ESP32 WIFI IOT tutorial

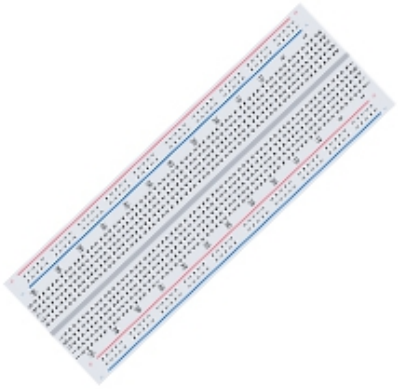
202102025V1.0

List


5V 2-CHANNEL
RELAY MODULE
1PC
PHOTOSENSITIVE
RESISTOR
MODULE
1PC
ESP32 ESP-32S
WIFIBOARD
1PC
IR SENSOR
MODULE
1PC
WITH MICRO
USB CABLE
1PC


128X64 OLED DISPLAY MODULE 1PC
HC-SR501 PIR SENSOR MODULE 1PC
REMOVE SEAL AFTER WASHING
ACTIVE BUZZER 1PC
LED BLUE/WHITE/ RED/YELLOW (5PCS/EACH) 4 X 5 PCS
PASSIVE BUZZER 1PC

830 BREADBOARD
1PC



BUTTON SWITCH
12 * 12 WITH
YELLOW KEYCAP
2 X 5 PCS

DHT11 SENSOR
MODULE
1PC


20PIN 20CM MF
JUMPER WIRE
1PC
20PIN 20CM MM
JUMPER WIRE
1PC
65PCS JUMPER
WIRE
1PC
10R ~1M METAL
FILM RESISTOR
10 X 10 PCS

Preface

Lesson 0 Installing the ESP32 Board in Arduino IDE

There's an add-on for the Arduino IDE that allows you to program the ESP32 using the Arduino IDE and its programming language. 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.

Install Arduino IDE

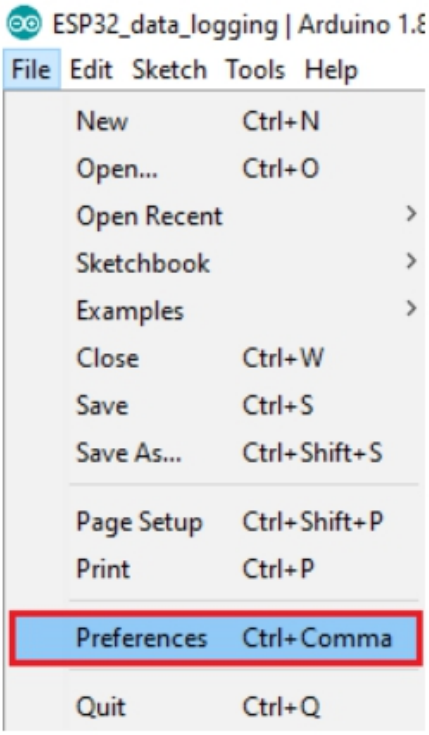
Before starting this installation process, please make sure you have installed the latest version of Arduino IDE on your computer. If not, please uninstall and reinstall. Otherwise, it may not work.

Install the latest Arduino IDE software from arduino.cc/en/Main/Software, continue this tutorial.

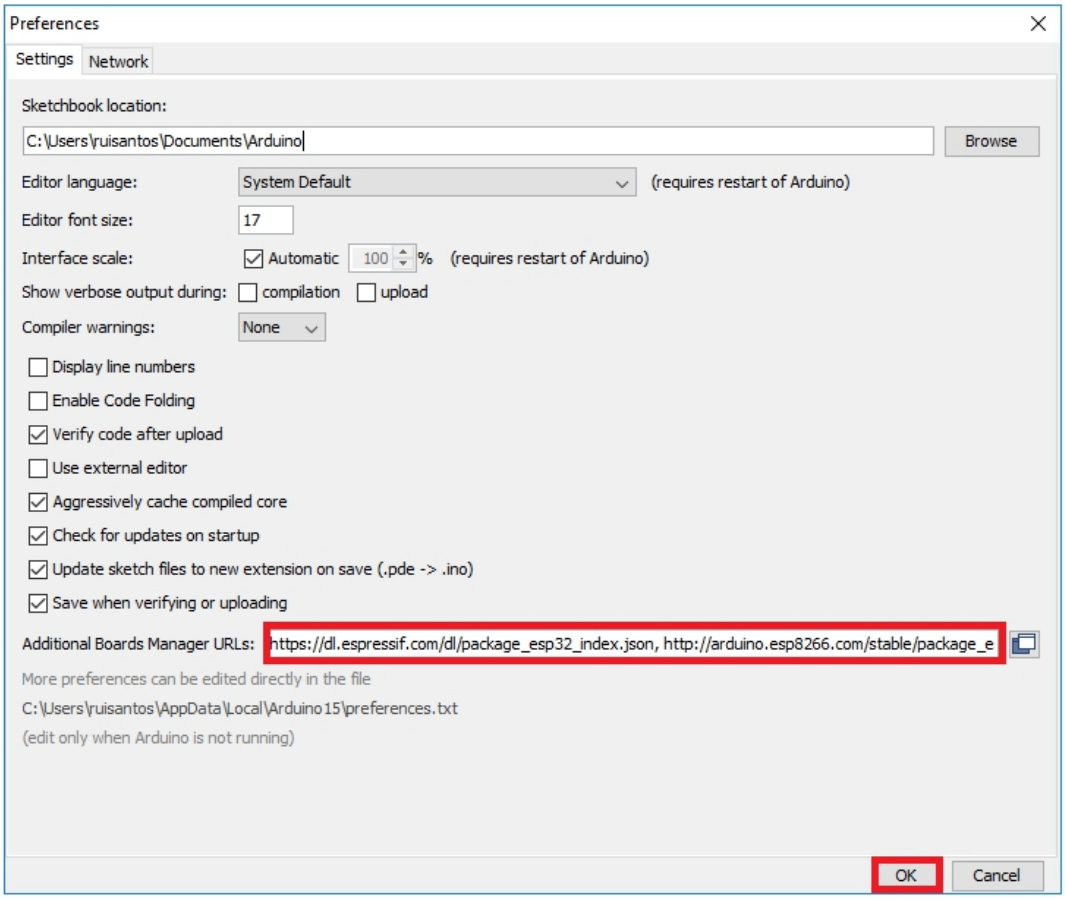
Installing ESP32 Add-on in Arduino IDE

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

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

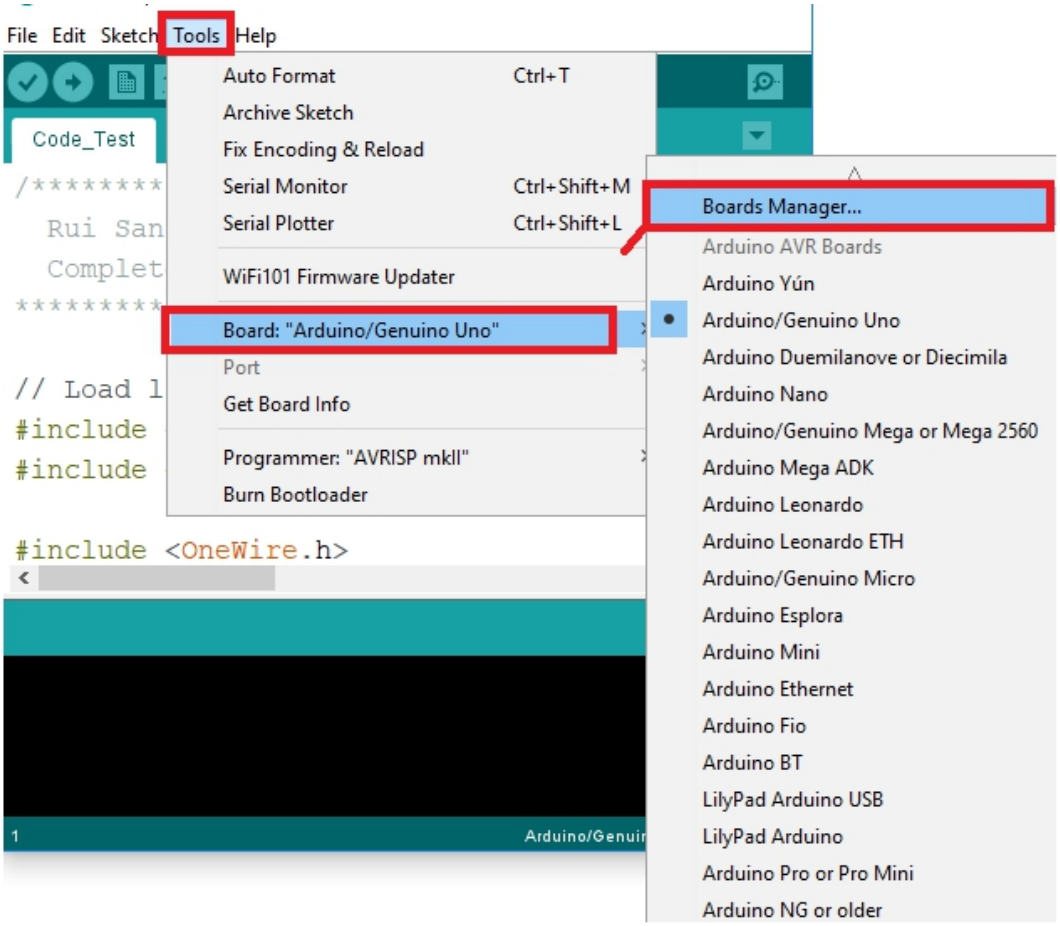


2、Enter https://dl.espressif.com/dl/package_esp32_index.json into the “Additional Board Manager URLs” field as shown in the figure below. Then, click the “OK” button:

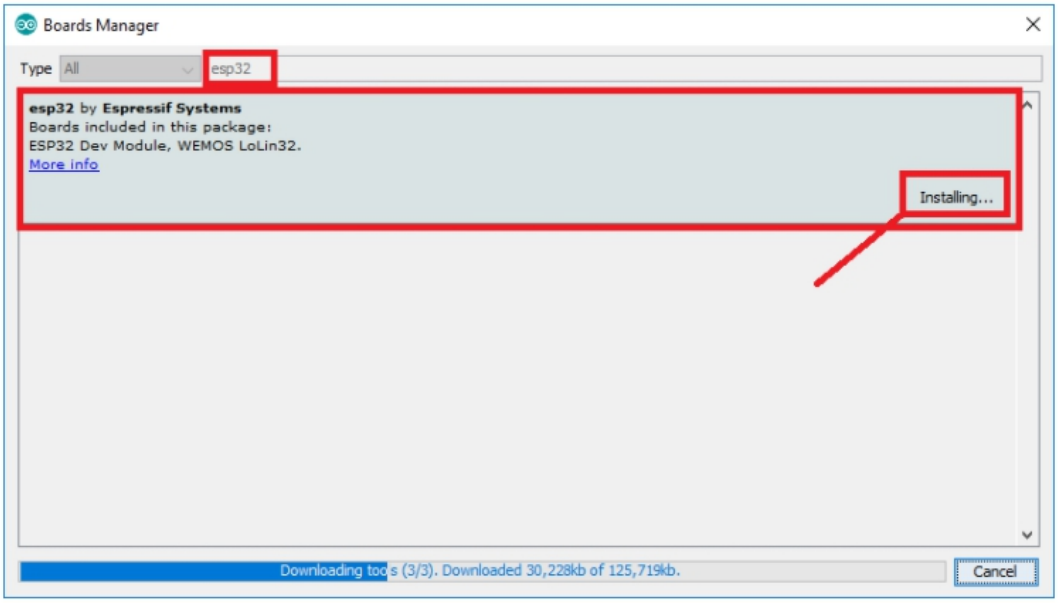


Note: if you already have the ESP8266 boards URL, you can separate the URLs with a comma as follows:
https://dl.espressif.com/dl/package_esp32_index.json,
http://arduino.esp8266.com/stable/package_esp8266com_index.json

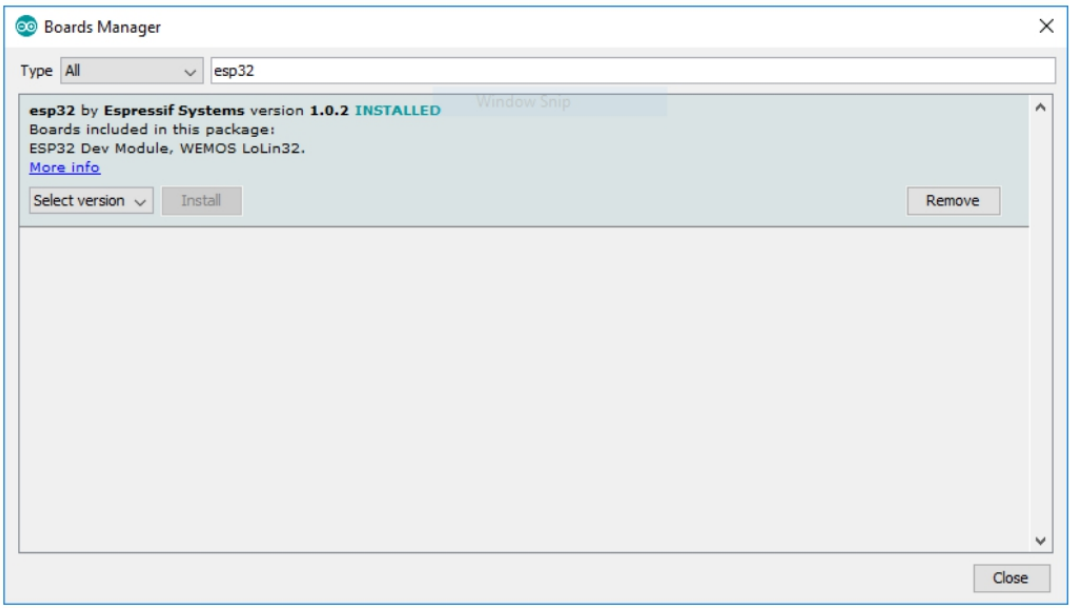
3、 Open the Boards Manager. Go to Tools > Board > Boards Manager…



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



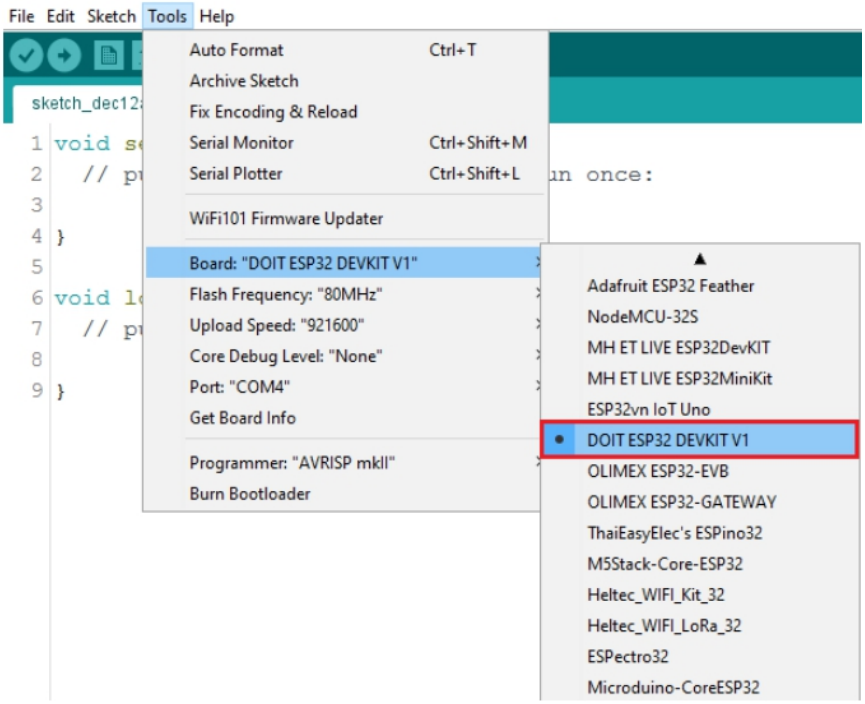
5、That’s it. It should be installed after a few seconds.



