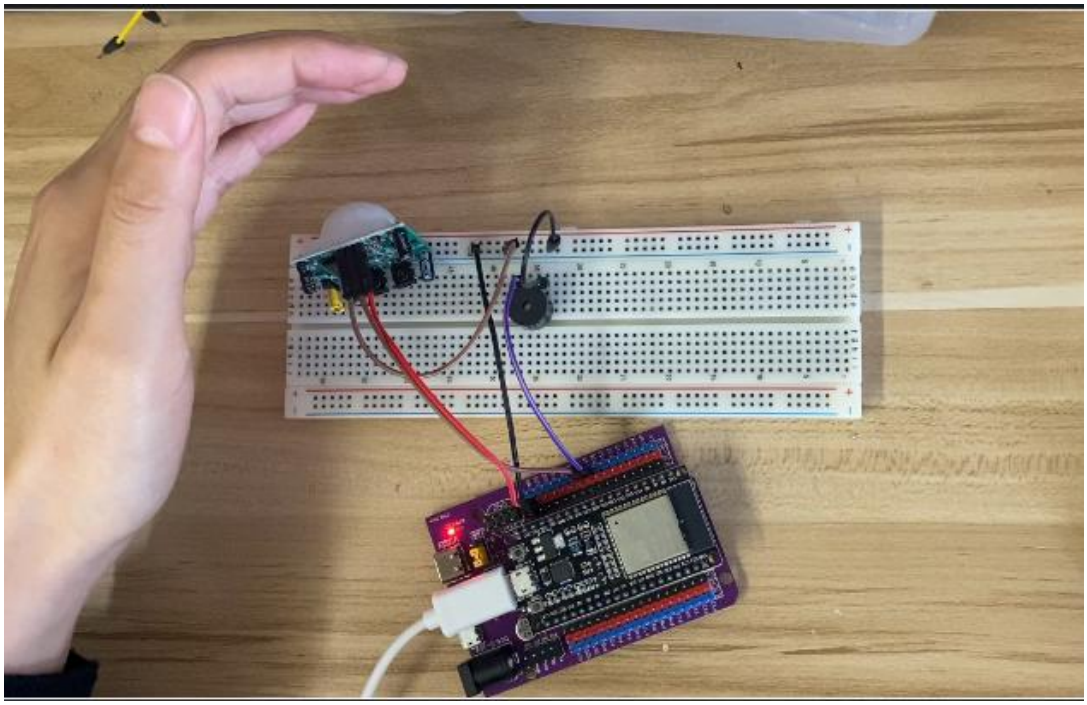
Open the code Project_4_ESP32_PIR_Motion_Sensor.ino in arduino IDE.

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 buzzer should turn on, and the message is printed in the Serial Monitor saying “Motion detected!Buzzer alarm”. After 4 seconds the buzzer should turn off.



Project 5 ESP32 Switch Web Server

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

Project Overview

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

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

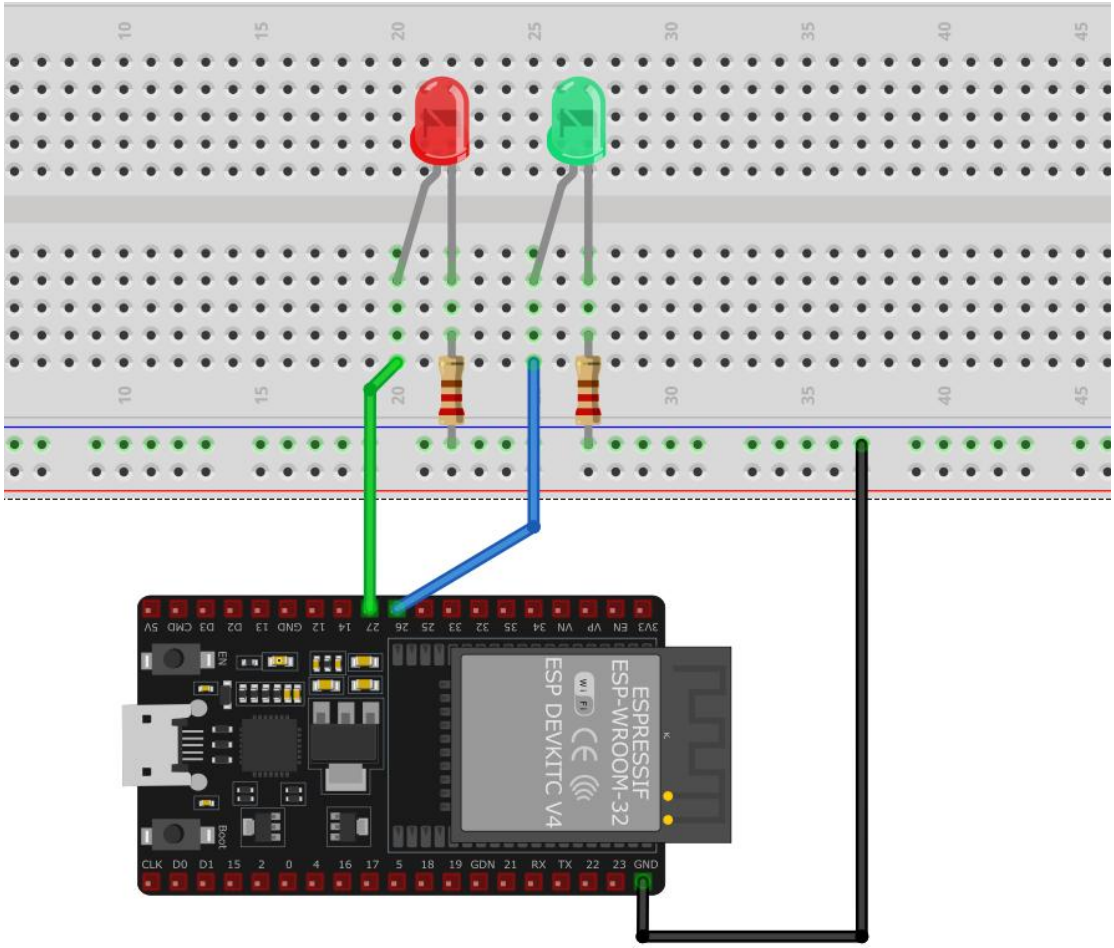
Parts Required

For this tutorial you'll need the following parts:

- ESP32-DevKitC V4
- 38P Expansion board
- 2x 5mm LED
- 2x 200 Ohm resistor
- Breadboard
- Jumper wires

Schematic

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



Code

Here we provide the code that creates the ESP32 web server. Open the code Project_5_ESP32_Switch _Web_Server.ino in arduino IDE, but don’t upload it yet. You need to make some changes to make it work for you.

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:(If you have already done this step, you can skip to the next step.)

Installing ESP32 Add-on in Arduino IDE

Setting Your Network Credentials

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

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

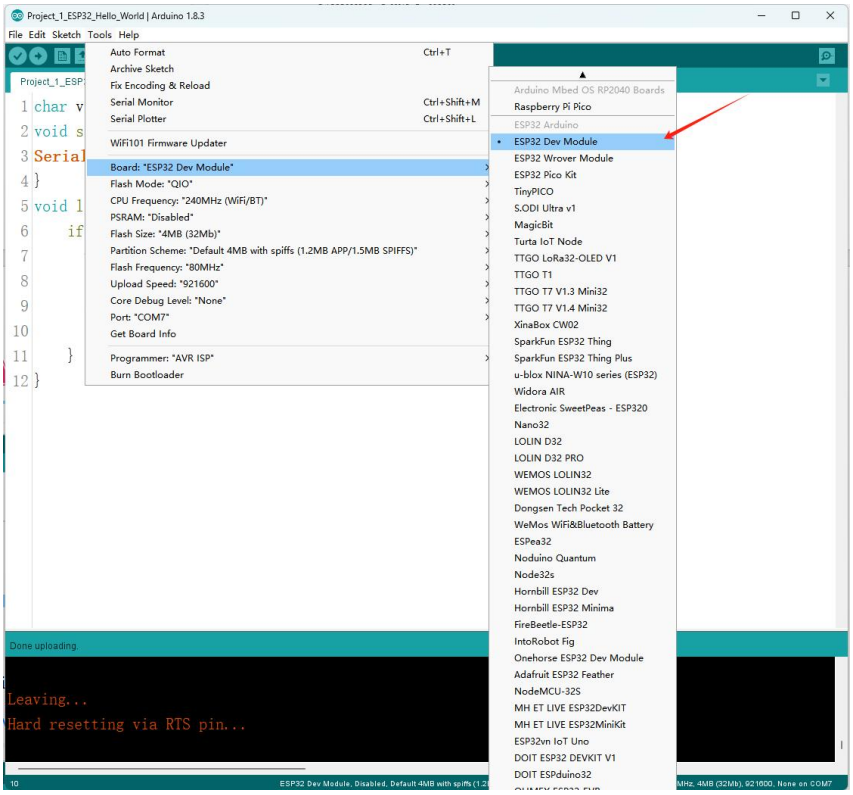
Uploading the Code

Now, you can upload the code and and the web server will work straight away.

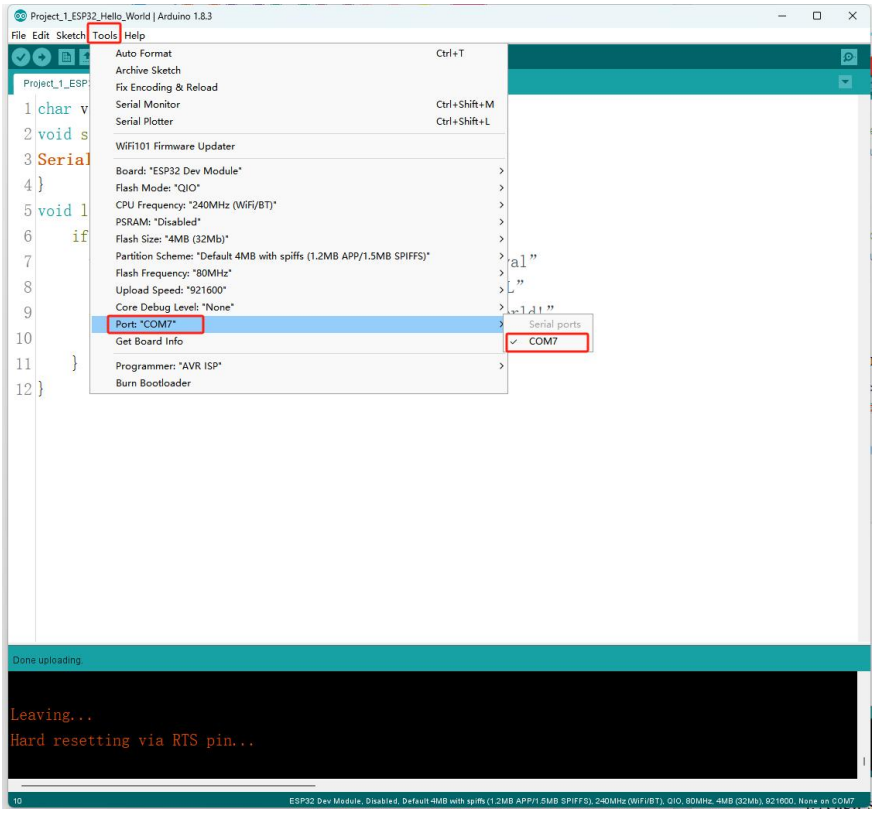
Follow the next steps to upload code to the ESP32:

1) Plug your ESP32 board in your computer;

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

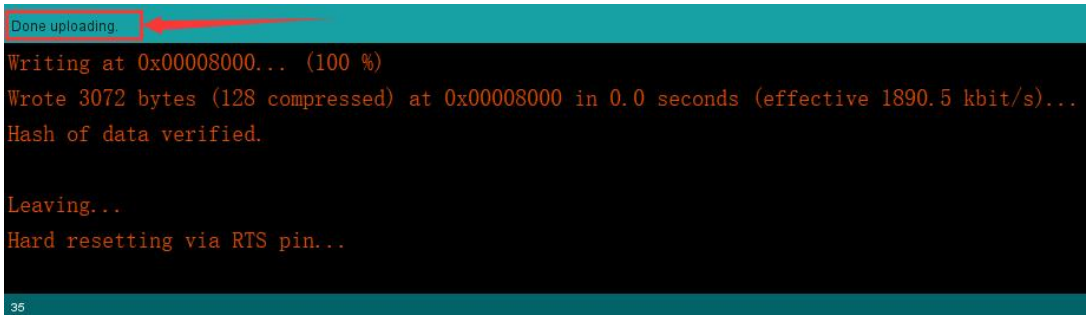


3) Select the COM port in **Tools > Port**.