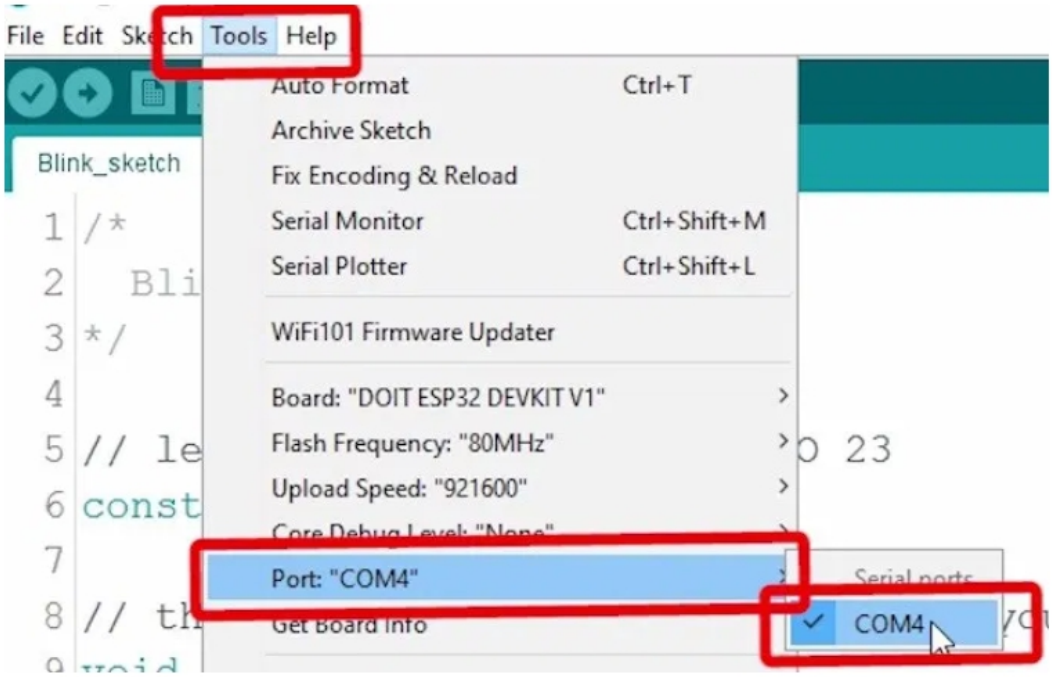
Testing the Installation

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

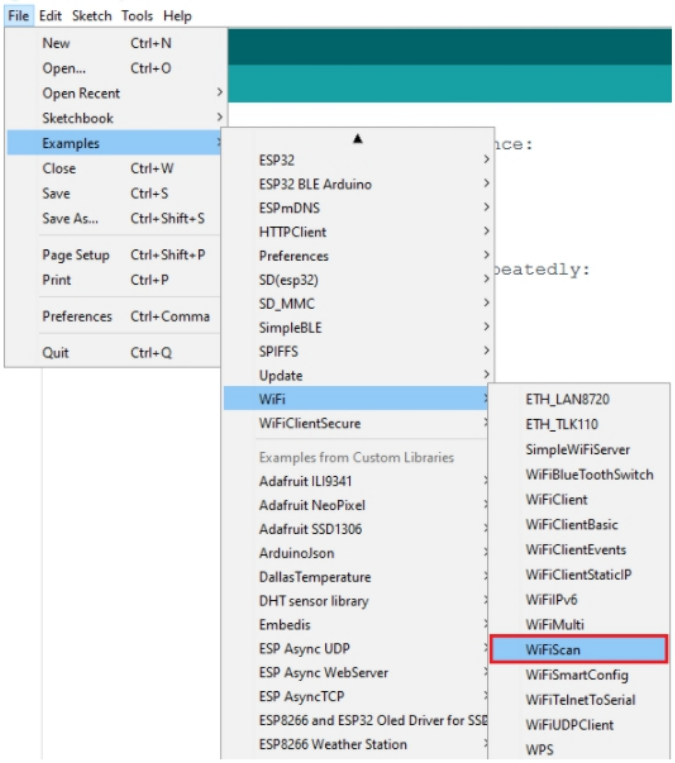
1. Select your Board in Tools > Board menu (in my case it’s the DOIT ESP32 DEVKIT V1)



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



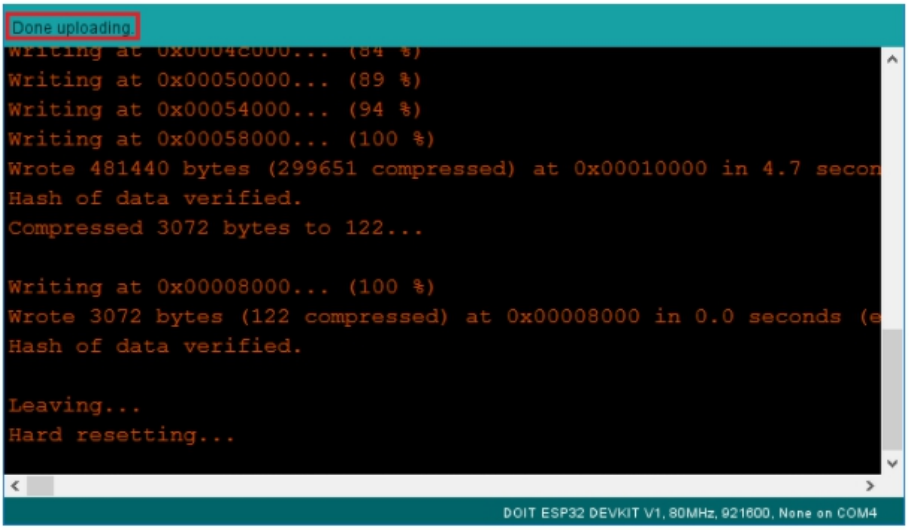
3、Open the following example under File > Examples > WiFi (ESP32) > WiFiScan



4、A new sketch opens in your Arduino IDE:

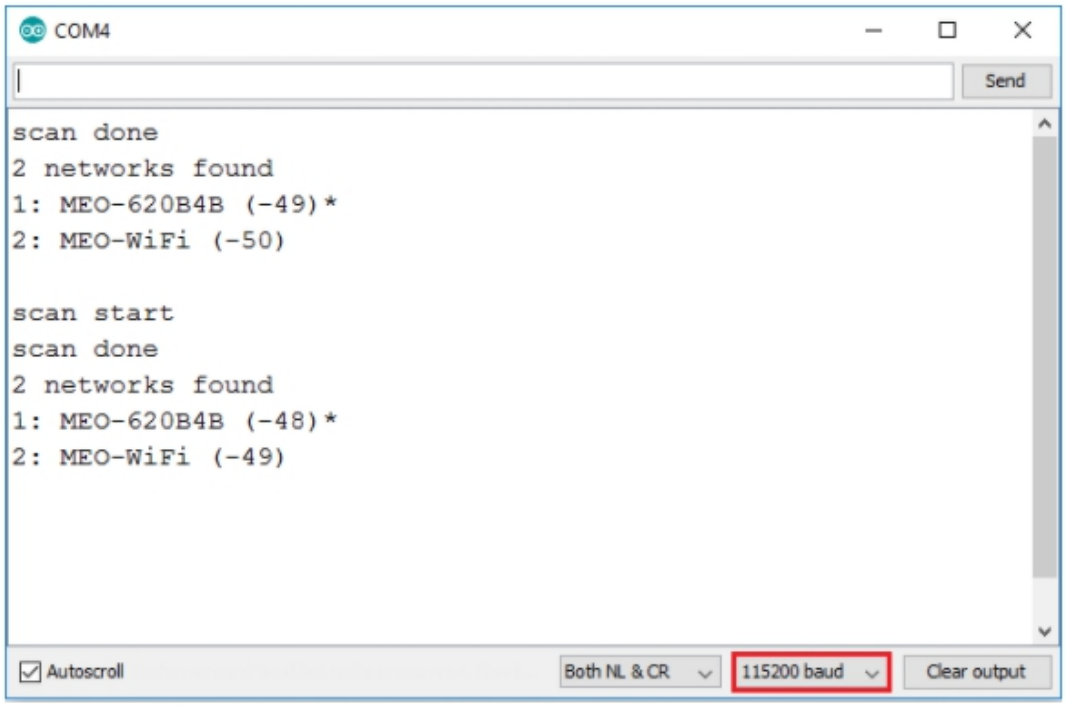
5、 Press the Upload button in the Arduino IDE. Wait a few seconds while the code compiles and uploads to your board.

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:

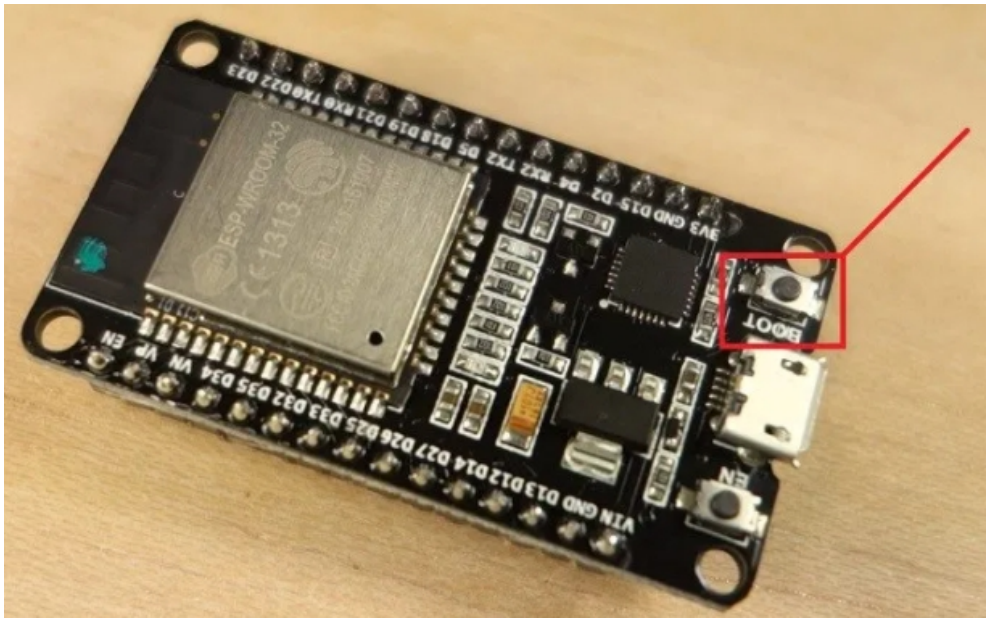


Troubleshooting

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… Connecting…“. It means that your ESP32 is not in flashing/uploading mode.

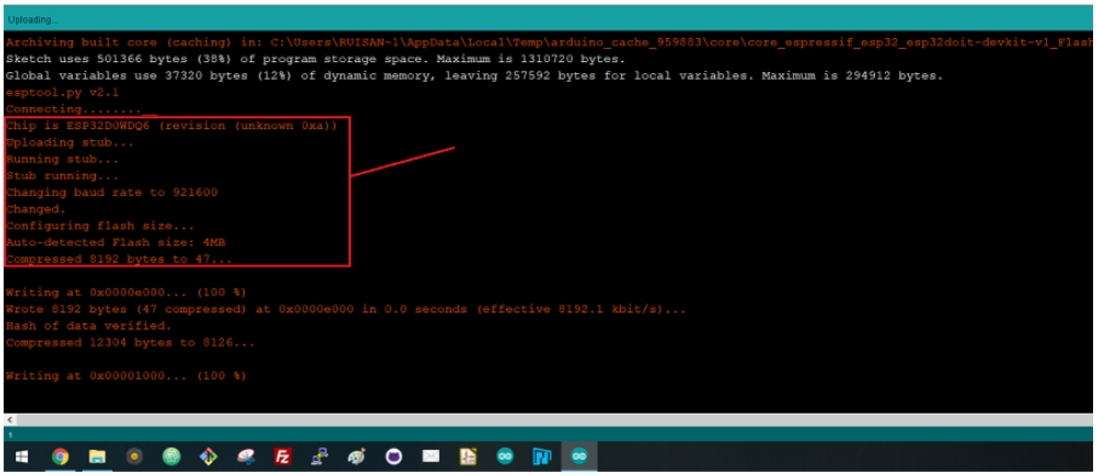
Having the right board name and COM por selected, follow these steps:

- Hold-down the “**BOOT**” button in your ESP32 board



Press the “Upload” button in the Arduino IDE to upload your sketch:

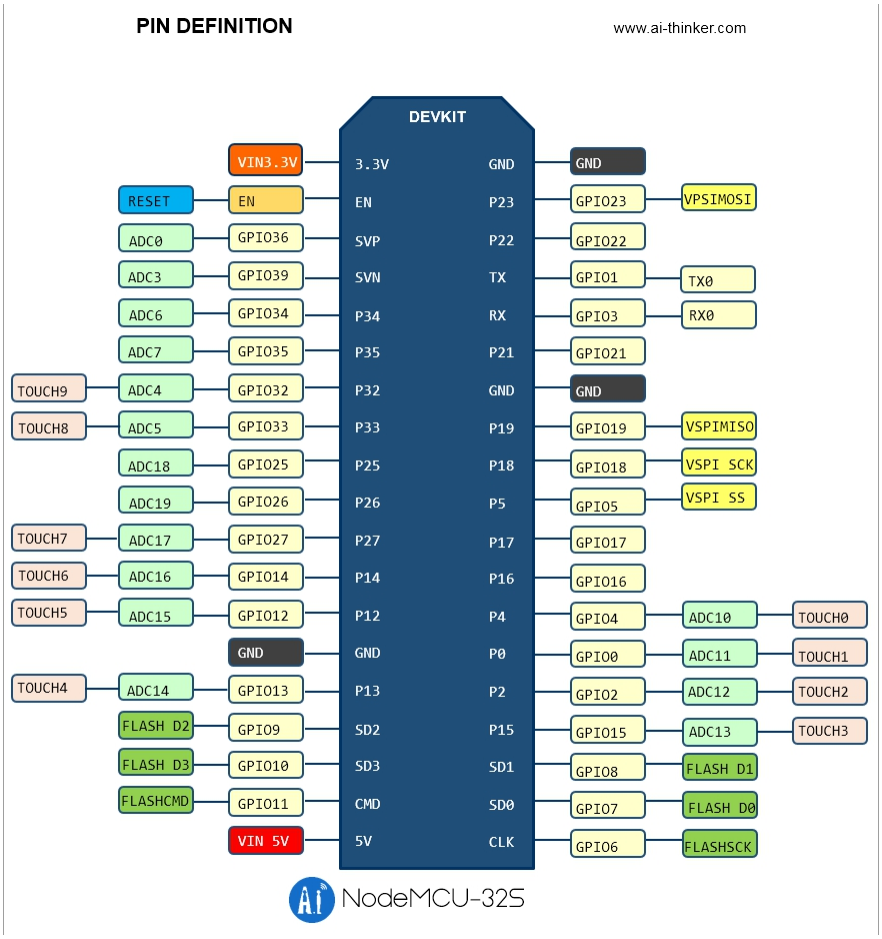
- After you see the “Connecting….” message in your Arduino IDE, release the finger from the “BOOT” button:



- After that, you should see the “Done uploading” message
- That’s it. Your ESP32 should have the new sketch running. Press the “ENABLE” button to restart the ESP32 and run the new uploaded sketch.
- You’ll also have to repeat that button sequence every time you want to upload a new sketch.

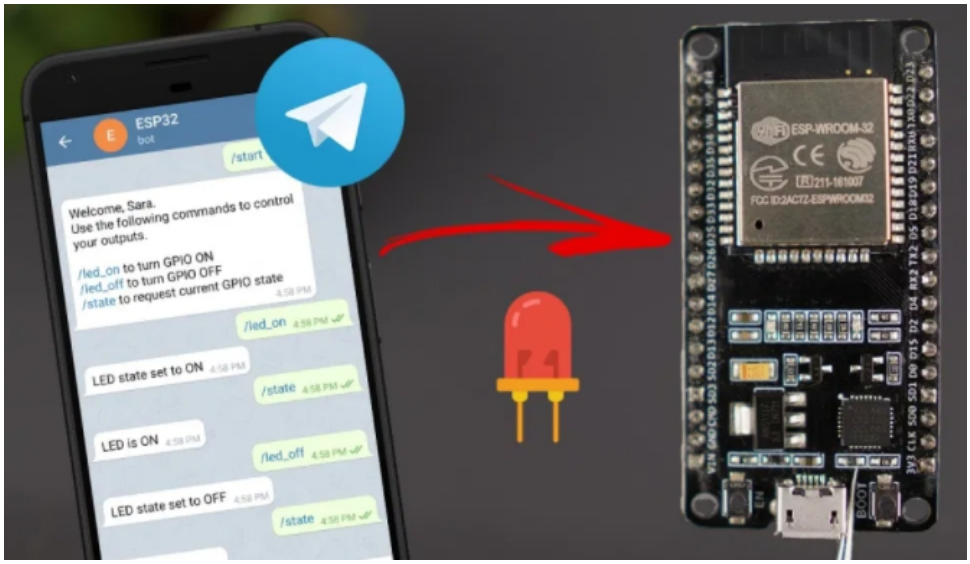
ESP32 Peripherals

The ESP32 peripherals include:
18 Analog-to-Digital Converter (ADC) channels
3 SPI interfaces
3 UART interfaces
2 I2C interfaces
16 PWM output channels
2 Digital-to-Analog Converters (DAC)
2 I2S interfaces
10 Capacitive sensing GPIOs
The ADC (analog to digital converter) and DAC (digital to analog converter) features are assigned to specific static pins. However, you can decide which pins are UART, I2C, SPI, PWM, etc – you just need to assign them in the code. This is possible due to the ESP32 chip's multiplexing feature.
Although you can define the pins properties on the software, there are pins assigned by default as shown in the following figure .



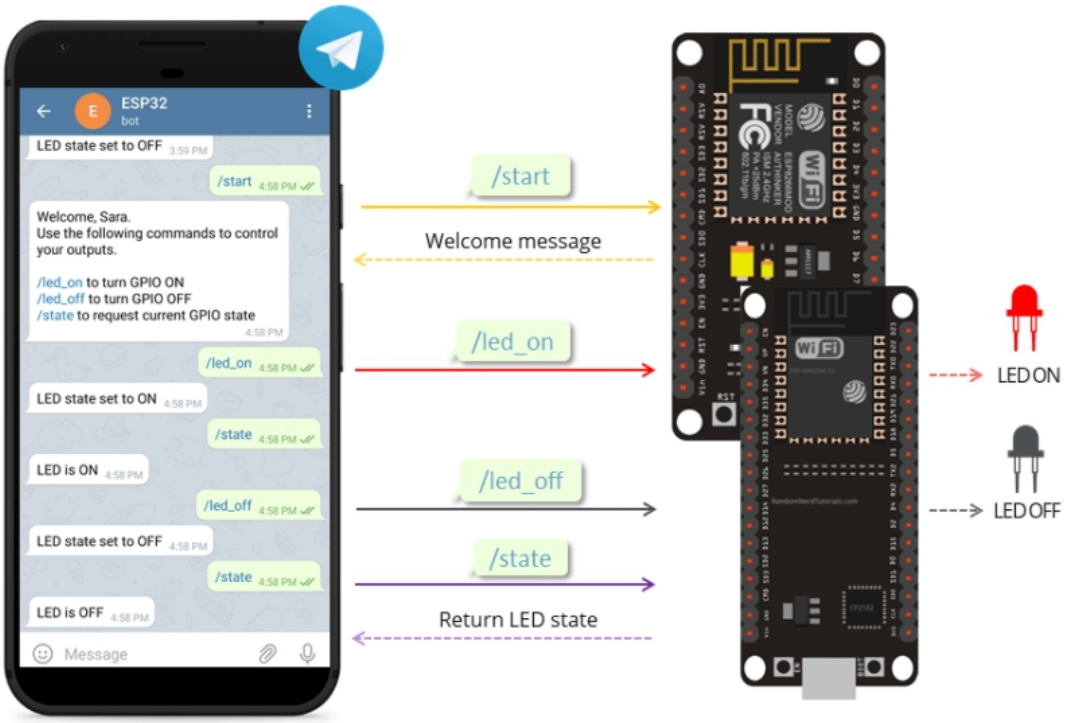
Lesson 1 Telegram: Control ESP32 Outputs

This guide shows how to control the ESP32 or ESP8266 NodeMCU GPIOs from anywhere in the world using Telegram. As an example, we'll control an LED, but you can control any other output. You just need to send a message to your Telegram Bot to set your outputs HIGH or LOW. The ESP boards will be programmed using Arduino IDE.



Project Overview：

In this tutorial we'll build a simple project that allows you to control ESP32 or ESP8266 NodeMCU GPIOs using Telegram. You can also control a relay module.



You'll create a Telegram bot for your ESP32/ESP8266 board;

You can start a conversation with the bot;

When you send the message /led_on to the bot, the ESP board receives the message and turns GPIO 2 on;

Similarly, when you send the message /led_off, it turns GPIO 2 off;

Additionally, you can also send the message /state to request the current GPIO state. When the ESP receives that message, the bot responds with the current GPIO state;

You can send the /start message to receive a welcome message with the commands to control the board.

This is a simple project, but shows how you can use Telegram in your IoT and Home Automation projects. The idea is to apply the concepts learned in your own projects.

Introducing Telegram：

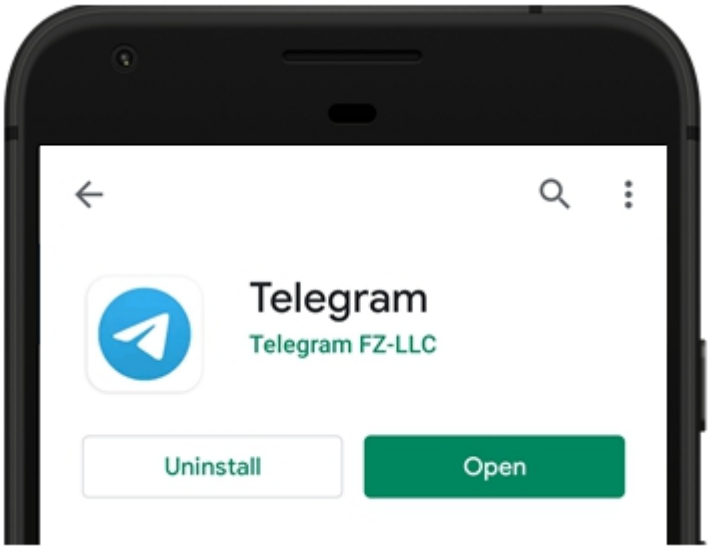
Telegram Messenger is a cloud-based instant messaging and voice over IP service. You can easily install it in your smartphone (Android and iPhone) or computer (PC, Mac and Linux). It is free and without any ads. Telegram allows you to create bots that you can interact with.

“Bots are third-party applications that run inside Telegram. Users can interact with bots by sending them messages, commands and inline requests. You control your bots using HTTPS requests to Telegram Bot API“.

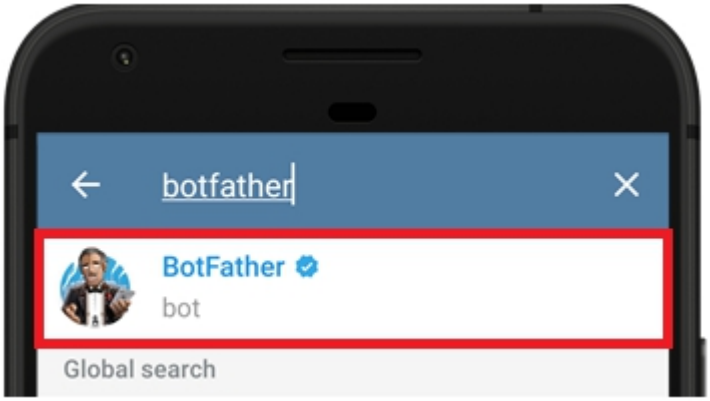
The ESP32/ESP8266 will interact with the Telegram bot to receive and handle the messages, and send responses. In this tutorial you’ll learn how to use Telegram to send messages to your bot to control the ESP outputs from anywhere (you just need Telegram and access to the internet).

Creating a Telegram Bot：

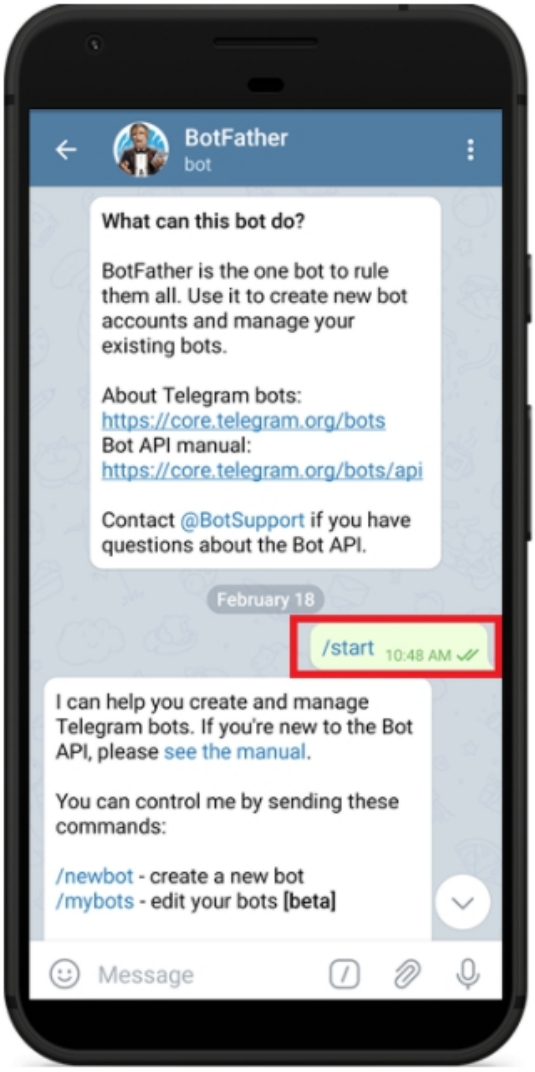
Go to Google Play or App Store, download and install Telegram.



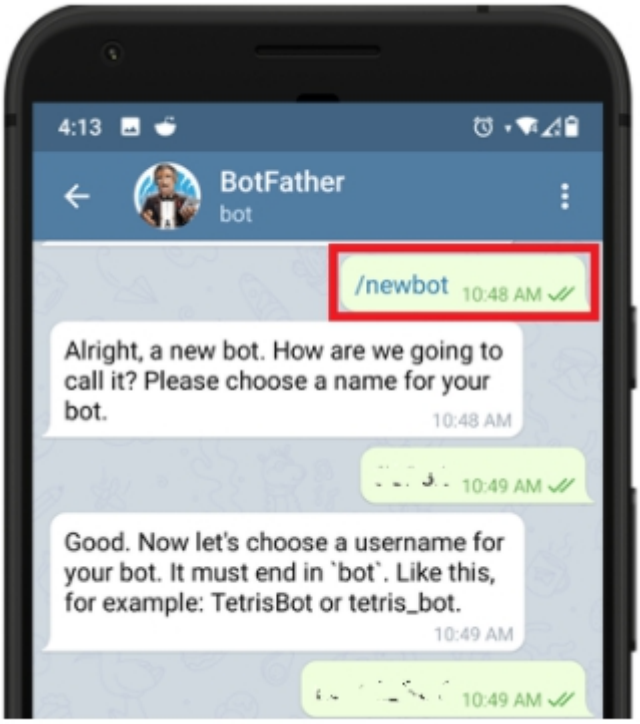
Open Telegram and follow the next steps to create a Telegram Bot. First, search for “botfather” and click the BotFather as shown below. Or open this link t.me/botfather in your smartphone.



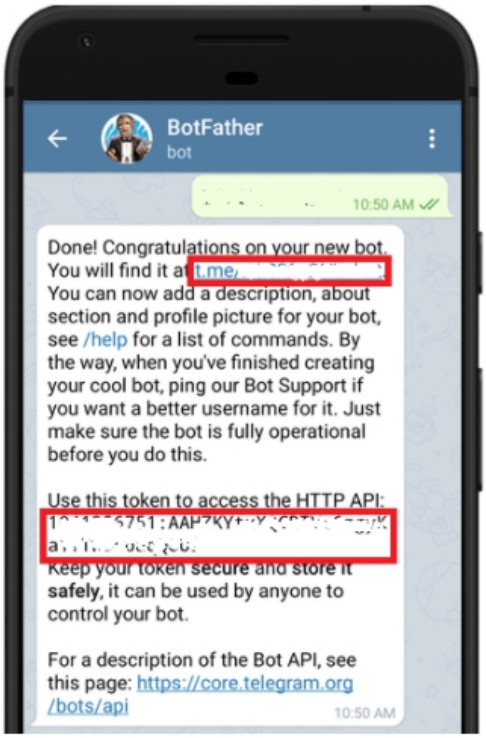
The following window should open and you’ll be prompted to click the start button.



Type /newbot and follow the instructions to create your bot. Give it a name and username.



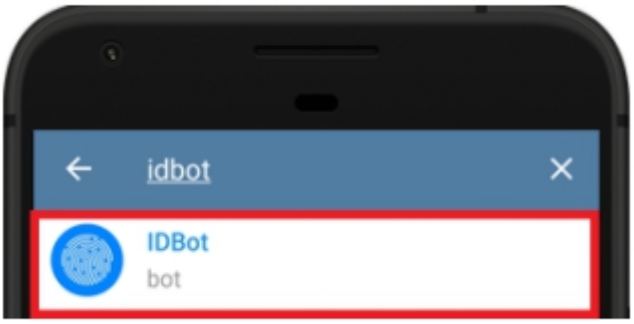
If your bot is successfully created, you'll receive a message with a link to access the bot and the bot token. Save the bot token because you'll need it so that the ESP32/ESP8266 can interact with the bot.