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

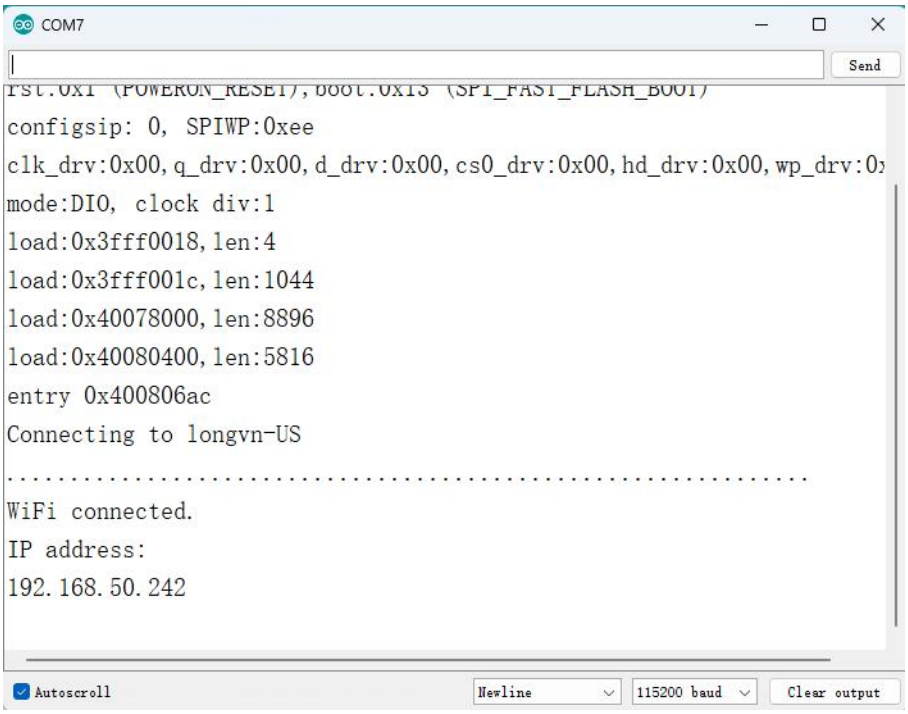
5) Wait for the "**Done uploading**" message.



Finding the ESP IP Address

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

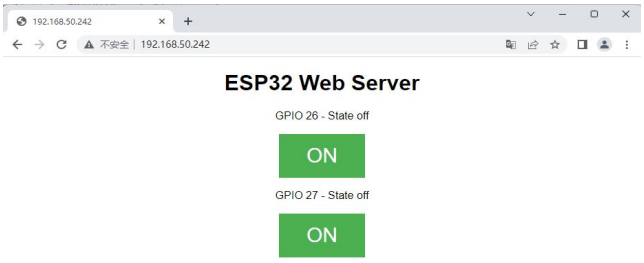
Press the ESP32 EN button (reset). The ESP32 connects to Wi-Fi, and outputs the ESP IP address on the Serial Monitor. Copy that IP address, because you need it to access the ESP32 web server.



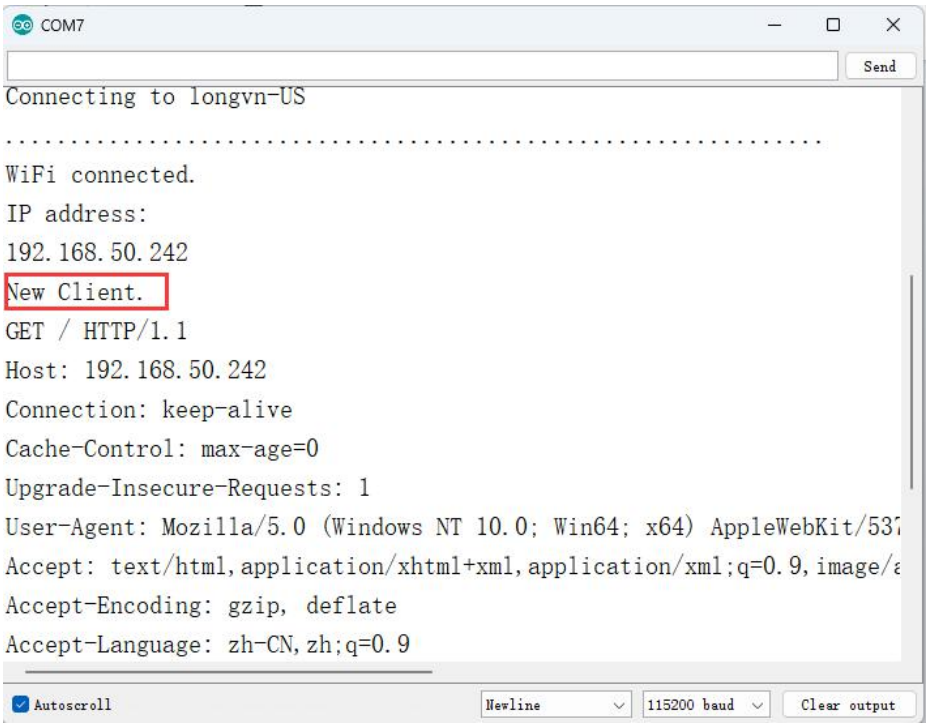
Accessing the Web Server

To access the web server, open your browser, paste the ESP32 IP address, and you'll see the following page.

Note: Your browser and ESP32 should be connected to the same LAN.



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

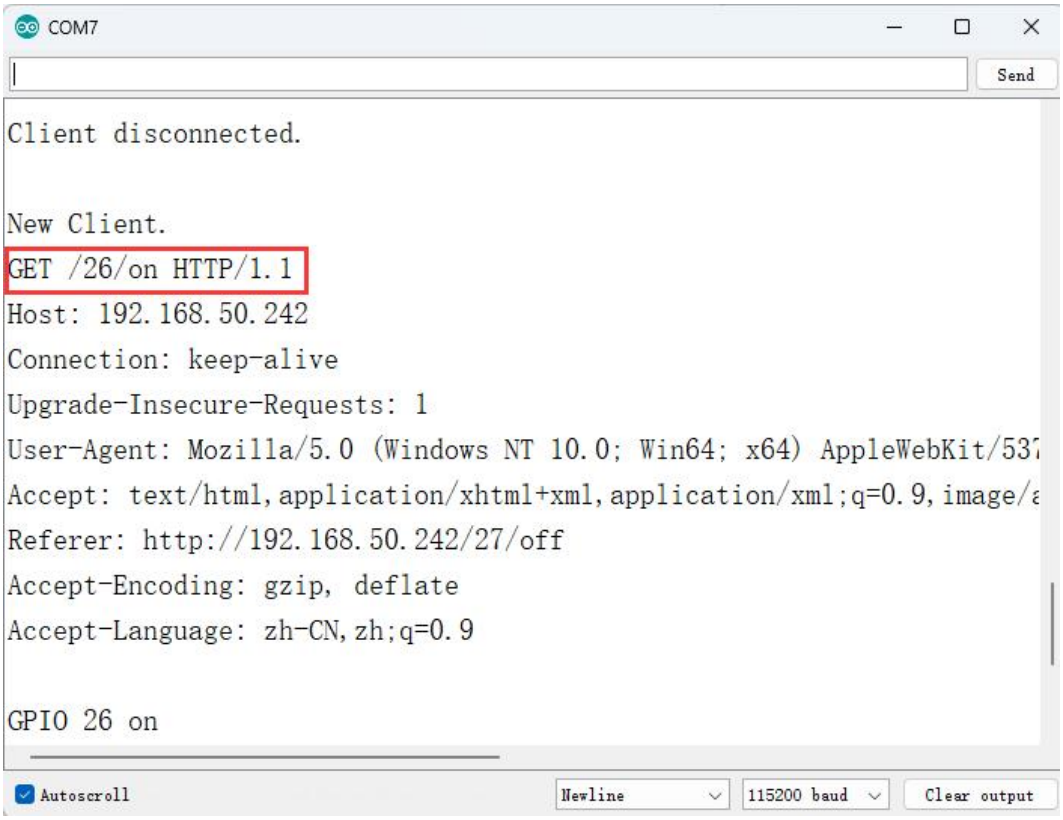


You can also see other information about the HTTP request.

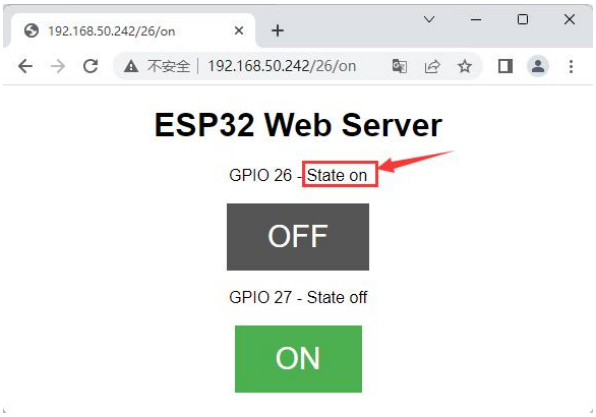
Demonstration

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

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



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



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

How the Code Works

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

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

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

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

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

Then, you set your web server to port 80.

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

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

```
String header;
```

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

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

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

```
const int output26 = 26;
```

```
const int output27 = 27;
```

setup()

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

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

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

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

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

```
// Connect to Wi-Fi network with SSID and password
Serial.print("Connecting to ");
Serial.println(ssid);
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
```

```
delay(500);
Serial.print(".");
}
// Print local IP address and start web server
Serial.println("");
Serial.println("WiFi connected.");
Serial.println("IP address: ");
Serial.println(WiFi.localIP());
server.begin();
```

loop()

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

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

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

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

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

```
output27State = "off";
digitalWrite(output27, LOW);}
```

For example, if you've press the GPIO 26 ON button, the ESP32 receives a request on the **/26/ON URL** (we can see that that information on the HTTP header on the Serial Monitor). So, we can check if the header contains the expression **GET /26/on**. If it contains, we change the output26state variable to ON, and the ESP32 turns the LED on.