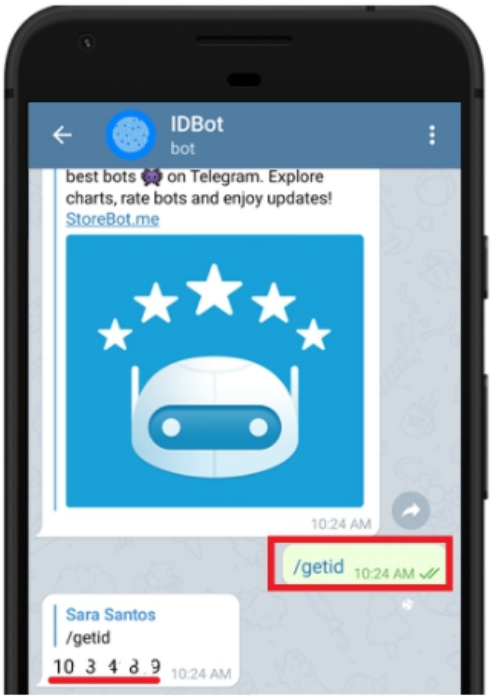
Get Your Telegram User ID

Anyone that knows your bot username can interact with it. To make sure that we ignore messages that are not from our Telegram account (or any authorized users), you can get your Telegram User ID. Then, when your telegram bot receives a message, the ESP can check whether the sender ID corresponds to your User ID and handle the message or ignore it.

In your Telegram account, search for “IDBot” or open this link t.me/myidbot in your smartphone.



Start a conversation with that bot and type /getid. You will get a reply back with your user ID. Save that user ID, because you’ll need it later in this tutorial.



Universal Telegram Bot Library ：

To interact with the Telegram bot, we'll use the **Universal Telegram Bot Library** created by Brian Lough that provides an easy interface for the Telegram Bot API.

Follow the next steps to install the latest release of the library.

Click here to download the Universal Arduino Telegram Bot library.

Go to Sketch > Include Library > Add.ZIP Library...

Add the library you've just downloaded.

And that's it. The library is installed.

Important: don't install the library through the Arduino Library Manager because it might install a deprecated version.

For all the details about the library, take a look at the Universal Arduino Telegram Bot Library GitHub page.

ArduinoJson Library

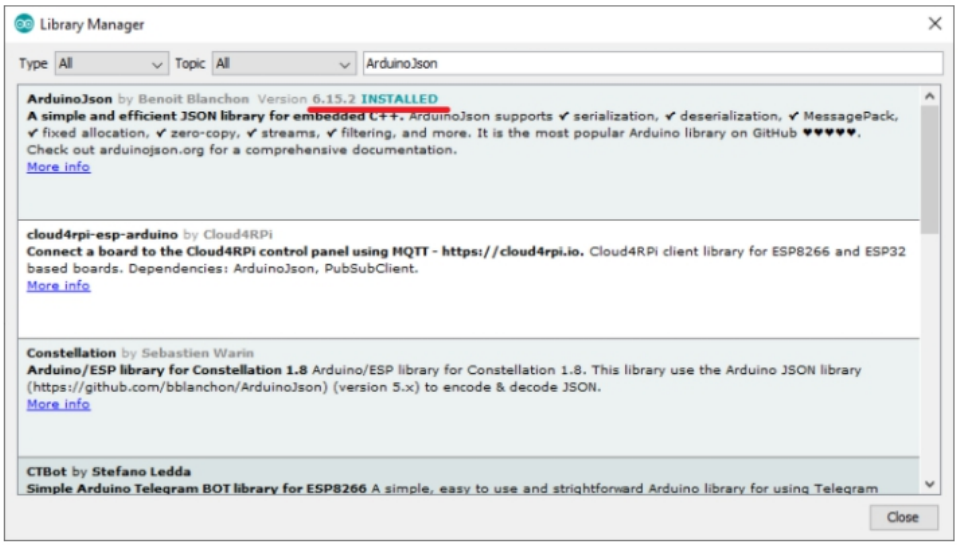
You also have to install the ArduinoJson library. Follow the next steps to install the library.

Go to Skech > Include Library > Manage Libraries.

Search for "ArduinoJson".

Install the library.

We're using ArduinoJson library version 6.15.2.



Control Outputs using Telegram – ESP32/ESP8266 Sketch：

The following code allows you to control your ESP32 or ESP8266 NodeMCU GPIOs by sending messages to a Telegram Bot. To make it work for you, you need to insert your network credentials (SSID and password), the Telegram Bot Token and your Telegram User ID.

Code：

```
#ifdef ESP32
  #include <WiFi.h>
#else
  #include <ESP8266WiFi.h>
#endif
#include <WiFiClientSecure.h>
#include <UniversalTelegramBot.h>  // Universal Telegram Bot Library written by Brian Lough: https://github.com/witnessmenow/Universal-Arduino-Telegram-Bot
#include <ArduinoJson.h>

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

// Initialize Telegram BOT
```

```
#define  BOTtoken  "xxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

// your Bot Token (Get from Botfather)


// Use @myidbot to find out the chat ID of an individual or a group

// Also note that you need to click "start" on a bot before it can

// message you

#define CHAT_ID "1461852131"


WiFiClientSecure client;

UniversalTelegramBot bot(BOTtoken, client);


// Checks for new messages every 1 second.

int botRequestDelay = 1000;

unsigned long lastTimeBotRan;


const int ledPin =2;

bool ledState = LOW;


// Handle what happens when you receive new messages

void handleNewMessages(int numNewMessages) {

    Serial.println("handleNewMessages");

    Serial.println(String(numNewMessages));
```

```
  for (int i=0; i<numNewMessages; i++) {
    // Chat id of the requester
    String chat_id = String(bot.messages[i].chat_id);
    if (chat_id != CHAT_ID){
      bot.sendMessage(chat_id, "Unauthorized user", "");
      continue;
    }

    // Print the received message
    String text = bot.messages[i].text;
    Serial.println(text);

    String from_name = bot.messages[i].from_name;

    if (text == "/start") {
      String welcome = "Welcome, " + from_name + ".\n";
      welcome += "Use the following commands to control your outputs.\n\n";
      welcome += "/led_on to turn GPIO ON \n";
      welcome += "/led_off to turn GPIO OFF \n";
      welcome += "/state to request current GPIO state \n";
```

```
    bot.sendMessage(chat_id, welcome, "");
}

if (text == "/led_on") {
    bot.sendMessage(chat_id, "LED state set to ON", "");
    ledState = HIGH;
    digitalWrite(ledPin, ledState);
}

if (text == "/led_off") {
    bot.sendMessage(chat_id, "LED state set to OFF", "");
    ledState = LOW;
    digitalWrite(ledPin, ledState);
}

if (text == "/state") {
    if (digitalRead(ledPin)){
        bot.sendMessage(chat_id, "LED is ON", "");
    }
    else{
        bot.sendMessage(chat_id, "LED is OFF", "");
    }
```

```
    }
  }
}

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

  #ifdef ESP32
    client.setInsecure();
  #endif

  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, ledState);

  // Connect to Wi-Fi
  WiFi.mode(WIFI_STA);
  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());
}

void loop() {
    if (millis() > lastTimeBotRan + botRequestDelay)    {
        int numNewMessages = bot.getUpdates(bot.last_message_received
+ 2);

        while(numNewMessages) {
            Serial.println("got response");
            handleNewMessages(numNewMessages);
            numNewMessages  =  bot.getUpdates(bot.last_message_received
+ 2);
        }
        lastTimeBotRan = millis();
    }
}
```

How the Code Works：

This sections explain how the code works.
Start by importing the required libraries.

```
#ifdef ESP32

  #include <WiFi.h>#else

  #include <ESP8266WiFi.h>

#endif#include <WiFiClientSecure.h>

#include <UniversalTelegramBot.h>

#include <ArduinoJson.h>
```

Network Credentials
Insert your network credentials in the following variables.

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

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

Define Output
Set the GPIO you want to control. In our case, we'll control GPIO 2 (built-in LED) and its state is LOW by default.

```
const int ledPin = 2;

bool ledState = LOW;
```

Note: if you're using an ESP8266, the built-in LED works with inverted logic. So, you should send a LOW signal to turn the LED on and a HIGH signal to turn it off.

Telegram Bot Token
Insert your Telegram Bot token you've got from Botfather on the BOTtoken variable.

```
#define BOTtoken "XXXXXXXXXX:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  //
your Bot Token (Get from Botfather)
```

Telegram User ID

Insert your chat ID. The one you've got from the IDBot.

```
#define CHAT_ID "XXXXXXXXXX"
```

Create a new WiFi client with WiFiClientSecure.

```
WiFiClientSecure client;
```

Create a bot with the token and client defined earlier.

```
UniversalTelegramBot bot(BOTtoken, client);
```

The botRequestDelay and lastTimeBotRan are used to check for new Telegram messages every x number of seconds. In this case, the code will check for new messages every second (1000 milliseconds). You can change that delay time in the botRequestDelay variable.

```
int botRequestDelay = 1000;

unsigned long lastTimeBotRan;
```

handleNewMessages()

The handleNewMessages() function handles what happens when new messages arrive.

```
void handleNewMessages(int numNewMessages) {

  Serial.println("handleNewMessages");

  Serial.println(String(numNewMessages));
```

It checks the available messages:

```
for (int i=0; i<numNewMessages; i++) {
```

Get the chat ID for that particular message and store it in the chat_id variable. The chat ID allows us to identify who sent the message.

```
String chat_id = String(bot.messages[i].chat_id);
```

If the chat_id is different from your chat ID (CHAT_ID), it means that someone (that is not you) has sent a message to your bot. If that's the case, ignore the message and wait for the next message.

```
if (chat_id != CHAT_ID) {
  bot.sendMessage(chat_id, "Unauthorized user", "");
  continue;}
```

Otherwise, it means that the message was sent from a valid user, so we'll save it in the text variable and check its content.

```
String text = bot.messages[i].text;
Serial.println(text);
```

The from_name variable saves the name of the sender.

```
String from_name = bot.messages[i].from_name;
```

If it receives the /start message, we'll send the valid commands to control the ESP32/ESP8266. This is useful if you happen to forget what are the commands to control your board.

```
if (text == "/start") {
  String welcome = "Welcome, " + from_name + ".\n";
  welcome += "Use the following commands to control your outputs.\n\n";
  welcome += "/led_on to turn GPIO ON \n";
```

```
  welcome += "/led_off to turn GPIO OFF \n";

  welcome += "/state to request current GPIO state \n";

  bot.sendMessage(chat_id, welcome, "");}
```

Sending a message to the bot is very simply. You just need to use the sendMessage() method on the bot object and pass as arguments the recipient's chat ID, the message, and the parse mode.

```
bool sendMessage(String chat_id, String text, String parse_mode = "")
```

In our particular example, we'll send the message to the ID stored on the chat_id variable (that corresponds to the person who've sent the message) and send the message saved on the welcome variable.

```
bot.sendMessage(chat_id, welcome, "");
```

If it receives the /led_on message, turn the LED on and send a message confirming we've received the message. Also, update the ledState variable with the new state.

```
if (text == "/led_on") {

  bot.sendMessage(chat_id, "LED state set to ON", "");

  ledState = HIGH;

  digitalWrite(ledPin, ledState);}
```

Do something similar for the /led_off message.

```
if (text == "/led_off") {

  bot.sendMessage(chat_id, "LED state set to OFF", "");

  ledState = LOW;

  digitalWrite(ledPin, ledState);}
```

Note: if you're using an ESP8266, the built-in LED works with inverted logic. So, you should send a LOW signal to turn the LED on and a HIGH signal to turn it off.

Finally, if the received message is /state, check the current GPIO state and send a message accordingly.

```
if (text == "/state") {
  if (digitalRead(ledPin)){
    bot.sendMessage(chat_id, "LED is ON", "");
  }
  else{
    bot.sendMessage(chat_id, "LED is OFF", "");
  }}
```

setup()

In the setup(), initialize the Serial Monitor.

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

If you're using the ESP8266, you need to use the following line:

```
#ifdef ESP8266
  client.setInsecure();
#endif
```

In the library examples for the ESP8266 they say: "This is the simplest way of getting this working. If you are passing sensitive information, or controlling something important, please either use certStore or at least client.setFingerPrint".

Set the LED as an output and set it to LOW when the ESP first starts:

```
pinMode(ledPin, OUTPUT);

digitalWrite(ledPin, ledState);
```

Init Wi-Fi
Initialize Wi-Fi and connect the ESP to your local network with the SSID and password defined earlier.

```
WiFi.mode(WIFI_STA);

WiFi.begin(ssid, password);while (WiFi.status() != WL_CONNECTED) {

  delay(1000);

  Serial.println("Connecting to WiFi..");}
```

loop()
In the loop(), check for new messages every second.

```
void loop() {

  if (millis() > lastTimeBotRan + botRequestDelay)  {

    int numNewMessages = bot.getUpdates(bot.last_message_received + 1);



    while(numNewMessages) {

      Serial.println("got response");

      handleNewMessages(numNewMessages);

      numNewMessages = bot.getUpdates(bot.last_message_received + 1);

    }

    lastTimeBotRan = millis();

  }}
```

When a new message arrives, call the handleNewMessages function.

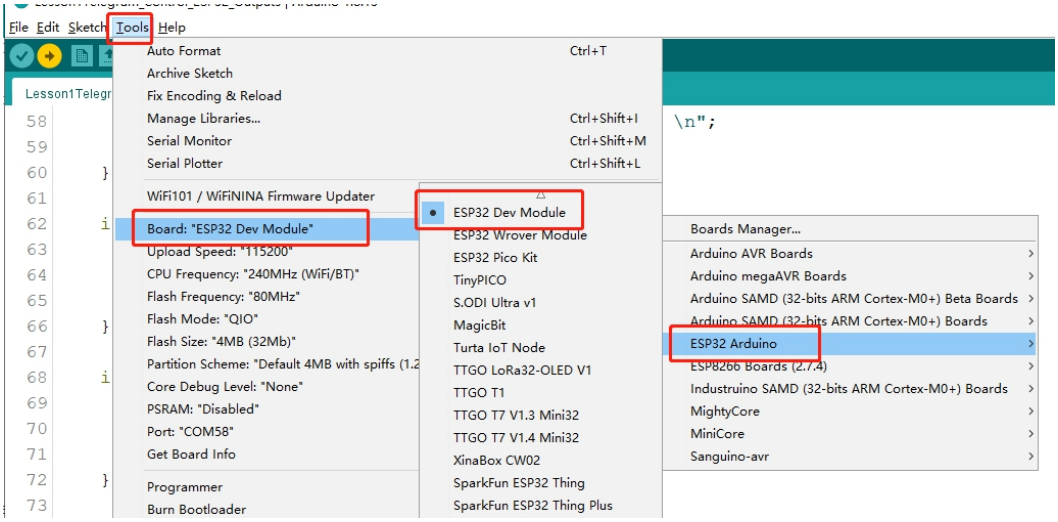
```
while(numNewMessages) {

  Serial.println("got response");

  handleNewMessages(numNewMessages);

  numNewMessages = bot.getUpdates(bot.last_message_received + 1);}
```

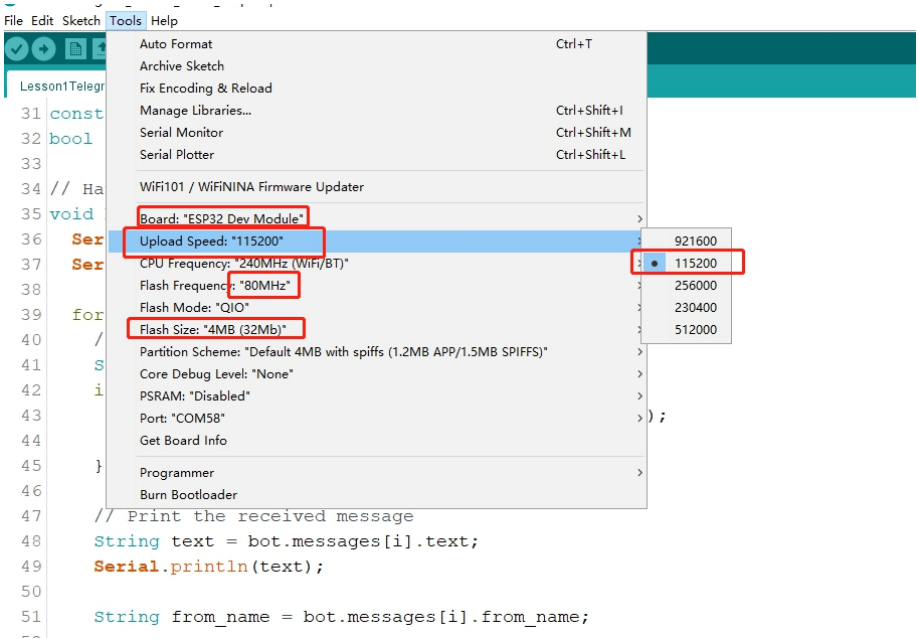
That’s pretty much how the code works.