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

Displaying the HTML web page

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

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

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

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

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

```
client.println("<head><meta name=\"viewport\"
content=\"width=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("<h1>ESP32 Web Server</h1>");
```

Displaying the Buttons and Corresponding State

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

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

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

```
if (output26State=="off") {
```

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

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

Project 6 ESP32 Relay Web Server

Using a relay with the ESP32 is a great way to control AC household appliances remotely. This tutorial explains how to control a relay module with the ESP32.

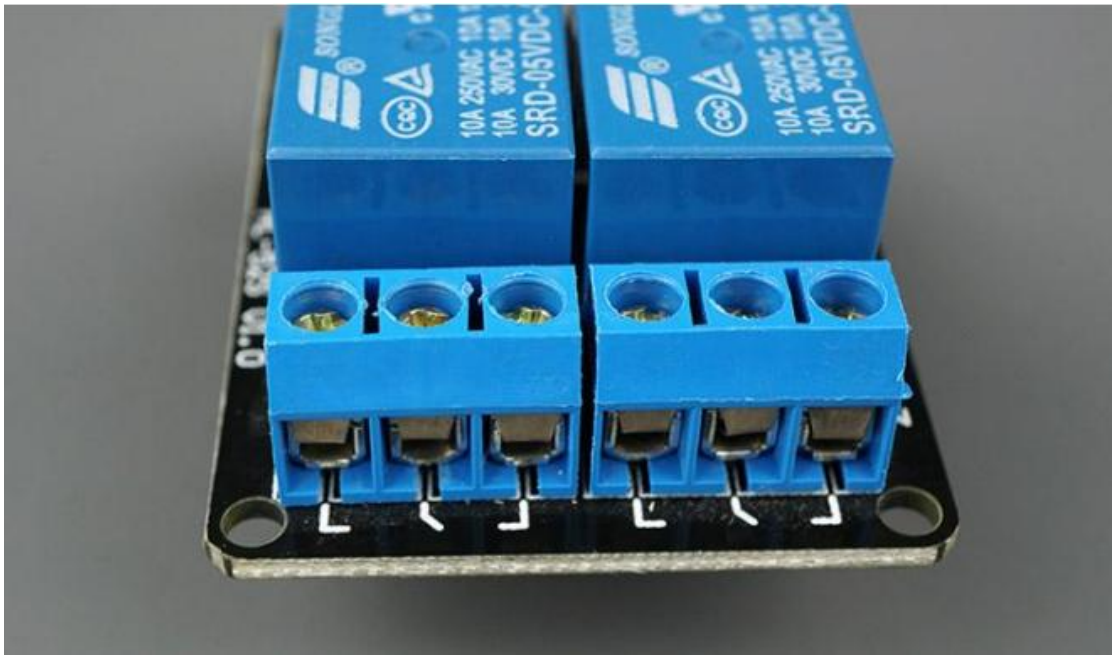
We'll take a look at how a relay module works, how to connect the relay to the ESP32 and build a web server to control a relay remotely.

Introducing Relays

A relay is an electrically operated switch and like any other switch, it that can be turned on or off, letting the current go through or not. It can be controlled with low voltages, like the 3.3V provided by the ESP32 GPIOs and allows us to control high voltages like 12V, 24V or mains voltage (230V in Europe and 120V in the US).

On the left side, there are two sets of three sockets to connect high voltages, and the pins on the right side (low-voltage) connect to the ESP32 GPIOs.

Mains Voltage Connections



The relay module shown in the previous photo has two connectors, each with three sockets: common (COM), Normally Closed (NC), and Normally Open (NO).

- COM: connect the current you want to control (mains voltage).
- NC (Normally Closed): the normally closed configuration is used when you want the relay to be closed by default. The NC are COM pins are connected, meaning the current is flowing unless you send a signal from the ESP32 to the relay module to open the circuit and stop

the current flow.

- NO (Normally Open): the normally open configuration works the other way around: there is no connection between the NO and COM pins, so the circuit is broken unless you send a signal from the ESP32 to close the circuit.

Control Pins

The low-voltage side has a set of four pins and a set of three pins. The first set consists of VCC and GND to power up the module, and input 1 (IN1) and input 2 (IN2) to control the bottom and top relays, respectively. If your relay module only has one channel, you'll have just one IN pin. If you have four channels, you'll have four IN pins, and so on.

The signal you send to the IN pins, determines whether the relay is active or not.

The relay is triggered when the input goes below about 2V. This means that you'll have the following scenarios:

- Normally Closed configuration (NC):

HIGH signal – current is flowing

LOW signal – current is not flowing

- Normally Open configuration (NO):

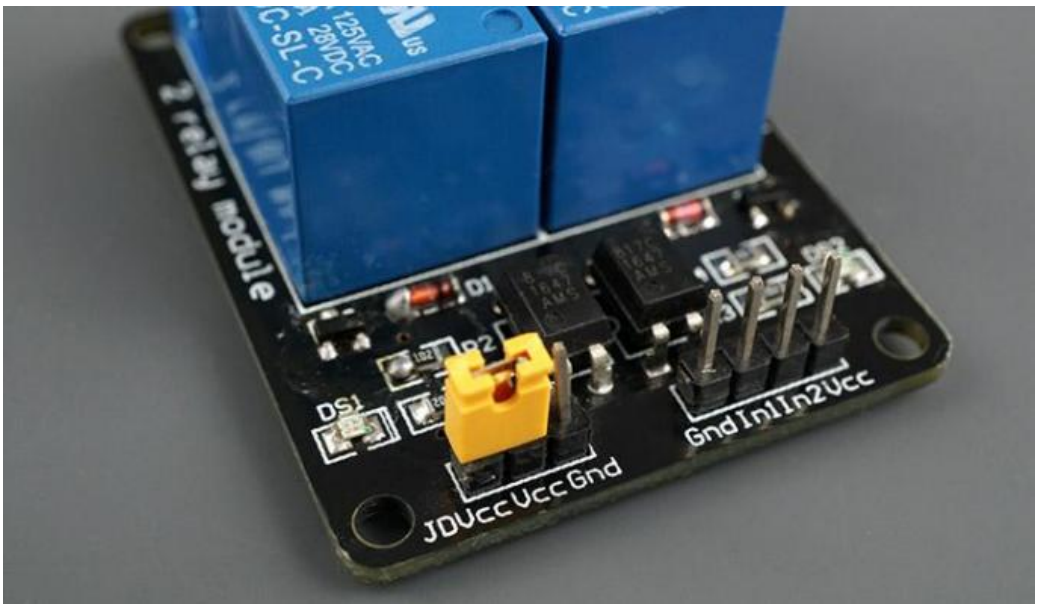
HIGH signal – current is not flowing

LOW signal – current in flowing

You should use a normally closed configuration when the current should be flowing most of the times, and you only want to stop it occasionally.

Use a normally open configuration when you want the current to flow occasionally (for example, turn on a lamp occasionally).

Power Supply Selection



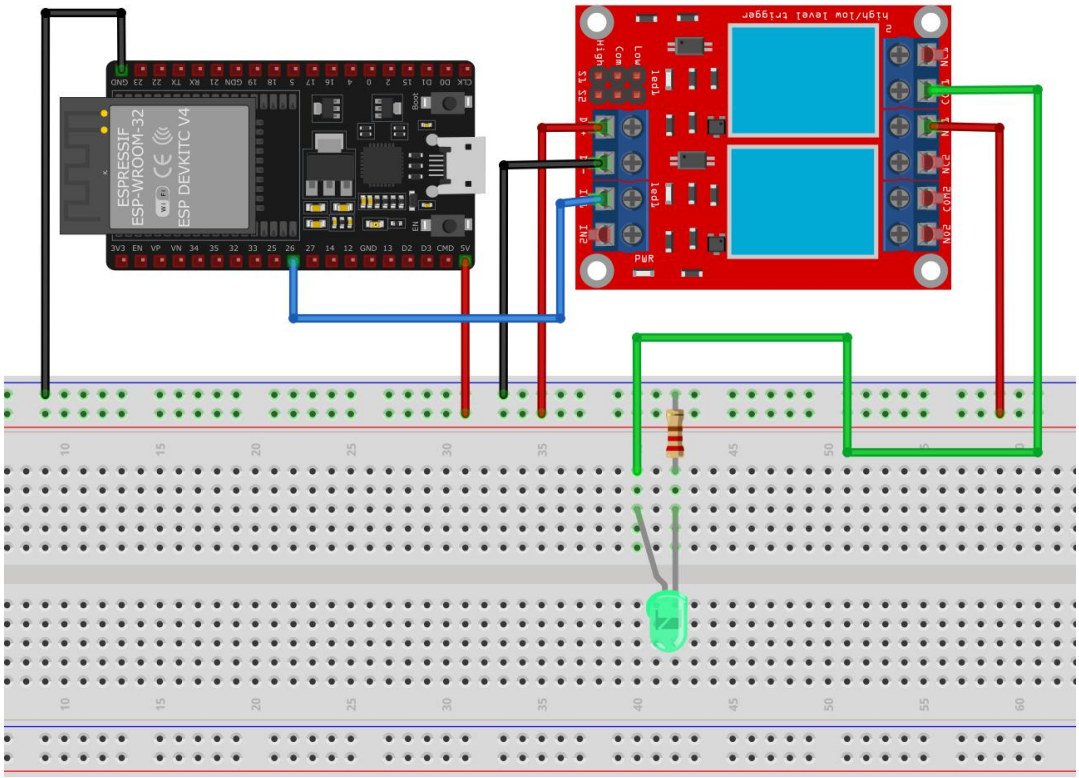
The second set of pins consists of GND, VCC, and JD-VCC pins.

The JD-VCC pin powers the electromagnet of the relay. Notice that the module has a jumper cap connecting the VCC and JD-VCC pins; the one shown here is yellow, but yours may be a different color.

With the jumper cap on, the VCC and JD-VCC pins are connected. That means the relay electromagnet is directly powered from the ESP32 power pin, so the relay module and the ESP32 circuits are not physically isolated from each other.

Without the jumper cap, you need to provide an independent power source to power up the relay's electromagnet through the JD-VCC pin. That configuration physically isolates the relays from the ESP32 with the module's built-in optocoupler, which prevents damage to the ESP32 in case of electrical spikes.

Schematic



Installing the Library for ESP32

To build this web server, we use the ESPAsyncWebServer library and AsyncTCP Library.

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

Alternatively, in your Arduino IDE, you can go to **Sketch** > **Include Library** > **Add .ZIP library…** and select the library you've just downloaded.

Installing the AsyncTCP 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

4. Finally, re-open your Arduino IDE

Alternatively, in your Arduino IDE, you can go to **Sketch** > **Include Library** > **Add .ZIP library…** and select the library you've just downloaded.

Code

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:(If you have already done this step, you can skip to the next step.)

Installing ESP32 Add-on in Arduino IDE

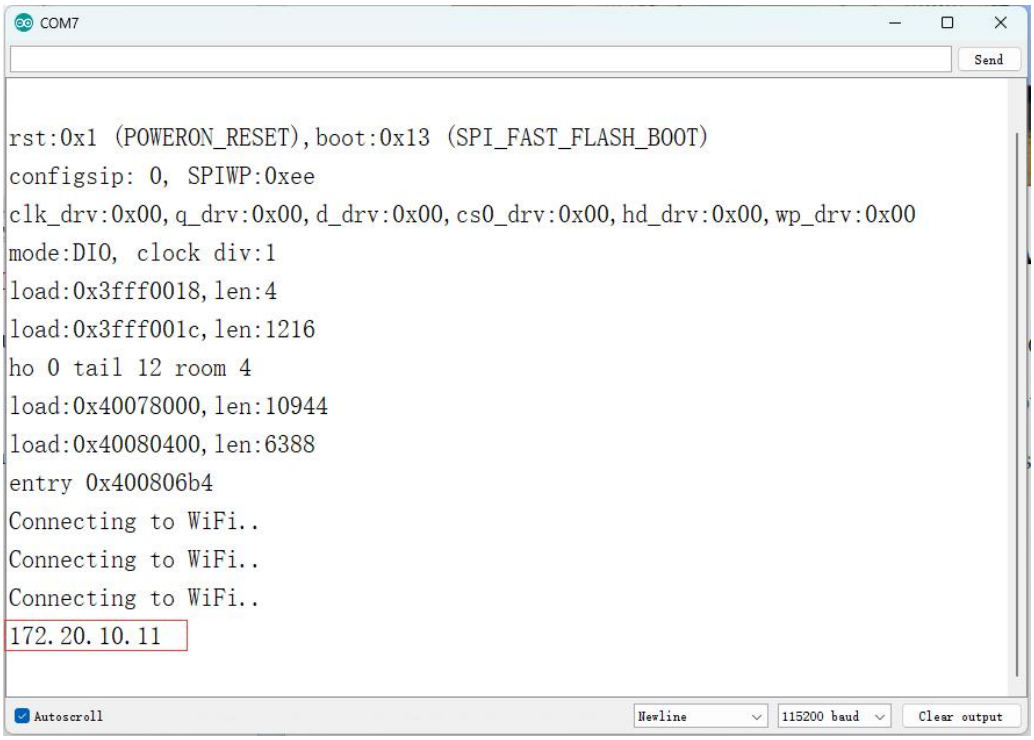
After installing the required libraries, Open the code Project_6_ESP32_Relay_Web_Server.ino in arduino IDE.

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_SSID";
```

Demonstration

After making the necessary changes, upload the code to your ESP32. Open the Serial Monitor at a baud rate of 115200 and press the ESP32 EN button to get its IP address.Then, open a browser in your local network and type the ESP32 IP address to get access to the web server.

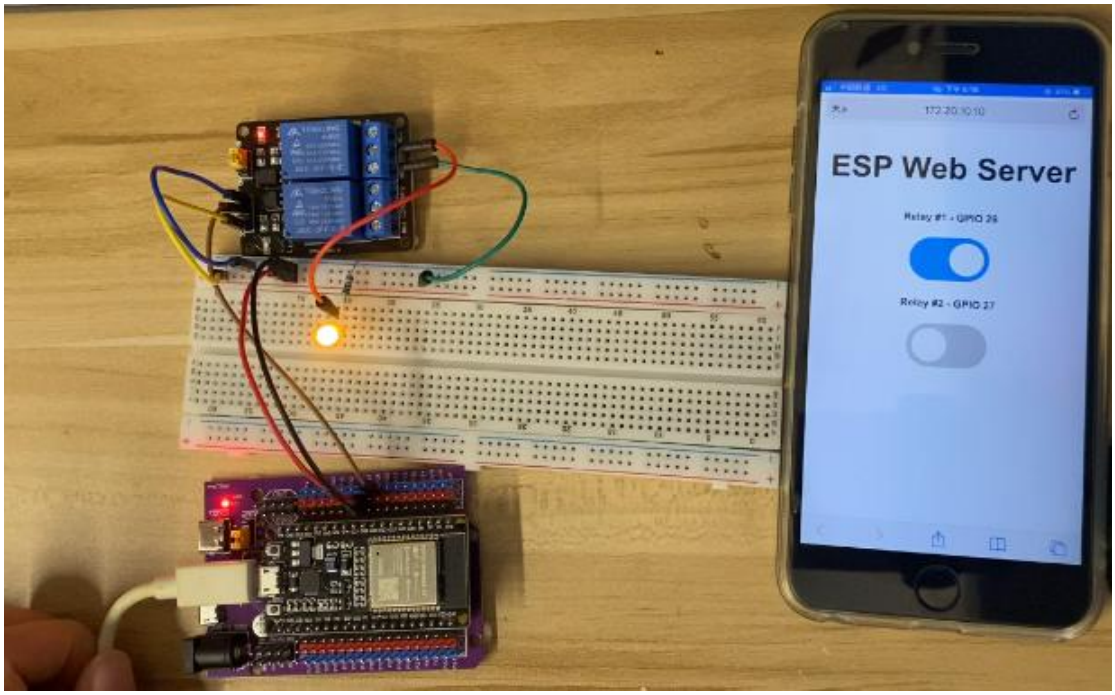


Note: Your browser and ESP32 should be connected to the same LAN.

You should get something as follows with as two buttons as the number of relays you've defined in your code.

Now, you can use the buttons to control your relays using your phone.

LROBRUYA

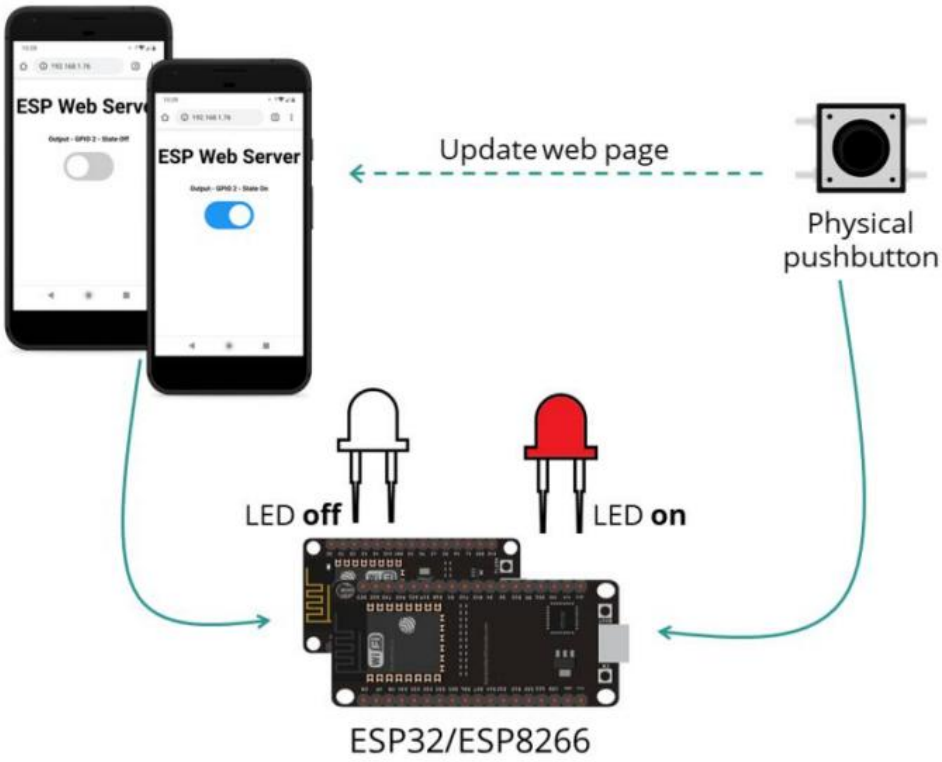


Project 7 Output State Synchronization Web Server

This Project shows how to control the ESP32 or ESP8266 outputs using a web server and a physical button simultaneously. The output state is updated on the web page whether it is changed via physical button or web server.

Project Overview

Let’s take a quick look at how the project works.



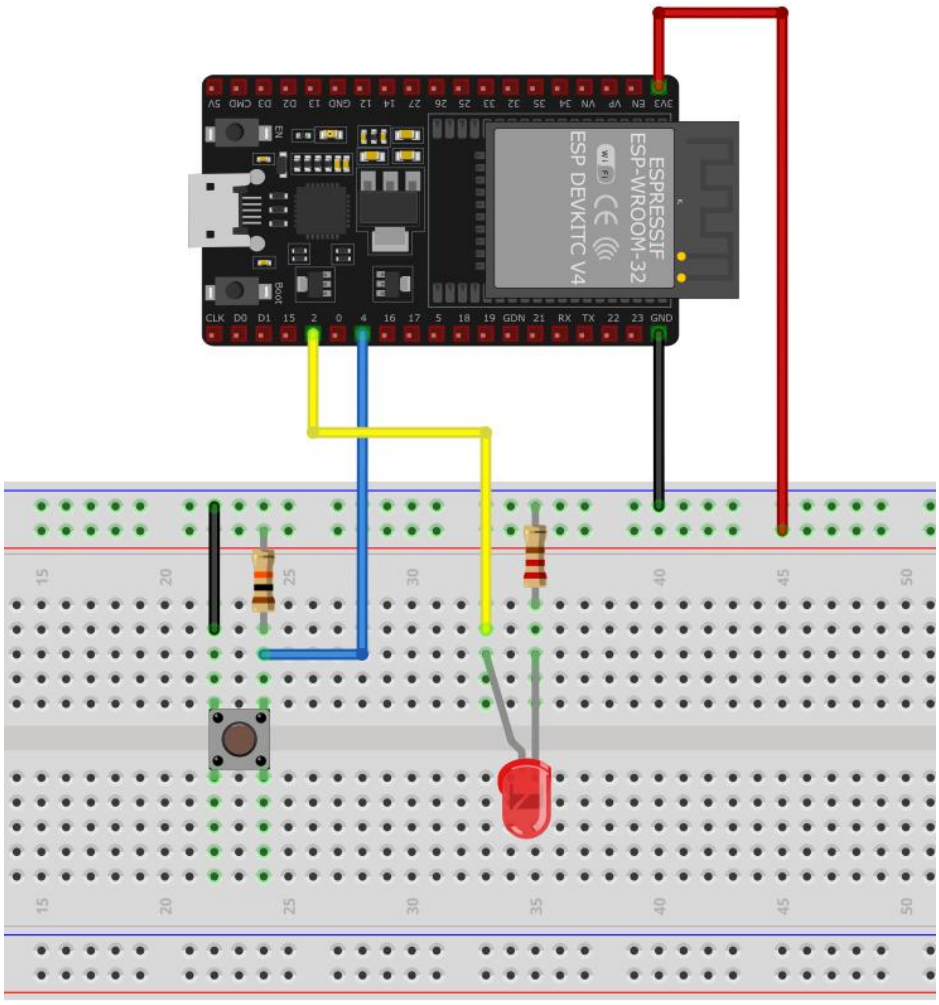
- The ESP32 or ESP8266 hosts a web server that allows you to control the state of an output;
- The current output state is displayed on the web server;
- The ESP is also connected to a physical push button that controls the same output;
- If you change the output state using the physical push button, its current state is also updated on the web server.
- In summary, this project allows you to control the same output using a web server and a push button simultaneously. Whenever the output state changes, the web server is updated.

Parts Required

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

- ESP32-DevKitC V4
- 38P Expansion board
- 5 mm LED
- 220Ohm resistor
- Push button
- 10k Ohm resistor
- Breadboard
- Jumper wires

Schematic



Installing the Library for ESP32

To build this web server, we use the ESPAsyncWebServer library and AsyncTCP Library.(If you have already done this step, you can skip to the next step.)

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

Alternatively, in your Arduino IDE, you can go to **Sketch** > **Include Library** > **Add .ZIP library…** and select the library you've just downloaded.

Installing the AsyncTCP 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

Alternatively, in your Arduino IDE, you can go to **Sketch** > **Include Library** > **Add .ZIP library…** and select the library you've just downloaded.

Code

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:(If you have already done this step, you can skip to the next step.)

Installing ESP32 Add-on in Arduino IDE

After installing the required libraries, Open the code Project_7_Output_State_Synchronization_Web_Server.ino in arduino IDE.

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_SSID";
```

How the Code Works

Button State and Output State

The ledState variable holds the LED output state. For default, when the web server starts, it is LOW.

```
int ledState = LOW; // the current state of the output pin
```

The buttonState and lastButtonState are used to detect whether the pushbutton was pressed or not.

```
int buttonState; // the current reading from the input pin

int lastButtonState = HIGH; // the previous reading from the input pin
```

Button (web server)

We didn't include the HTML to create the button on the the index_html variable.