Demonstration：

Upload the code to your ESP32 or ESP8266 board. Don’t forget to go to Tools > Board and select the board you’re using. Go to Tools > Port and select the COM port your board is connected to.

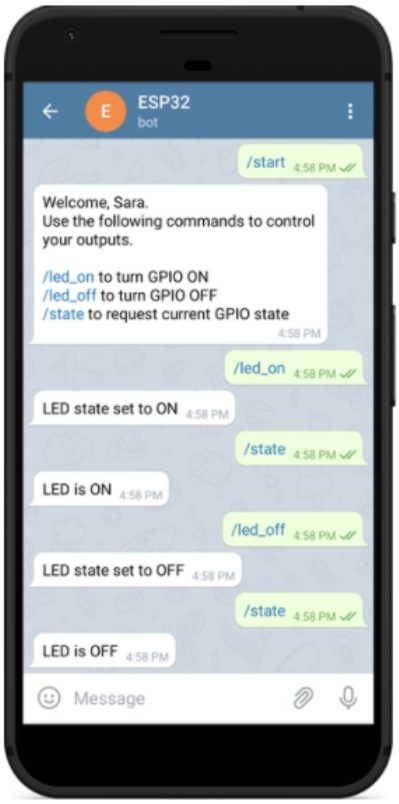
After uploading the code, press the ESP32/ESP8266 on-board EN/RST button so that it starts running the code. Then, you can open the Serial Monitor to check what’s happening in the background.



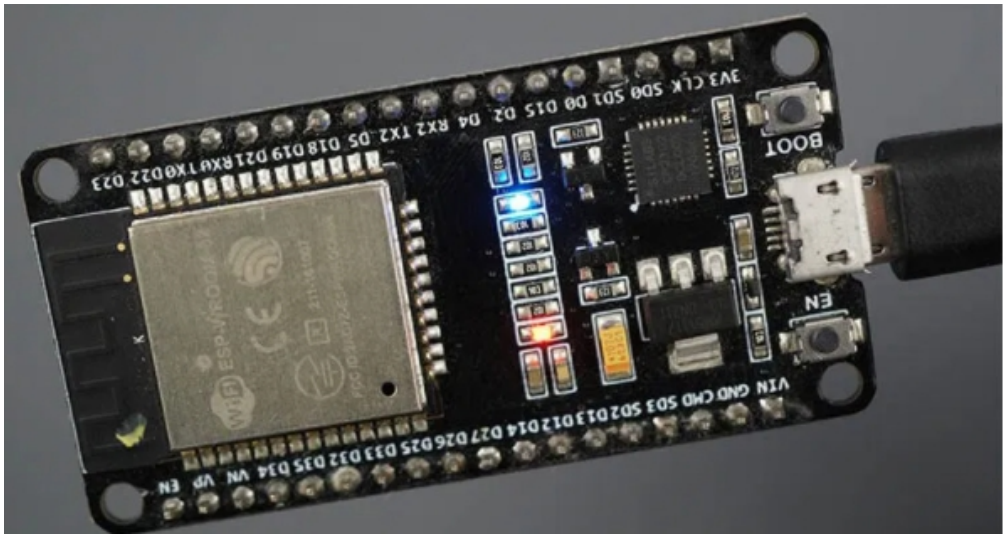


Go to your Telegram account and open a conversation with your bot. Send the following commands and see the bot responding:

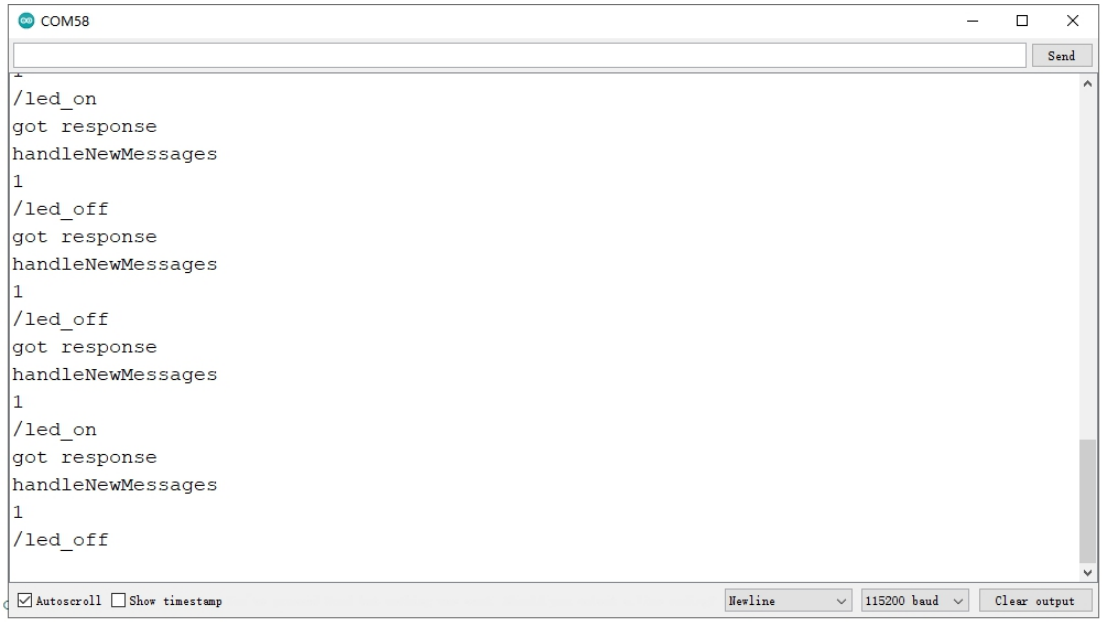
/start shows the welcome message with the valid commands.
/led_on turns the LED on.
/led_off turns the LED off.
/state requests the current LED state.



The on-board LED should turn on and turn off accordingly (the ESP8266 on-board LED works in reverse, it's off when you send /led_on and on when you send /led_off).

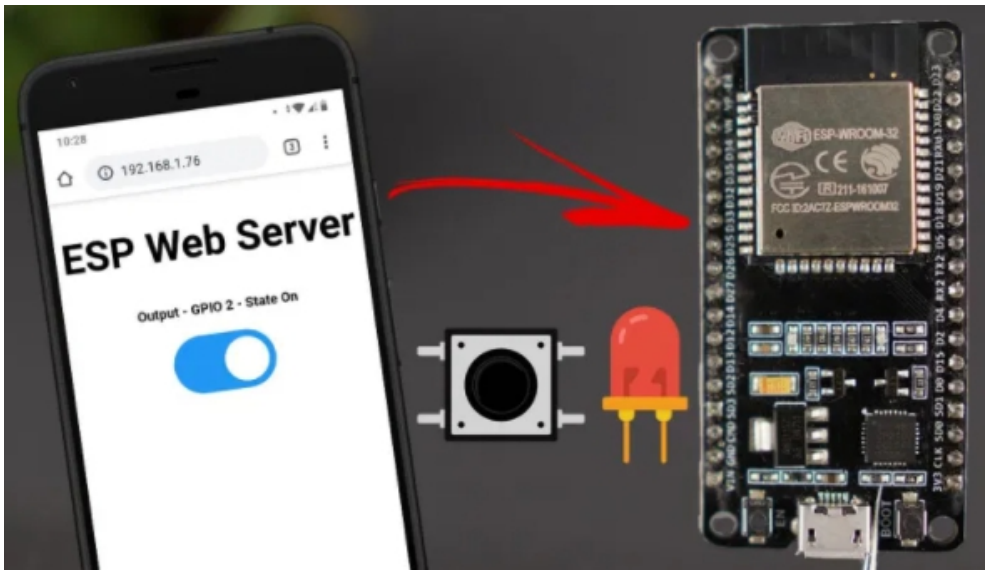


At the same time, on the Serial Monitor you should see that the ESP is receiving the messages.



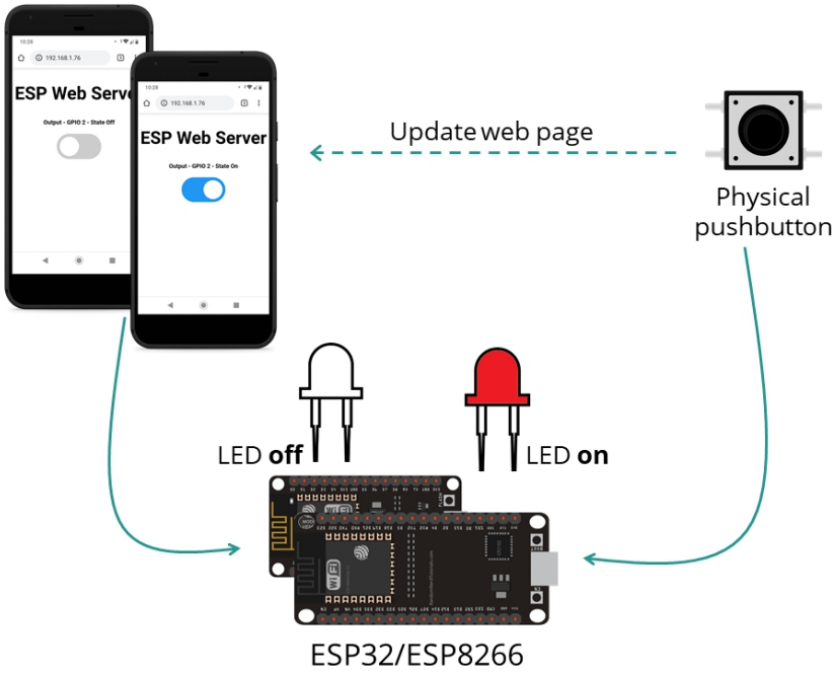
Lesson2 ESP32: Control Outputs with Web Server and a Physical Button Simultaneously

This tutorial shows how to control the ESP32 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 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 pushbutton that controls the same output;
- If you change the output state using the physical puhsbutton, its current state is also updated on the web server.

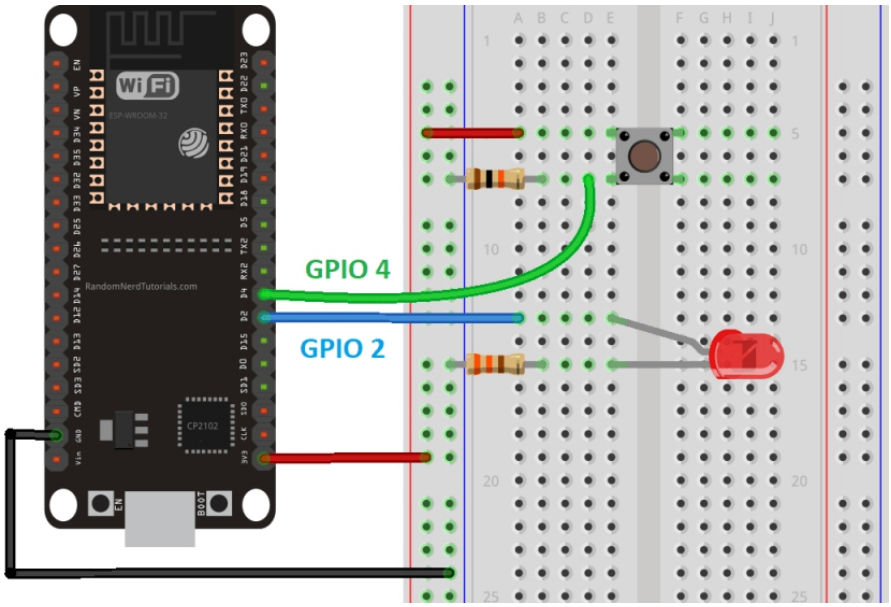
In summary, this project allows you to control the same output using a web server and a push button simultaneously. Whenever the output state changes, the web server is updated.

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

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

ESP32 Schematic

Note: The pin interface of the board should be checked with the actual board interface before connecting and energizing.



Installing Libraries – Async Web Server：

To build the web server you need to install the following libraries:

ESP32: install the ESPAsyncWebServer and the AsyncTCP libraries.

ESP8266: install the ESPAsyncWebServer and the ESPAsyncTCP libraries.

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.

Code:

```
// Import required libraries#ifdef ESP32

  #include <WiFi.h>

  #include <AsyncTCP.h>#else

  #include <ESP8266WiFi.h>

  #include <ESPAsyncTCP.h>#endif#include <ESPAsyncWebServer.h>

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

const char* PARAM_INPUT_1 = "state";

const int output = 2;const int buttonPin = 4;

// Variables will change:int ledState = LOW;          // the current state
of the output pinint buttonState;             // the current reading from
the input pinint 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 toggledunsigned long debounceDelay = 50;    // the debounce time;
increase if the output flickers
```

```
// 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">

  <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: 34px}

    .slider:before {position: absolute; content: ""; height: 52px; width:
52px; left: 8px; bottom: 8px; background-color: #fff;
-webkit-transition: .4s; transition: .4s; border-radius: 68px}

    input:checked+.slider {background-color: #2196F3}

    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?state=1", true); }
  else { xhr.open("GET", "/update?state=0", true); }
  xhr.send();}
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 ) ;</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 ="";

    String outputStateValue = outputState();

    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>";

    return buttons;

  }

  return String();}



String outputState(){

  if(digitalRead(output)){

    return "checked";

  }

  else {

    return "";

  }

  return "";}

void setup(){
```

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

pinMode(output, OUTPUT);
digitalWrite(output, LOW);
pinMode(buttonPin, INPUT);

// 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?state=<inputMessage>
```

```
  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 message sent";
      inputParam = "none";
    }
    Serial.println(inputMessage);
    request->send(200, "text/plain", "OK");
  });

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

```
// Start server
server.begin();}
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;
      }
    }
  }


  // set the LED:
  digitalWrite(output, ledState);


  // save the reading. Next time through the loop, it'll be the
lastButtonState:
  lastButtonState = reading;}
```

You only need to enter the network credentials (SSID and password), and the web server will work immediately. This code is compatible with ESP32 and ESP8266 boards and controls GPIO 2-you can change the code to control any other GPIO.