That's because we want to be able to change it depending on the current LED state that can also be changed with the pushbutton.

So, we've create a placeholder for the button %BUTTONPLACEHOLDER% that will be replaced with HTML text to create the button later on the code (this is done in the processor() function).

<h2>ESP Web Server</h2>

%BUTTONPLACEHOLDER%

processor()

The processor() function replaces any placeholders on the HTML text with actual values. First, it checks whether the HTML texts contains any placeholders %BUTTONPLACEHOLDER%.

```
if(var == "BUTTONPLACEHOLDER"){
```

Then, call theoutputState() function that returns the current output state. We save it in the outputStateValue variable.

```
String outputStateValue = outputState();
```

After that, use that value to create the HTML text to display the button with the right state:

```
buttons+= "<h4>Output - GPIO 2 - State <span
id=\"outputState\"><span></h4><label class=\"switch\"><input
type=\"checkbox\" onchange=\"toggleCheckbox(this)\" id=\"output\" "
+ outputStateValue + "><span class=\"slider\"></span></label>";
```

HTTP GET Request to Change Output State (JavaScript)

When you press the button, thetoggleCheckbox() function is called. This function will make a request on different URLs to turn the LED on or off.

```
function toggleCheckbox(element) {
```

```
var xhr = new XMLHttpRequest();
if(element.checked){ xhr.open("GET", "/update?state=1", true); }
else { xhr.open("GET", "/update?state=0", true); }
xhr.send();}
```

To turn on the LED, it makes a request on the ***/update?state=1*** URL:

```
if(element.checked){ xhr.open("GET", "/update?state=1", true); }
```

Otherwise, it makes a request on the ***/update?state=0*** URL.

HTTP GET Request to Update State (JavaScript)

To keep the output state updated on the web server, we call the following function that makes a new request on the ***/state*** URL every second.

```
setInterval(function ( ) {
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
var inputChecked;
var outputStateM;
if( this.responseText == 1){
inputChecked = true;
outputStateM = "On";
}
```

```
else {
inputChecked = false;
outputStateM = "Off";
}
document.getElementById("output").checked = inputChecked;
document.getElementById("outputState").innerHTML =
outputStateM;
}
};
xhttp.open("GET", "/state", true);
xhttp.send();}, 1000 ) ;
```

Handle Requests

Then, we need to handle what happens when the ESP32 or ESP8266 receives requests on those URLs.

When a request is received on the root / URL, we send the HTML page as well as the processor.

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest*request){
request->send_P(200, "text/html", index_html, processor);});
```

The following lines check whether you received a request on the ***/update?state=1*** or ***/update?state=0*** URL and changes

the ledState accordingly.

```
server.on("/update", HTTP_GET, [] (AsyncWebServerRequest *request)
{
String inputMessage;
String inputParam;
// GET input1 value on <ESP_IP>/update?state=<inputMessage>
if (request->hasParam(PARAM_INPUT_1)) {
inputMessage = request->getParam(PARAM_INPUT_1)->value();
inputParam = PARAM_INPUT_1;
digitalWrite(output, inputMessage.toInt());
ledState = !ledState;
}
else {
inputMessage = "No messag
inputParam = "none";
}
Serial.println(inputMessage);
request->send(200, "text/plain", "OK");});
```

When a request is received on the ***/state*** URL, we send the current output state:

```
server.on("/state", HTTP_GET, [] (AsyncWebServerRequest *request) {
request->send(200, "text/plain",
String(digitalRead(output)).c_str());});
```

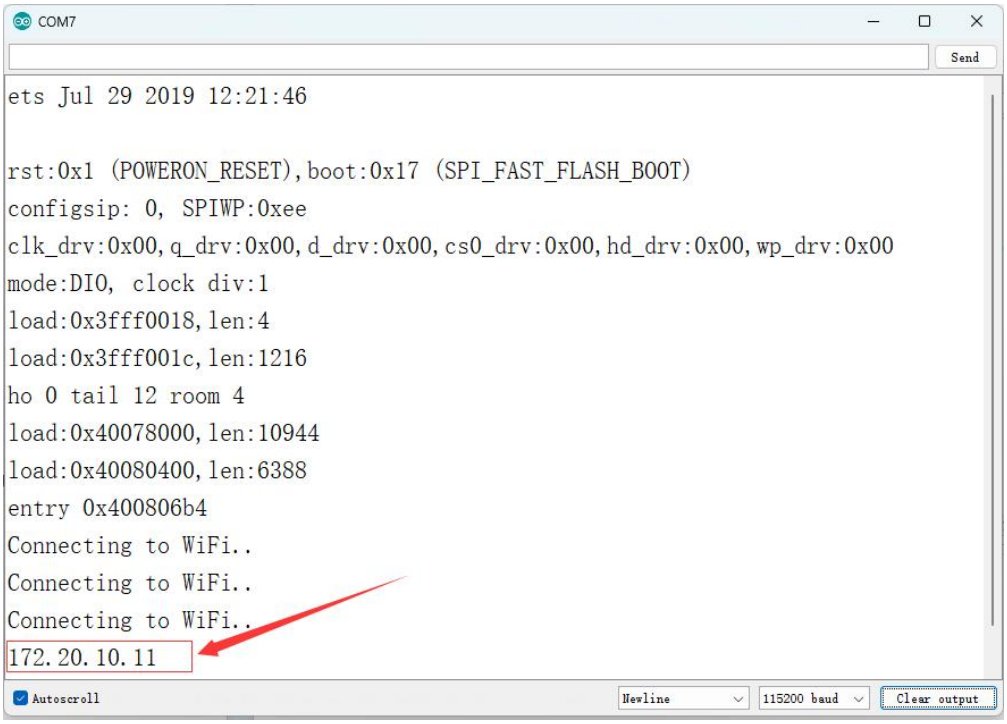
loop()

In the loop(), we debounce the pushbutton and turn the LED on or off depending on the value of the ledState variable.

```
digitalWrite(output, ledState);
```

Demonstration

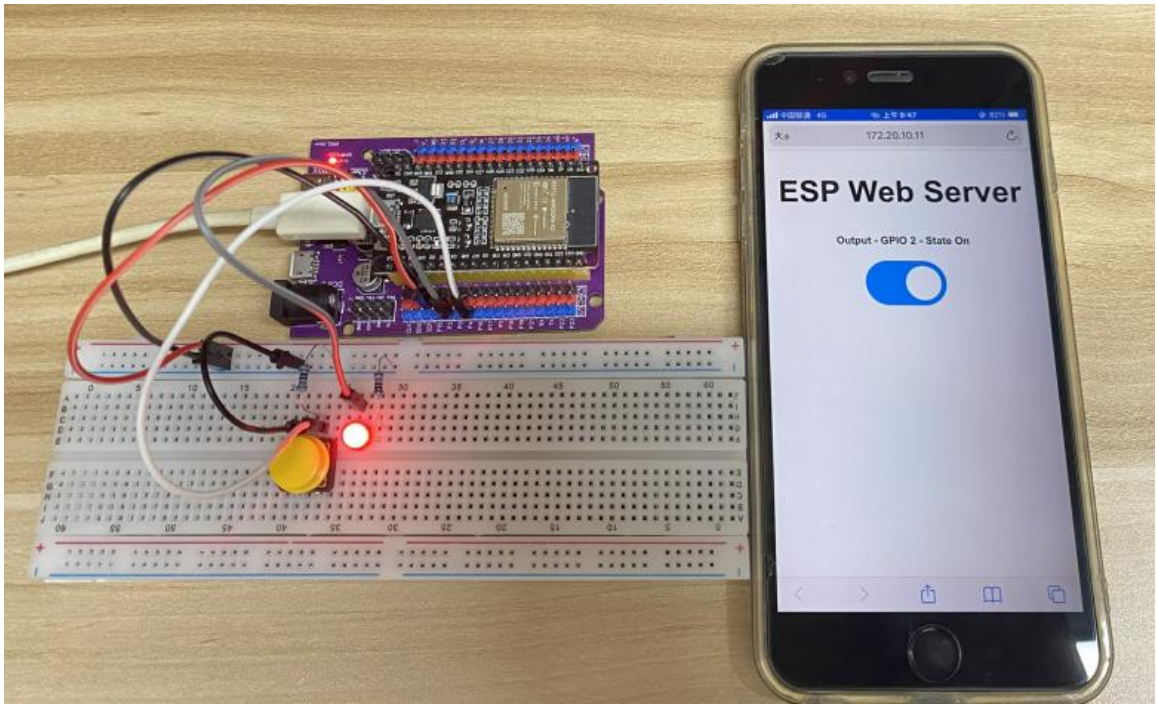
Upload the code to your ESP32 board. Then, open the Serial Monitor at a baud rate of 115200. Press the on-board EN/RST button to get is IP address.



Open a browser on your local network, and type the ESP IP address. You should have access to the web server as shown below.

Note: Your browser and ESP32 should be connected to the same LAN.

You can toggle the button on the web server to turn the LED on.



You can also control the same LED with the physical push button. Its state will always be updated automatically on the web server.

Project 8 ESP32 DHT11 Web Server

In this project, you'll learn how to build an asynchronous ESP32 web server with the DHT11 that displays temperature and humidity using Arduino IDE.

Prerequisites

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;
- 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:

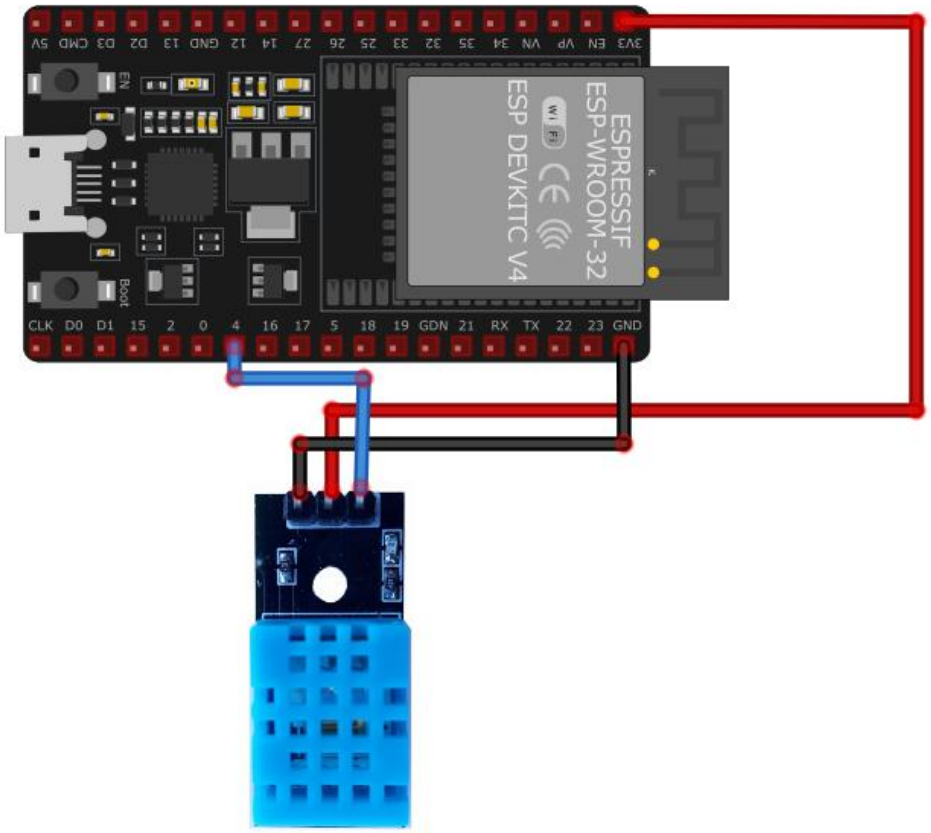
- "Handle more than one connection at the same time";
- "When you send the response, you are immediately ready to handle other connections while the server is taking care of sending the response in the background";
- "Simple template processing engine to handle templates";

Parts Required

To complete this tutorial you need the following parts:

- ESP32-DevKitC V4
- 38P Expansion board
- DHT11 Module
- Breadboard
- Jumper wires

Schematic



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

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

Alternatively, in your Arduino IDE, you can go to **Sketch** > **Include Library** > **Add .ZIP library…** and select the library you've just downloaded.

Installing the Adafruit Unified Sensor Driver

You also need to install the Adafruit Unified Sensor Driver library to work with the DHT sensor. 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 *Adafruit_sensor* folder to your Arduino IDE installation libraries folder

5. Finally, re-open your Arduino IDE

Alternatively, in your Arduino IDE, you can go to **Sketch** > **Include Library** > **Add .ZIP library…** and select the library you've just downloaded.

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

Alternatively, in your Arduino IDE, you can go to **Sketch** > **Include**

Library > **Add .ZIP library…** and select the library you've just downloaded.

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

Alternatively, in your Arduino IDE, you can go to **Sketch** > **Include Library** > **Add .ZIP library…** and select the library you've just downloaded.

Code

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:(If you have already done this step, you can skip to the next step.)

Installing ESP32 Add-on in Arduino IDE

After installing the required libraries, Open the code Project_8_ESP32_DHT11_Web_Server.ino in arduino IDE.

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_SSID";
```