How the Code Works

We've already explained in great details how web servers like this work in previous tutorials (DHT Temperature Web Server), so we'll just take a look at the relevant parts for this project.

Network Credentials

As said previously, you need to insert your network credentials in the following lines:

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

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 pinint
lastButtonState = LOW;  // 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 the outputState() 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, the toggleCheckbox() 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 message sent";
    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()

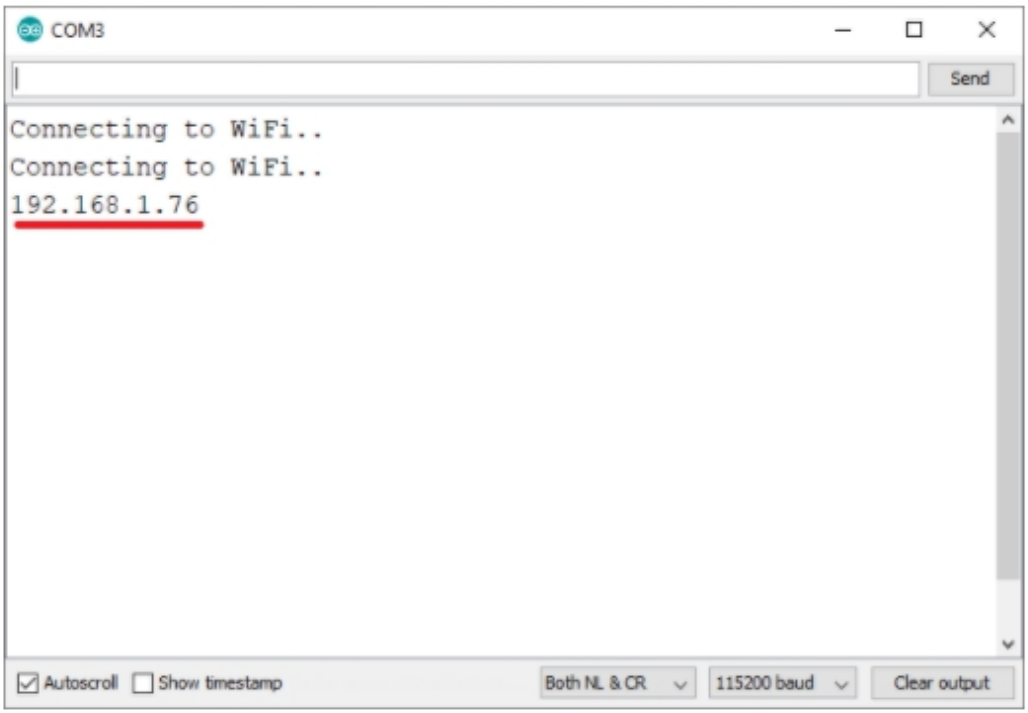
In the loop(), we debounce the pushbutton and turn the LED on or off depending on the value of the ledState variable.

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

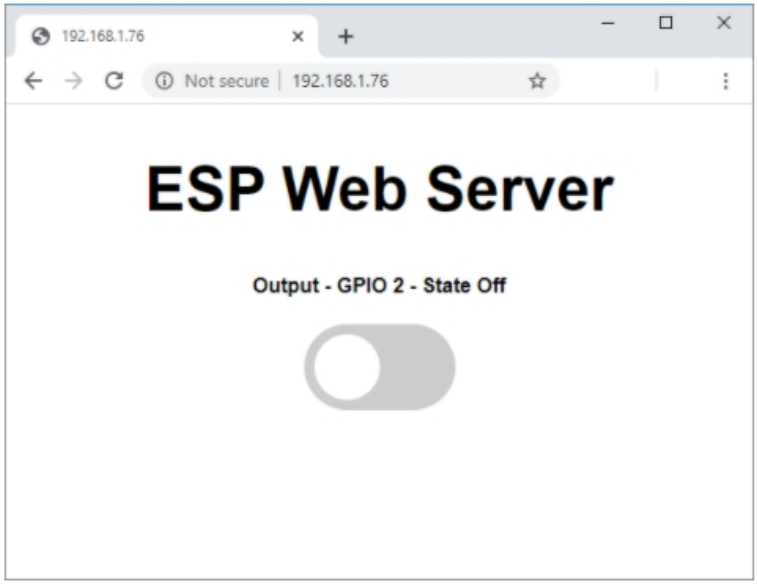
Demonstration

Upload the code to your ESP32 or ESP8266 board.

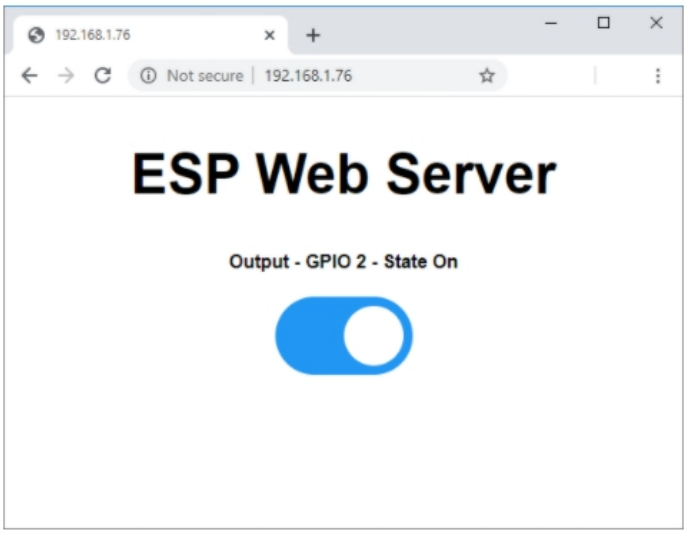
Then, open the Serial Monitor at a baud rate of 115200. Press the on-board EN/RST button to get is IP address.



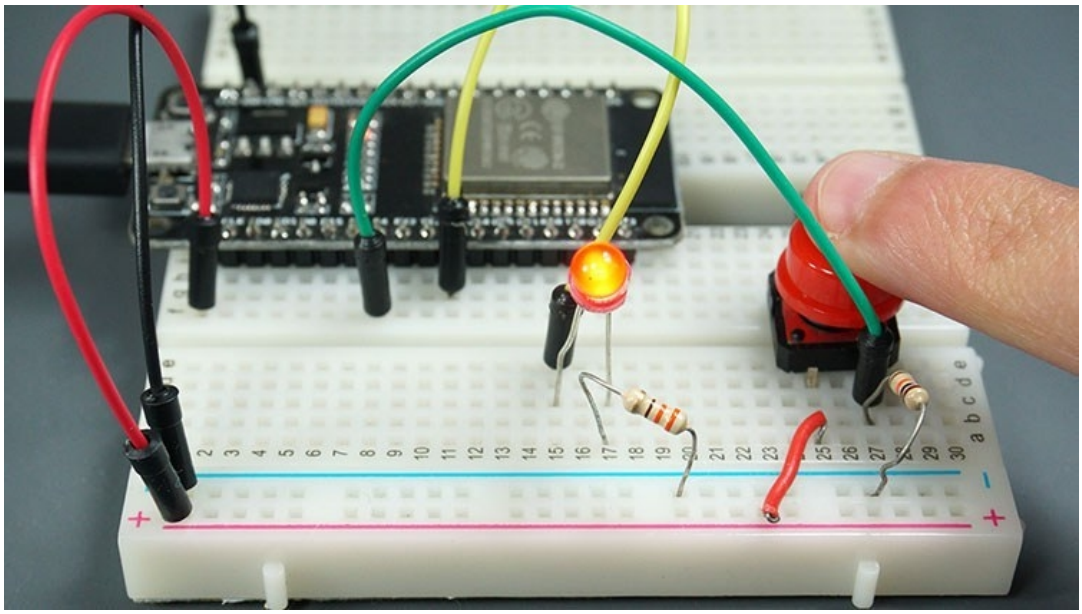
Open a browser on your local network, and type the ESP IP address. You should have access to the web server as shown below.



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



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

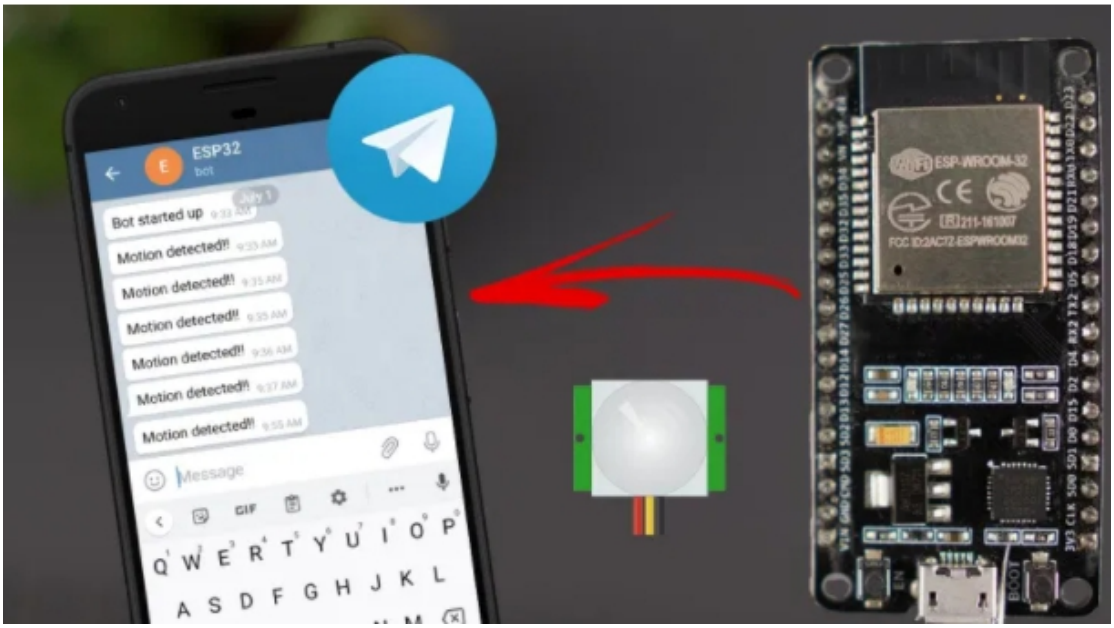


Wrapping Up

In this tutorial you've learned how to control ESP32/ESP8266 outputs with a web server and a physical button at the same time. The output state is always updated whether it is changed via web server or with the physical button.

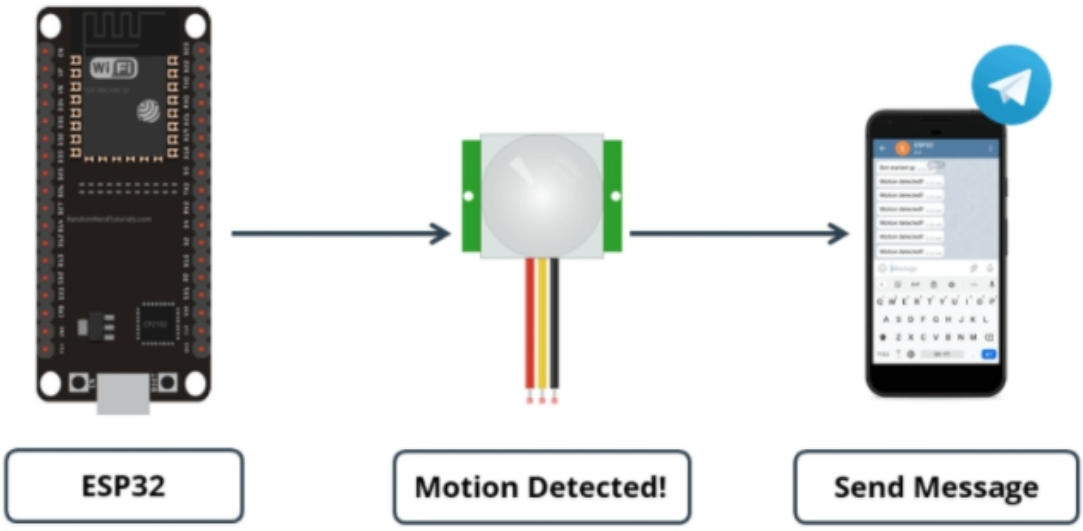
Lesson 3 Telegram: ESP32 Motion Detection with Notifications

This tutorial shows how to send a notification to your Telegram account when the ESP32 detects motion. As long as you can use your smartphone to access the Internet, you will receive notifications no matter where you are. The ESP board will be programmed using the Arduino IDE.



Project Overview:

This tutorial explains how to get notifications in your Telegram account when the ESP32 detects motion.



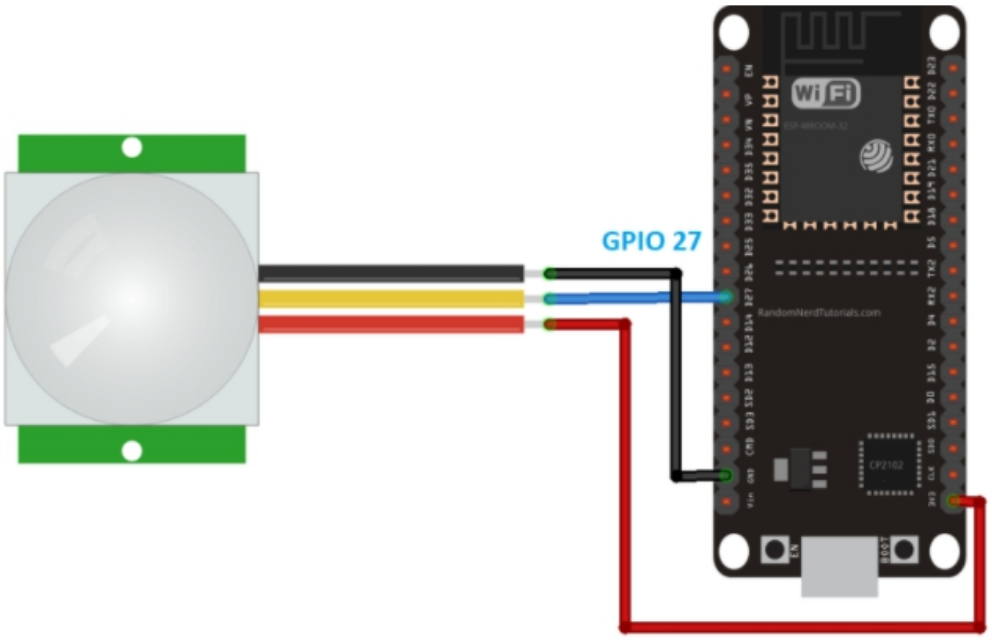
An overview of how the project works:

You will create a Telegram bot for ESP32.
ESP32 is connected to the PIR motion sensor.
When the sensor detects movement, the ESP32 will send a warning message to your Telegram account.
Whenever motion is detected, you will be notified in your Telegram account.
This is a simple project, but shows how to use Telegram in IoT and home automation projects. The idea is to apply the concepts learned in your own project to the project.

The introduction about telegram has been described in the first section of the project, and the specific operations can be viewed by referring to the first section of the course by turning the page.

Schematic diagram of the project:

For this project, you need to connect the PIR motion sensor to the ESP32 board. Please follow the next schematic.



In this example, we connect the PIR motion sensor data pin to GPIO 27. You can use any other suitable GPIO.

Telegram motion detection with notification-ESP32 code:

Whenever motion is detected, the following code will use your Telegram bot to send a warning message to your Telegram account. In order to make this sketch work for you, you need to insert network credentials (SSID and password), Telegram Bot token and Telegram user ID.

```
#include <WiFi.h>

#include <WiFiClientSecure.h>

#include <UniversalTelegramBot.h>

#include <ArduinoJson.h>// Replace with your network credentialsconst
char* ssid = "REPLACE_WITH_YOUR_SSID";

const char* password = "REPLACE_WITH_YOUR_PASSWORD";// Initialize
Telegram BOT#define BOTtoken
"XXXXXXXXXX:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  // your Bot Token
(Get from Botfather)// Use @myidbot to find out the chat ID of an
individual or a group// Also note that you need to click "start" on a
bot before it can// message you

#define CHAT_ID "XXXXXXXXXX"

WiFiClientSecure client;

UniversalTelegramBot bot(BOTtoken, client);

const int motionSensor = 27; // PIR Motion Sensor

bool motionDetected = false;// Indicates when motion is detectedvoid
IRAM_ATTR detectsMovement() {

  //Serial.println("MOTION DETECTED!!!");

  motionDetected = true;}void setup() {

  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);

  // Attempt to connect to Wifi network:

  Serial.print("Connecting Wifi: ");

  Serial.println(ssid);

  WiFi.mode(WIFI_STA);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {

    Serial.print(".");

    delay(500);

  }

  Serial.println("");

  Serial.println("WiFi connected");

  Serial.print("IP address: ");

  Serial.println(WiFi.localIP());

  bot.sendMessage(CHAT_ID, "Bot started up", "");}void loop() {

  if(motionDetected){

    bot.sendMessage(CHAT_ID, "Motion detected!!", "");

    Serial.println("Motion Detected");

    motionDetected = false;
```

```
}}
```

Code work explanation:

First import the required libraries.

```
#include <WiFi.h>

#include <WiFiClientSecure.h>

#include <UniversalTelegramBot.h>

#include <ArduinoJson.h>
```

Network credentials

Insert your network credentials in the following variables.

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

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

Telegram bot token

Insert the Telegram Bot token you got from Botfather into the BOT token.

```
#define BOTtoken "XXXXXXXXXX:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"  //
your Bot Token (Get from Botfather)
```

Telegram User ID

Insert your chat ID. The one you got from IDBot.

```
#define CHAT_ID "XXXXXXXXXX"
```

Use the following method to create a new WiFi client WiFiClientSecure.

```
WiFiClientSecure client;
```

Create a bot and use the token and client defined earlier.

```
UniversalTelegramBot bot(BOTtoken, client);
```

Motion sensor:

Define the GPIO to which the motion sensor is connected.

```
const int motionSensor = 27; // PIR Motion Sensor
```

This motionDetected Boolean variable is used to indicate whether motion is detected. Set to the wrong default.

```
bool motionDetected = false;
```

detectorMovement（）

The detectsmovement() function is a callback function that will be executed when motion is detected. In this case, it just changes the state motionDetected to true.

```
void IRAM_ATTR detectsMovement() {

  //Serial.println("MOTION DETECTED!!!");

  motionDetected = true;}
```

Setup()

Inside setup(), initialize the serial monitor.

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

PIR motion sensor interrupt

Set the PIR motion sensor to interrupt and set the detectorMovement() as a callback function (when motion is detected, this function will be executed):

```
// 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);
```

Initialize Wi-Fi

Initialize Wi-Fi and use the previously defined SSID and password to connect ESP32 to your local network.

```
WiFi.mode(WIFI_STA);

WiFi.begin(ssid, password);while (WiFi.status() != WL_CONNECTED) {

  delay(1000);

  Serial.println("Connecting to WiFi..");}
```

Finally, send a message to indicate that the bot has started:

```
bot.sendMessage(CHAT_ID, "Bot started up", "");
```

Loop()

In the loop(), the detection state of motionDetected is changeable.

```
void loop() {

  if(motionDetected){
```

If it is true, it means motion is detected. Therefore, send a message to your Telegram account stating that movement has been detected.

```
bot.sendMessage(CHAT_ID, "Motion detected!!", "");
```

Sending a message to the robot is very simple. You only need to use the method of sending a message () Robot object, and pass the recipient's chat ID, message and parsing mode as parameters.

```
bool sendMessage(String chat_id, String text, String parse_mode = "")
```

Finally, after sending the message, setting motionDetected can become wrong, so it can detect motion again.

```
motionDetected = false;
```

This is pretty much how the code works.

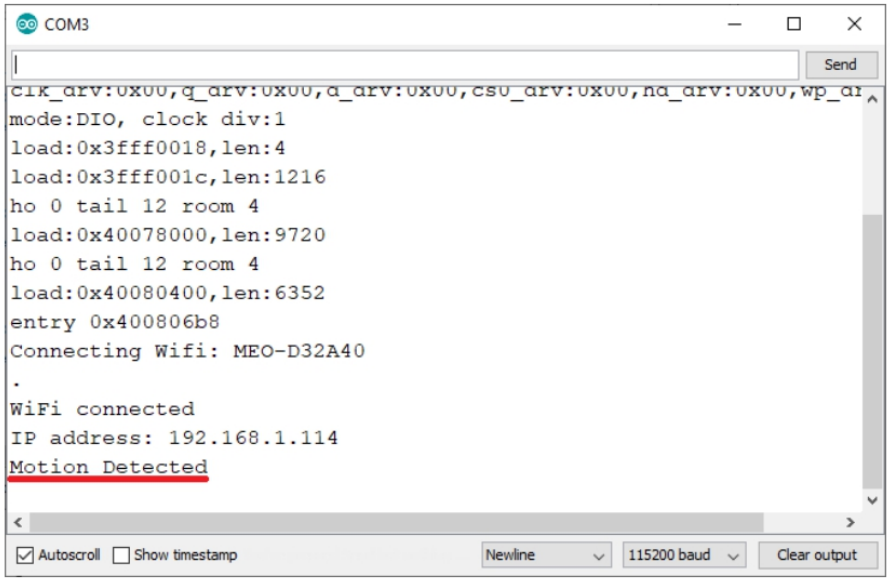
Example:

Upload the code to your ESP32 development board. Don't forget to go to "Tools"> "Development Board" and select the development board you are using. Go to "Tools"> "Port", and then select the COM port your motherboard is connected to.

After uploading the code, please press the onboard EN/RST button of ESP32 to make it start to run the code. You can then open the serial monitor to check what is happening in the background.

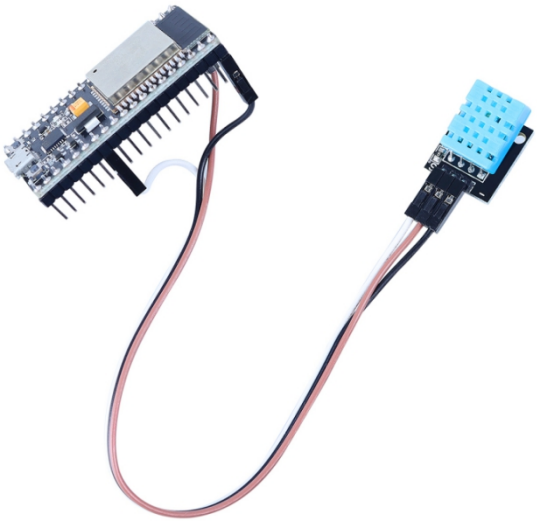
When your motherboard boots up for the first time, it will send a message to your Telegram account: "Boot has been activated". Then, move your hand to the front of the PIR motion sensor and check if you have received a notification that motion has been detected.

At the same time, this is what you should get on the serial monitor.



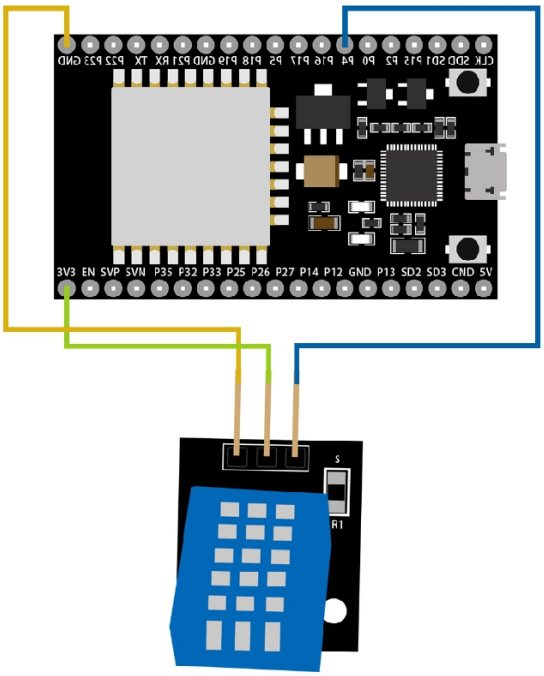
Lesson 4 Use ESP32 with DHT11 temperature and humidity sensor module

This tutorial introduces how to use DHT11 temperature and humidity sensor with ESP32 using Arduino IDE. We will quickly introduce these sensors, pinouts, wiring diagrams, and finally the Arduino sketches.



schematic diagram:

Connect the DHT11 sensor to the ESP32 development board as shown in the figure below.



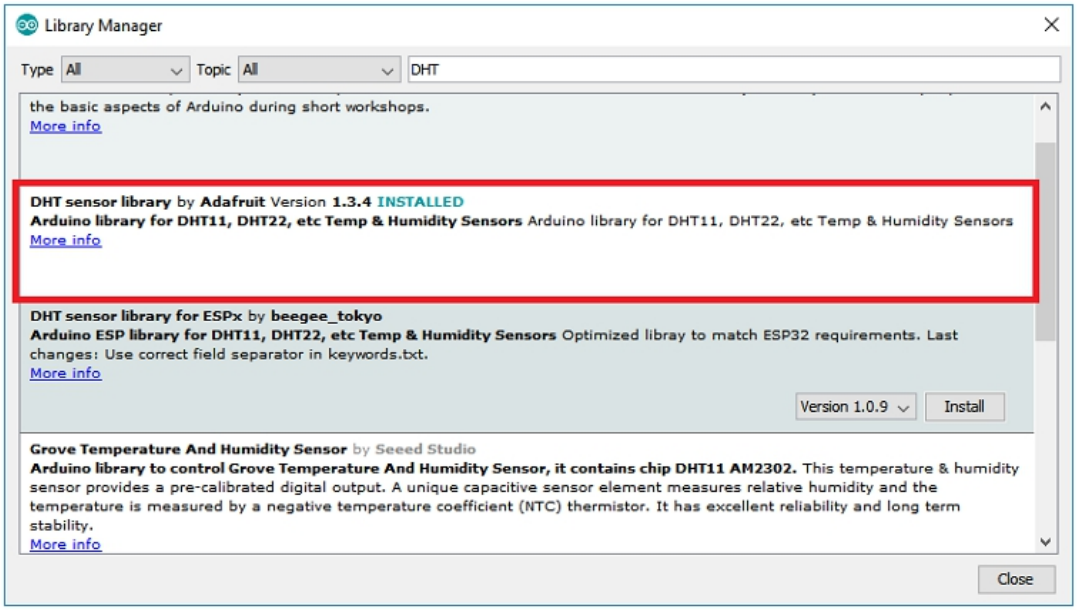
In this example, we connect the DHT data pin to GPIO 4. However, you can use any other appropriate numeric pin.

Installing Libraries:

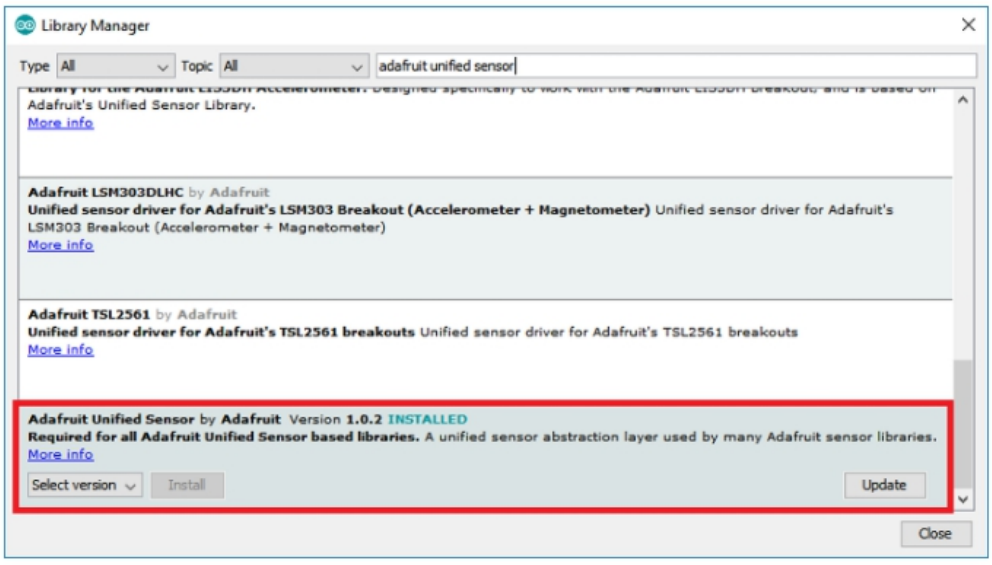
To read from the DHT sensor, we'll use the DHT library from Adafruit. To use this library you also need to install the Adafruit Unified Sensor library. Follow the next steps to install those libraries.

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

Search for "DHT" on the Search box and install the DHT library from Adafruit.



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.

How the Code Works

First, you need to import the DHT library:

#include "DHT.h"

Then, define the digital pin that the DHT sensor 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, you need to select the DHT sensor type you're using. The library supports DHT11, DHT22, and DHT21. Uncomment the sensor type you're using and comment all the others. In this case, we're using the DHT11 sensor.

#define DHTTYPE DHT11 // DHT 11

//#define DHTTYPE DHT22 // DHT 22 (AM2302), AM2321

//#define DHTTYPE DHT21 // DHT 21 (AM2301)

Create a DHT object called dht on the pin and with the sensor type you've specified previously.

DHT dht(DHTPIN, DHTTYPE);

In the setup(), initialize the Serial debugging at a baud rate of 9600, and print a message in the Serial Monitor.

Serial.begin(9600);

Serial.println(F("DHTxx test!"));

Finally, initialize the DHT sensor.

dht.begin();

The loop() starts with a 2000 ms (2 seconds) delay, because the DHT22 maximum sampling period is 2 seconds. So, we can only get readings every two seconds.

delay(2000);

The temperature and humidity are returned in float format. We create

float variables h, t, and f to save the humidity, temperature in Celsius and temperature in Fahrenheit, respectively.