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 <ESPAsyncTCP.h>
#include <Adafruit_Sensor.h>
#include <DHT.h>
```

Variables definition

Define the GPIO that the DHT data pin is connected to. In this case, it's connected to GPIO 4.

```
#define DHTPIN 4 // 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 DHT11
// DHT 11
```

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 We've created two functions: one to read the temperature (readDHTTemperature()) and the other to read humidity (readDHTHumidity()).

```
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 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);
```

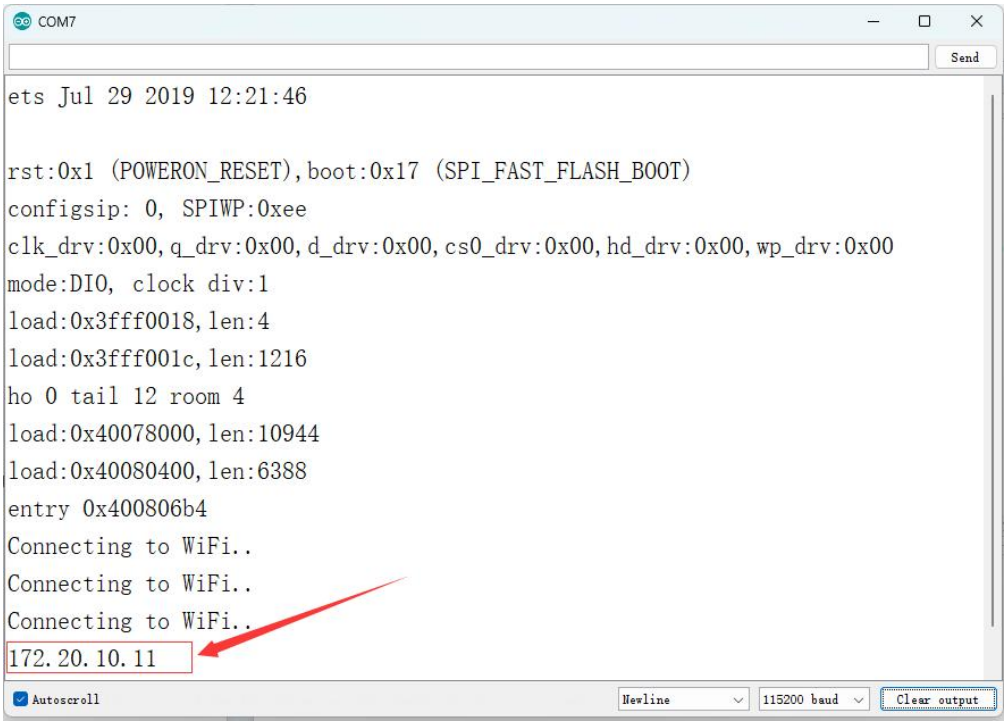
By default, we're reading the temperature in Celsius degrees. To get the temperature in Fahrenheit degrees, comment the temperature in Celsius and uncomment the temperature in Fahrenheit, so that you have the following:

```
float t = dht.readTemperature();
// Read temperature as Fahrenheit (isFahrenheit = true)
//float t = dht.readTemperature(true);
```

Upload the Code

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

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.



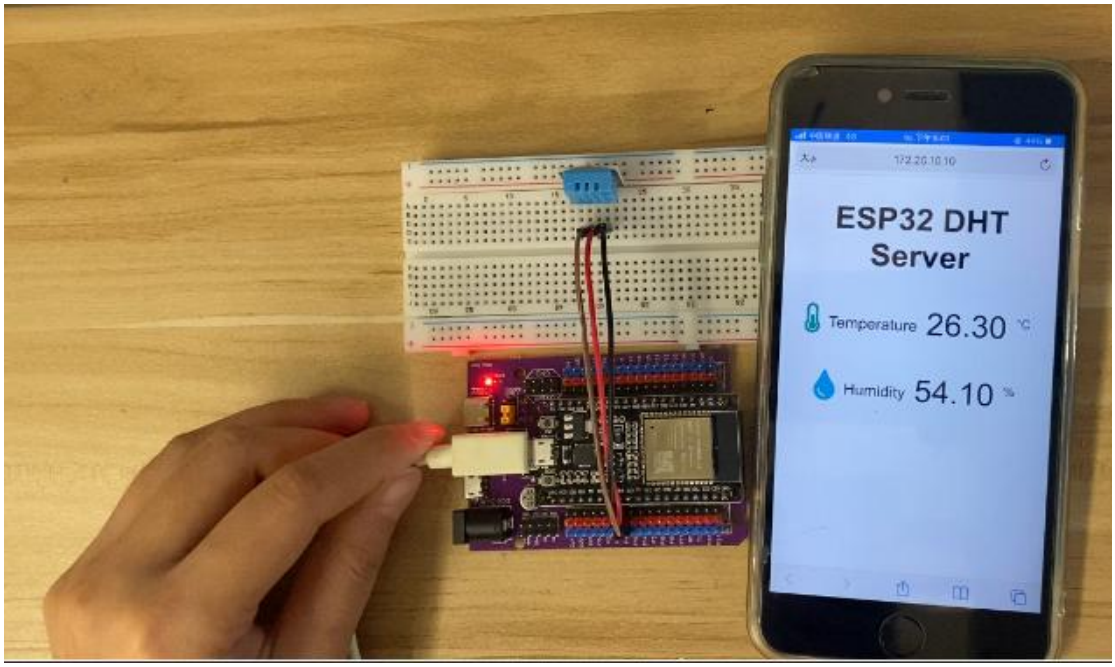
Demonstration

Open a browser and type the ESP32 IP address. Your web server should display the latest sensor readings.

Note: Your browser and ESP32 should be connected to the same LAN.

Notice that the temperature and humidity readings are updated automatically without the need to refresh the web page.

LROBRUYA

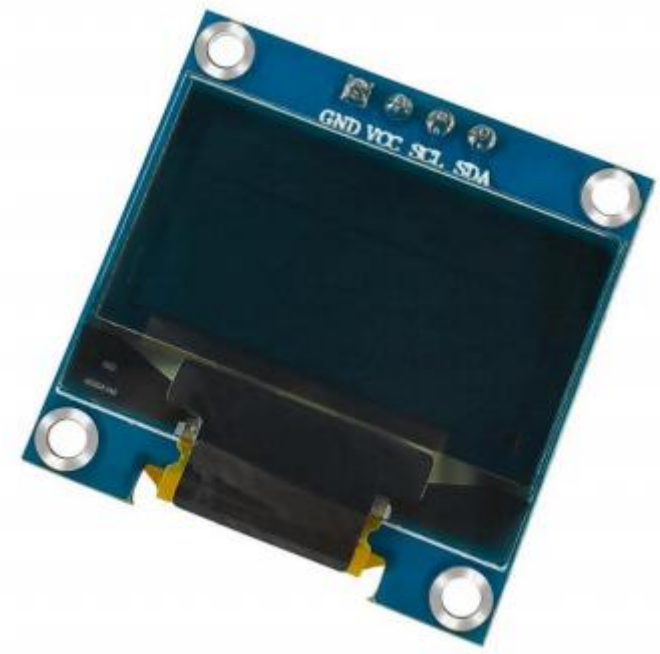


Project 9 ESP32_OLED_Display

This project shows how to use the 0.96 inch SSD1306 OLED display with ESP32 using Arduino IDE.

Introducing 0.96 inch OLED Display

The OLED display that we'll use in this tutorial is the SSD1306 model: a monocolor, 0.96 inch display with 128×64 pixels as shown in the following figure.

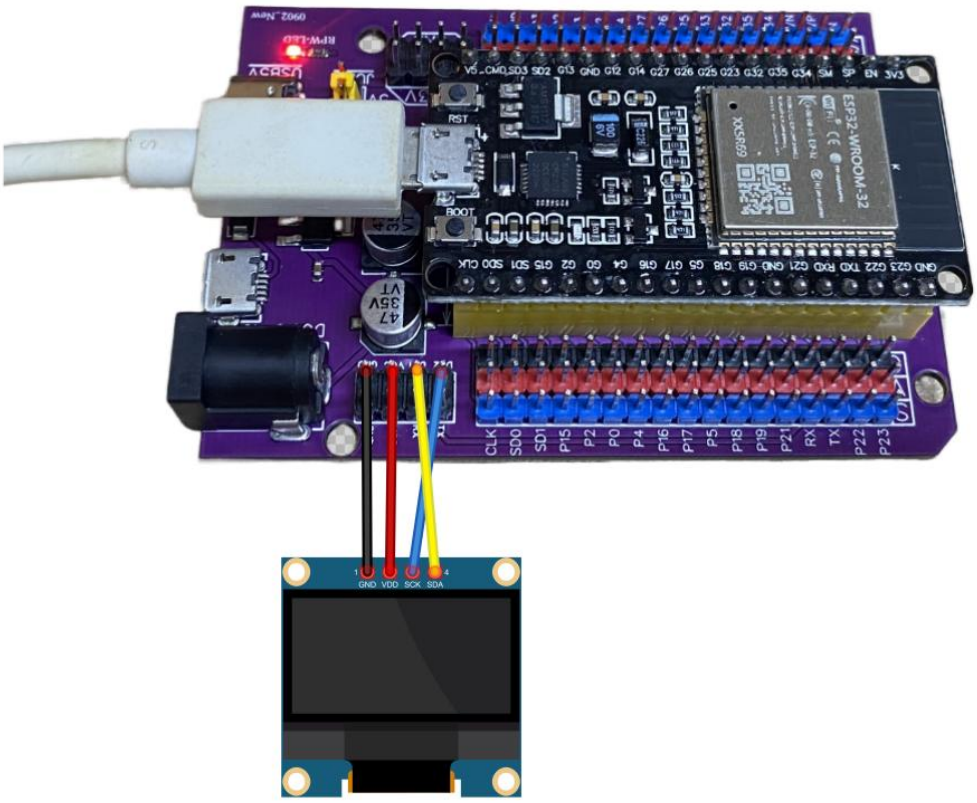


The OLED display doesn’t require backlight, which results in a very nice contrast in dark environments. Additionally, its pixels consume energy only when they are on, so the OLED display consumes less power when compared to other displays.

Because the OLED display uses I2C communication protocol, wiring is very simple. You can use the following table as a reference.

OLED Pin	ESP32
Vin	3.3V
GND	GND
SCL	GPIO 22
SDA	GPIO 21

Schematic



Installing SSD1306 OLED Library – ESP32

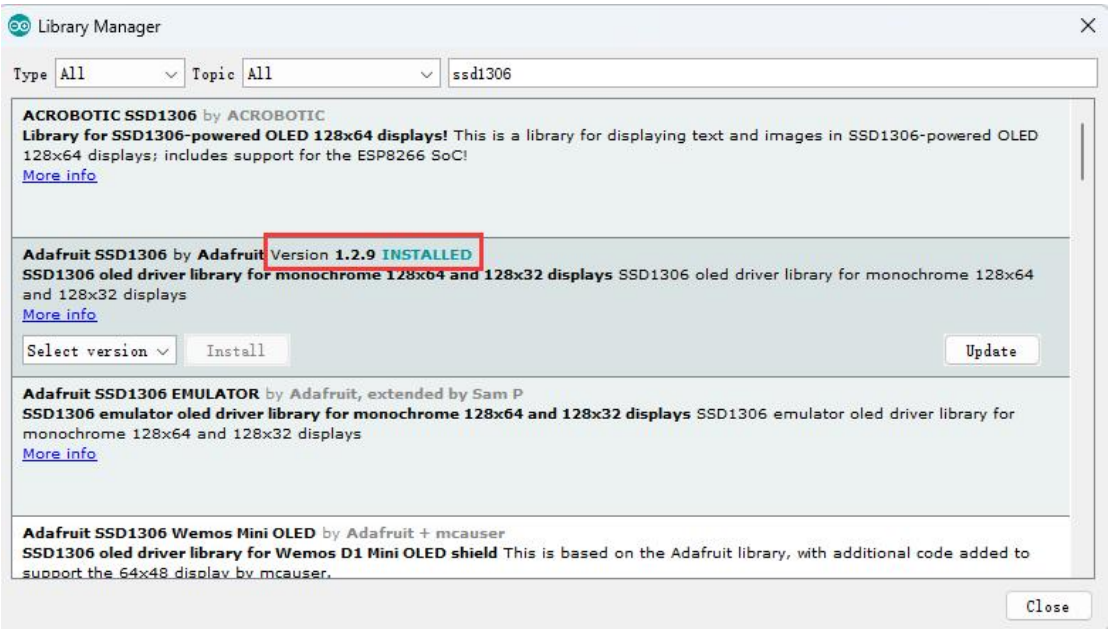
There are several libraries available to control the OLED display with the ESP32.

In this tutorial we'll use two Adafruit libraries: Adafruit_SSD1306 library and Adafruit_GFX library.

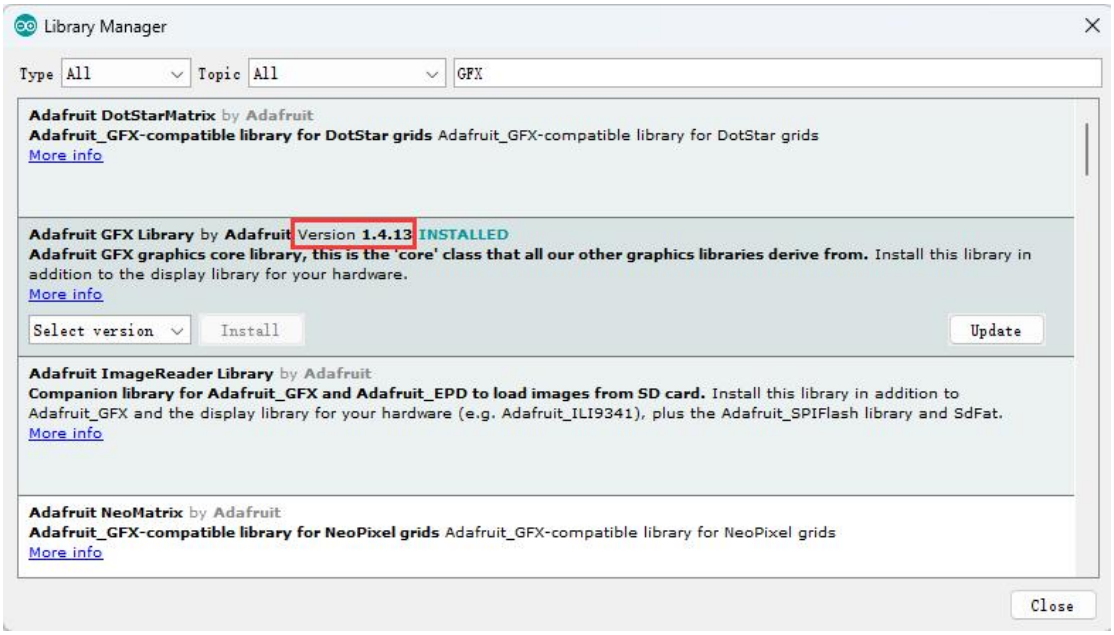
Follow the next steps to install those libraries.

1. Open your Arduino IDE and go to **Sketch** > **Include Library** > **Manage Libraries**. The Library Manager should open.

2. Type “**SSD1306**” in the search box and install the SSD1306 library from Adafruit.



3. After installing the SSD1306 library from Adafruit, type “**GFX**” in the search box and install the library.



4. After installing the libraries, restart your Arduino IDE.

Code

After installing the required libraries, Open the code Project_9_ESP32_OLED_Display.ino in arduino IDE.

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:(If you have already done this step, you can skip to the next step.)

Installing ESP32 Add-on in Arduino IDE

```
#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)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT,
&Wire, -1);
void setup() {
Serial.begin(115200);
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address
0x3D for 128x64
Serial.println(F("SSD1306 allocation failed"));
```

```
for(;;);
}
delay(2000);
display.clearDisplay();
display.setTextSize(2);
display.setTextColor(WHITE);
display.setCursor(0, 30);
// Display static text
display.println("LONTEN");
display.display();
delay(100);
}
void loop() {
// Scroll in various directions, pausing in-between:
display.startscrollright(0x00, 0x0F);
delay(7000);
display.stopscroll();
delay(1000);
display.startscrollleft(0x00, 0x0F);
delay(7000);
```

```
display.stopscroll();
delay(1000);
}
```

How the Code Works

Importing libraries

First, you need to import the necessary libraries. The Wire library to use I2C and the Adafruit libraries to write to the display: Adafruit_GFX and Adafruit_SSD1306.

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
```

Initialize the OLED display

Then, you define your OLED width and height. In this example, we're using a 128×64 OLED display. If you're using other sizes, you can change that in the SCREEN_WIDTH, and SCREEN_HEIGHT variables.

```
#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels
```

Then, initialize a display object with the width and height defined earlier with I2C communication protocol (&Wire).

```
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT,
&Wire, -1);
```

The (-1) parameter means that your OLED display doesn't have a RESET pin. If your OLED display does have a RESET pin, it should be connected to a GPIO. In that case, you should pass the GPIO number as a parameter.

In the setup(), initialize the Serial Monitor at a baud raute of 115200 for debugging purposes.

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

Initialize the OLED display with the begin() method as follows:

```
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
Serial.println("SSD1306 allocation failed");
for(;;); // Don't proceed, loop forever}
```

This snippet also prints a message on the Serial Monitor, in case we're not able to connect to the display.

```
Serial.println("SSD1306 allocation failed");
```

In case you're using a different OLED display, you may need to change the OLED address. In our case, the address is 0x3C.

```
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
```

After initializing the display, add a two second delay, so that the OLED has enough time to initialize before writing text:

```
delay(2000);
```

Clear display, set font size, color and write text

After initializing the display, clear the display buffer with the clearDisplay() method:

```
display.clearDisplay();
```

Before writing text, you need to set the text size, color and where the text will be displayed in the OLED.

Set the font size using the setTextSize() method:

```
display.setTextSize(1);
```

Set the font color with the setTextColor() method:

```
display.setTextColor(WHITE);
```

WHITE sets white font and black background.

Define the position where the text starts using the setCursor(x,y) method. In this case, we're setting the text to start at the (0,0) coordinates – at the top left corner.

```
display.setCursor(0,0);
```

Finally, you can send the text to the display using the println() method, as follows:

```
display.println("LONTEN");
```

Then, you need to call the display() method to actually display the text on the screen.

```
display.display();
```

The Adafruit OLED library provides useful methods to easily scroll text.

startscrollright(0x00, 0x0F): scroll text from left to right

startscrollleft(0x00, 0x0F): scroll text from right to left

startscrolldiagright(0x00, 0x07): scroll text from left bottom corner to right upper corner

startscrolldiagleft(0x00, 0x07): scroll text from right bottom corner to left upper corner

Upload the Code

Now, upload the code to your ESP32.

After uploading the code, the OLED will display scrolling text.

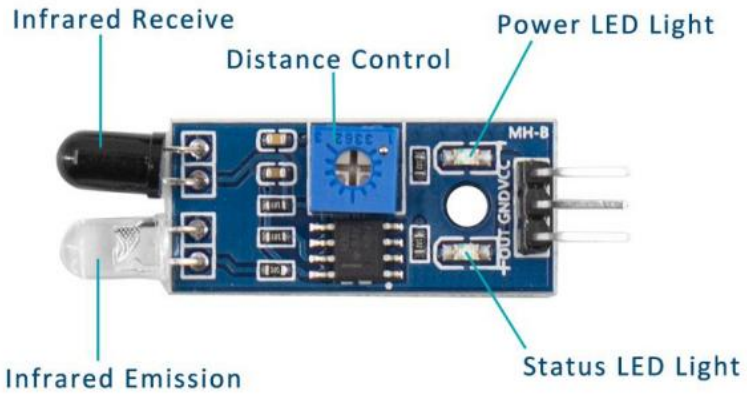
Project 10 Obstacle avoidance using infrared sensors

This project shows how to use the Obstacle avoidance using infrared sensors with ESP32 using Arduino IDE.

Introducing Obstacle avoidance using infrared sensors

The infrared obstacle detector sensor has a pair of infrared transmitting and receiving tubes.The transmitter emits an infrared rays of a certain frequency. When the detection direction encounters an obstacle(reflecting surface), the infrared rays are reflected back, and receiving tube will

receive it.At this time, the indicator (green LED) lights up. After processed by the circuit, the signal output terminal will output Digital signal.You can rotate the potentiometer on the shield to adjust the detection distance. It is better to adjust the potentiometer to make the green LED in a state between on and off. The detection distance is the best,almost 10cm.

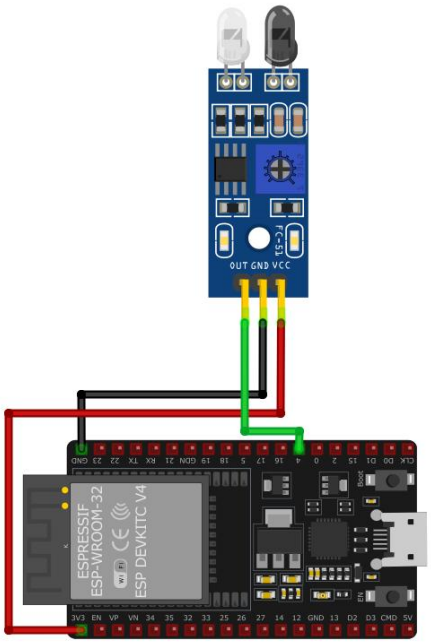


How to use the infrared obstacle avoidance sensor

we read the signal level of obstacle detector sensor to judge whether detect obstacles or not.

When detects an obstacle, sensor’ s signal pin outputs LOW (display 0); otherwise, output HIGH (display 1). Show the result on the serial monitor, and control the external LED module turn ON/OFF.

Schematic



Code

After installing the required libraries, Open the code Project_10_Obstacle_avoidance_using_infrared_sensors.ino in arduino IDE.

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:(If you have already done this step, you can skip to the next step.)

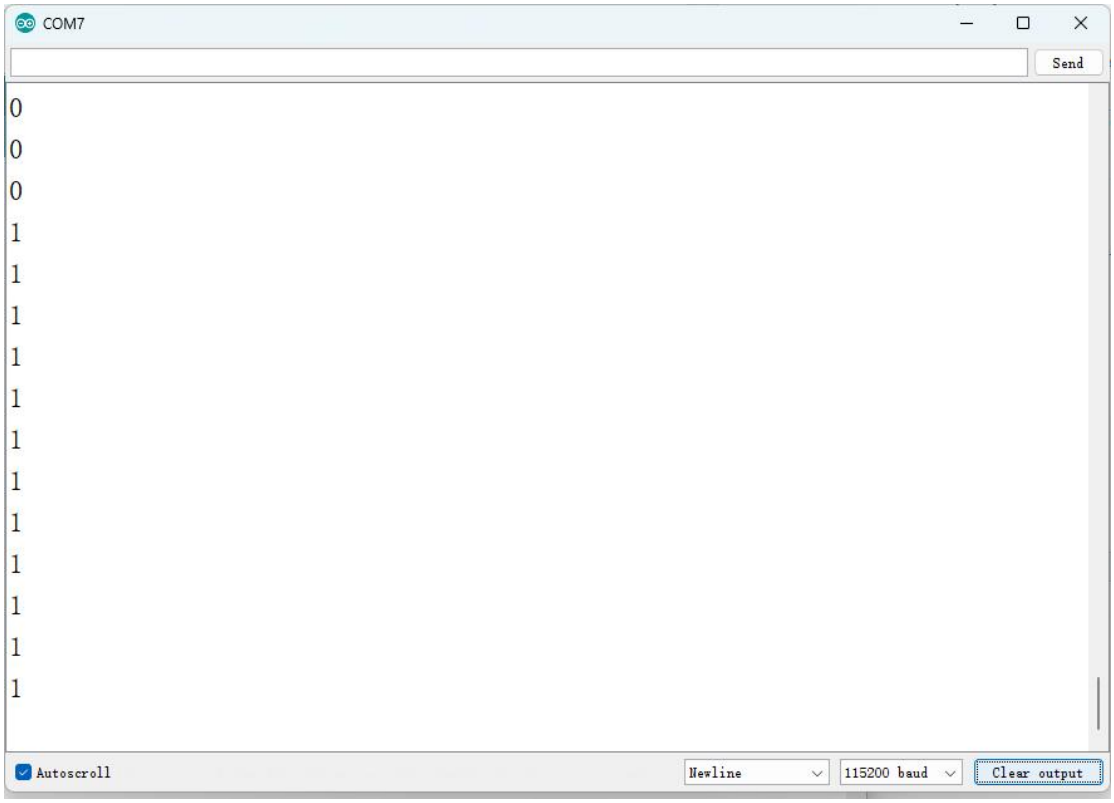
Installing ESP32 Add-on in Arduino IDE

```
// set pin numbers
const int valuePin = 4;
```

```
int value;

void setup() {

Serial.begin(115200);

// initialize the pin as an input

pinMode(valuePin, INPUT);

}

void loop() {

value = digitalRead(valuePin);

Serial.println(value);

delay(100);

}
```

Upload the Code

Now, upload the code to your ESP32.



Project 11 Photoresistor Controlled LED

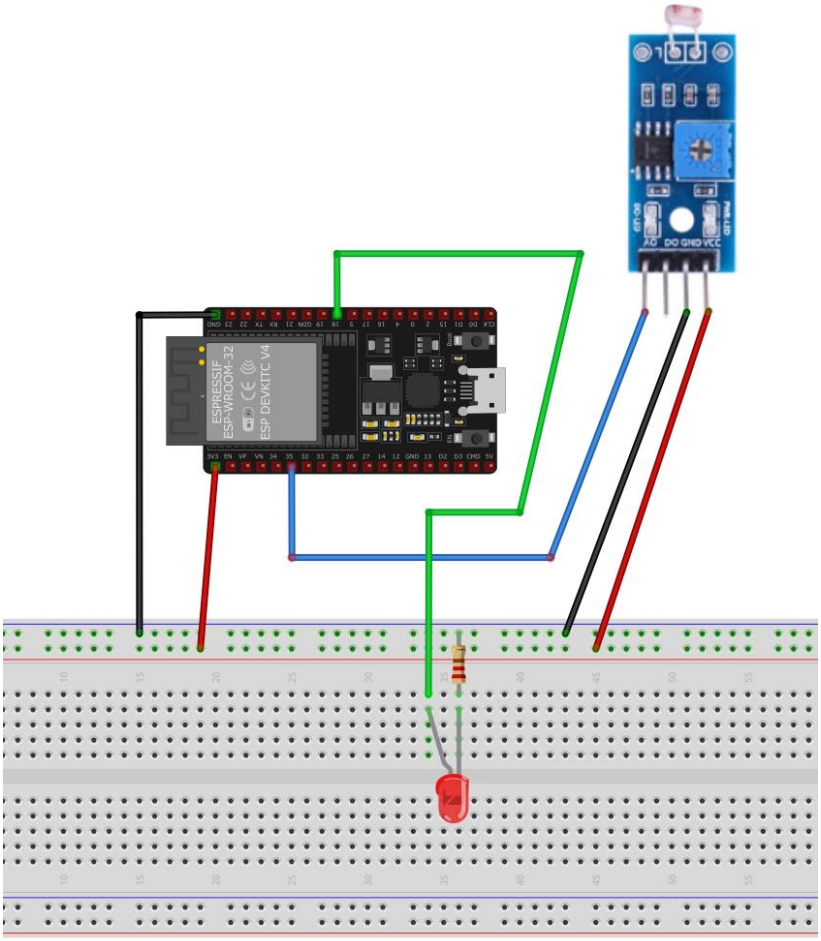
This project shows how to use the Photoresistor Controlled LED with ESP32 using Arduino IDE.

Introducing Photoresistor

The simplest electronic device in a photosensitive sensor is a photoresistor, which can sense changes in the brightness of light and output weak electrical signals. Through simple electronic circuit amplification processing, it can control the automatic switching of LED lamps. Therefore, it is widely used in automatic control and household

appliances, for remote lighting fixtures, such as automatic brightness adjustment in televisions and automatic exposure in cameras; In addition, in automatic control circuits such as streetlights and navigation aids, as well as automatic stop devices and anti-theft alarm devices.

Schematic



Code

After installing the required libraries, Open the code Project_11_Photoresistor_Controlled_LED.ino in arduino IDE.

We'll program the ESP32 using Arduino IDE, so make sure you have the ESP32 add-on installed before proceeding:(If you have already done this step, you can skip to the next step.)

Installing ESP32 Add-on in Arduino IDE

```
int freq = 5000;        // frequency
int channel = 0;        // thoroughfare
int resolution = 8;      // resolution ratio
const int led = 18;
const int gm = 35; //Photosensitive resistor pins
int Filter_Value;
void setup()
{
   Serial.begin(115200);
   //Set photoresistor input
   pinMode(gm,INPUT);
   randomSeed(analogRead(gm));
```

```
    ledcSetup(channel, freq, resolution); // Set up channels
    ledcAttachPin(led, channel);    // Connect the channel to the
corresponding pin
}
#define FILTER_N 20
int Filter() {
    int i;
    int filter_sum = 0;
    int filter_max, filter_min;
    int filter_buf[FILTER_N];
    for(i = 0; i < FILTER_N; i++) {
        filter_buf[i] = analogRead(gm);
        delay(1);
    }
    filter_max = filter_buf[0];
    filter_min = filter_buf[0];
    filter_sum = filter_buf[0];
    for(i = FILTER_N - 1; i > 0; i--) {
        if(filter_buf[i] > filter_max)
            filter_max=filter_buf[i];
```

```
        else if(filter_buf[i] < filter_min)
            filter_min=filter_buf[i];
        filter_sum = filter_sum + filter_buf[i];
        filter_buf[i] = filter_buf[i - 1];
    }
    i = FILTER_N - 2;
    filter_sum = filter_sum - filter_max - filter_min + i / 2; // +The purpose
of i/2 is to round
    filter_sum = filter_sum / i;
    return filter_sum;
}
void loop()
{
    Filter_Value = Filter();
    ledcWrite(0,map(Filter_Value,0,4095,0,255));
}
```

Upload the Code

Now, upload the code to your ESP32.

LROBRUYA