Getting the humidity and temperature is as easy as using the readHumidity() and readTemperature() methods on the dht object, as shown below:

```
float h = dht.readHumidity(); // Read temperature as Celsius (the default)
float t = dht.readTemperature();
```

If you want to get the temperature in Fahrenheit degrees, you need to pass the true parameter as argument to the readTemperature() method.

```
float f = dht.readTemperature(true);
```

There's also an if statement that checks if the sensor returned valid temperature and humidity readings.

```
if (isnan(h) || isnan(t) || isnan(f)) {
Serial.println(F("Failed to read from DHT sensor!"));
  return;
```

After getting the humidity and temperature, the library has a method that computes the heat index. You can get the heat index both in Celsius and Fahrenheit as shown below:

```
// Compute heat index in Fahrenheit (the default)
float hif = dht.computeHeatIndex(f, h); // Compute heat index in Celsius (isFahreheit = false)
float hic = dht.computeHeatIndex(t, h, false);
```

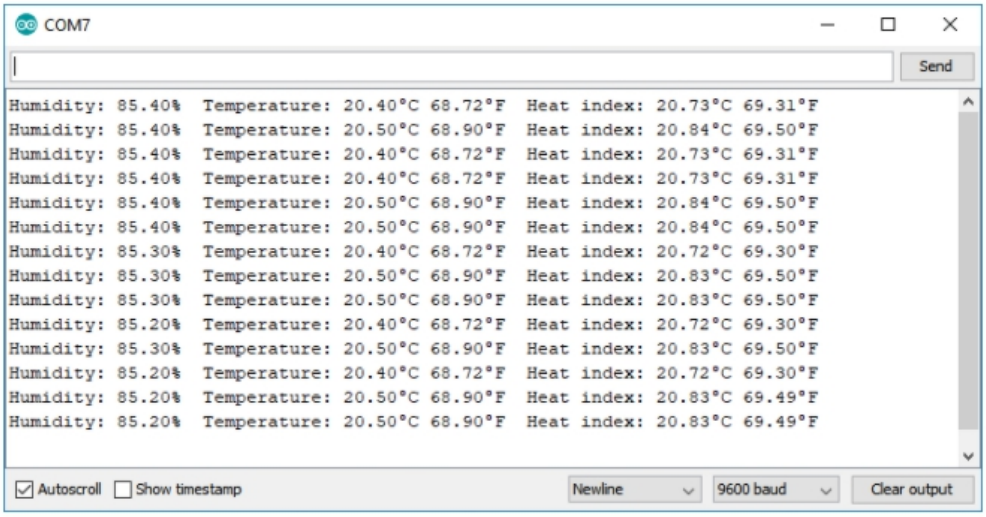
Finally, print all the readings on the Serial Monitor with the following commands:

```
Serial.print(F("Humidity: "));
Serial.print(h);
Serial.print(F("% Temperature: "));
Serial.print(t); Serial.print(F("°C "));
Serial.print(f); Serial.print(F("°F Heat index: "));
Serial.print(hic);
Serial.print(F("°C "));
Serial.print(hif);
Serial.println(F("°F"));
```

Demonstration

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

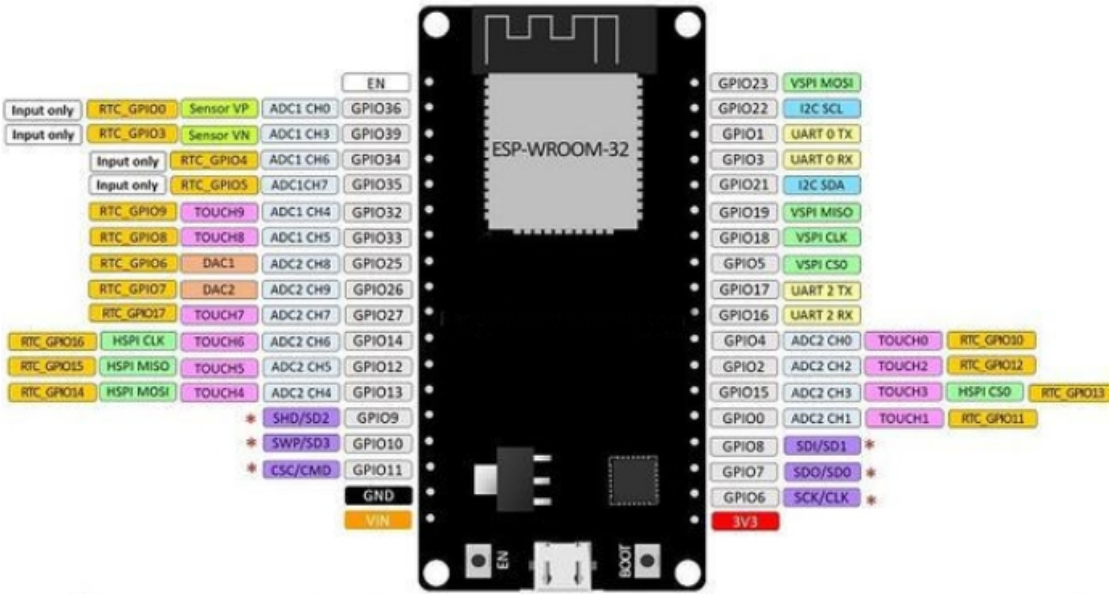
After uploading the code, open the Serial Monitor at a baud rate of 9600. You should get the latest temperature and humidity readings in the Serial Monitor every two seconds.



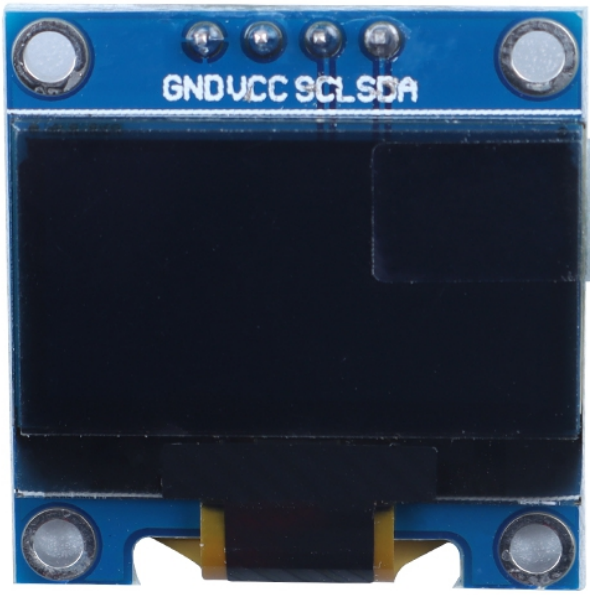
Lesson 5 : Using esp32 to control ssd1306 OLED display

This tutorial introduces how to use esp32 to control OLED display to display characters. You can modify the code according to your own idea to make OLED display what you want to display. The ESP board will use Arduino IDE for programming.

Pin display of esp32 development board:

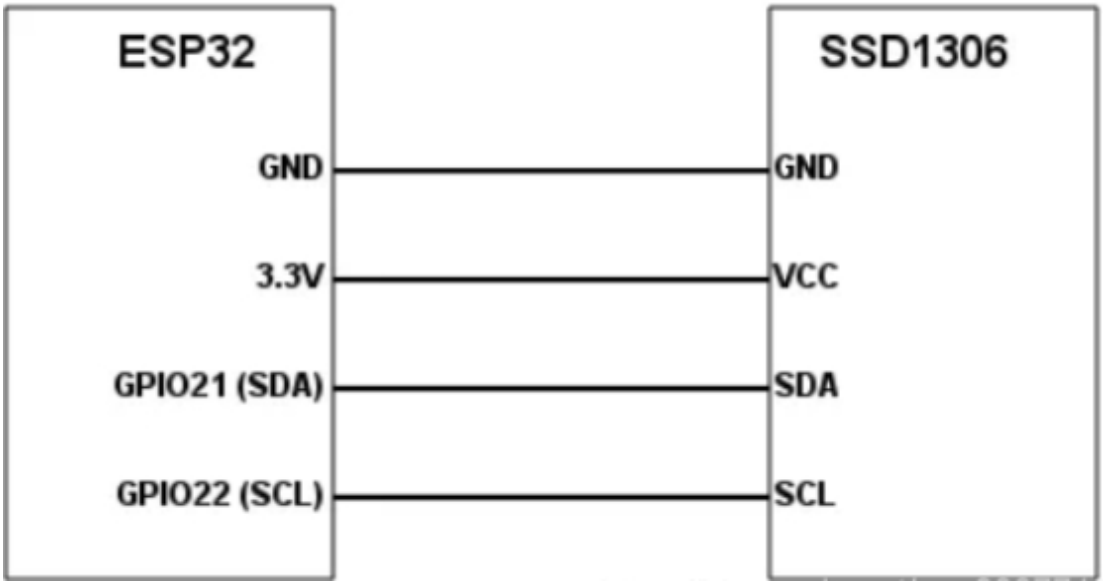


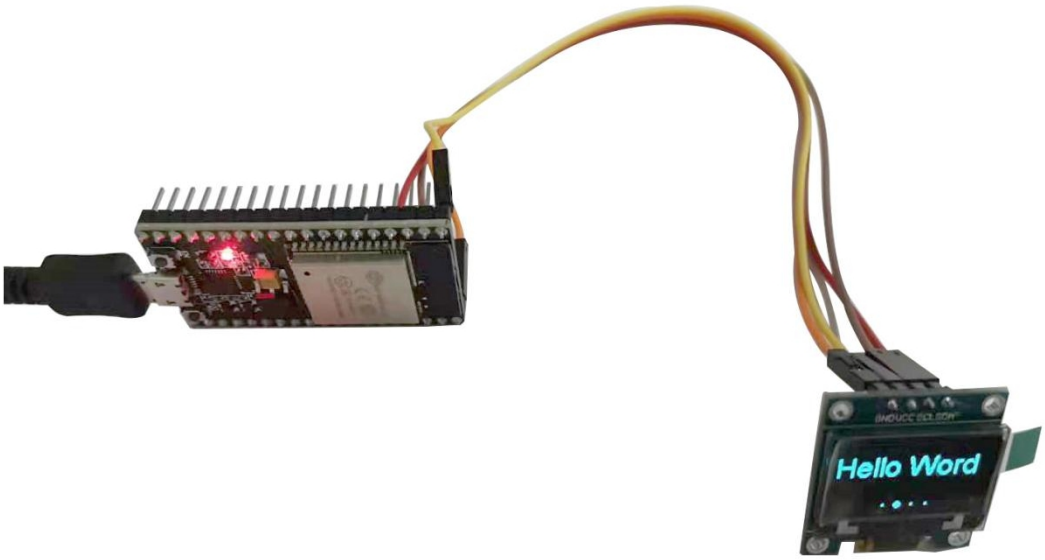
OLED screen Pin Display:



Schematic diagram of the project:

For this project, you need to connect the OLED screen pins to the esp32 board. Follow the next schematic.





n this example, we connect the pin data pin and clock pin of OLED screen to gpio21 and gpio22. Note that wrong connection is not allowed. Wrong connection will cause OLED screen not to be displayed.

OLED display code：

First, we need to include the wire. H library, which is required for I2C communication with OLED displays. We also need to include the ssd1306. H library, which we will use to interact with devices

```
#include <Wire.h>  // Only needed for Arduino 1.6.5 and earlier
#include "SSD1306Wire.h" // legacy include: `#include "SSD1306.h"`
#include "OLEDDisplayUi.h"
#include "images.h"
```

The constructor of the class mentioned below receives the I2C address of the device as the first parameter, namely, 0x3c. As the second and third parameters, the constructor receives the number of SDA and SCL pins, respectively. In our example, as shown in the diagram, we use pins 21 and 22 of esp32.

```
SSD1306Wire  display(0x3c, 21, 22);
```

Use the library to display the string you want to display.

```
OLEDDisplayUi ui  ( &display );
```

Frame 1 function, we display a logo.

```
void drawFrame1(OLEDDisplay *display, OLEDDisplayUiState* state, int16_t x, int16_t y)
{
  display->drawXbm(x + 34, y + 12, Logo_width, Logo_height, Logo_bits);
}
```

Frame 2 function, we show "Hello word".

```
void drawFrame2(OLEDDisplay *display, OLEDDisplayUiState* state, int16_t x, int16_t y) {
  display->setTextAlignment(TEXT_ALIGN_LEFT);
  display->setFont(ArialMT_Plain_24);
  display->drawStringMaxWidth(0 + x, 10 + y, 128, "Hello Word");
}
```

Frame 3 function, we show some titles.

```
void drawFrame3(OLEDDisplay *display, OLEDDisplayUiState* state, int16_t x, int16_t y) {
  display->setTextAlignment(TEXT_ALIGN_LEFT);
  display->setFont(ArialMT_Plain_24);
  display->drawString(0 + x, 0 + y, "Zhi Yi");
  display->setFont(ArialMT_Plain_10);
  display->drawString(0 + x, 24 + y, "science");
```

```
  display->setFont(ArialMT_Plain_10);
  display->drawString(0 + x, 34 + y, " and technology");
}
```

Frame 4 function, we show some introduction.

```
void drawFrame4(OLEDDisplay *display, OLEDDisplayUiState* state, int16_t x, int16
_t y) {
  display->setTextAlignment(TEXT_ALIGN_LEFT);
  display->setFont(ArialMT_Plain_10);
  display->drawStringMaxWidth(0 + x, 10 + y, 128, "Using Arduino ide to compile e
sp32 project and make OLED LCD display");
}
```

Next, we loop through the contents of each function.

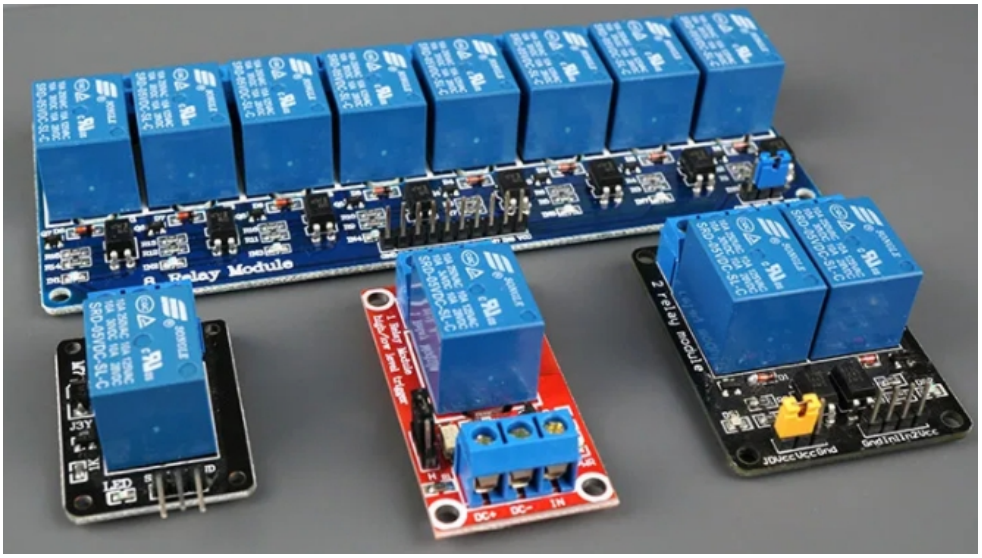
```
FrameCallback frames[] = { drawFrame1, drawFrame2, drawFrame3,drawFrame4,};

// how many frames are there?
int frameCount = 4;
```

Lesson 6 How to use ESP32 to control a relay module

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).

There are different relay modules with a different number of channels. You can find relay modules with one, two, four, eight and even sixteen channels. The number of channels determines the number of outputs we'll be able to control.

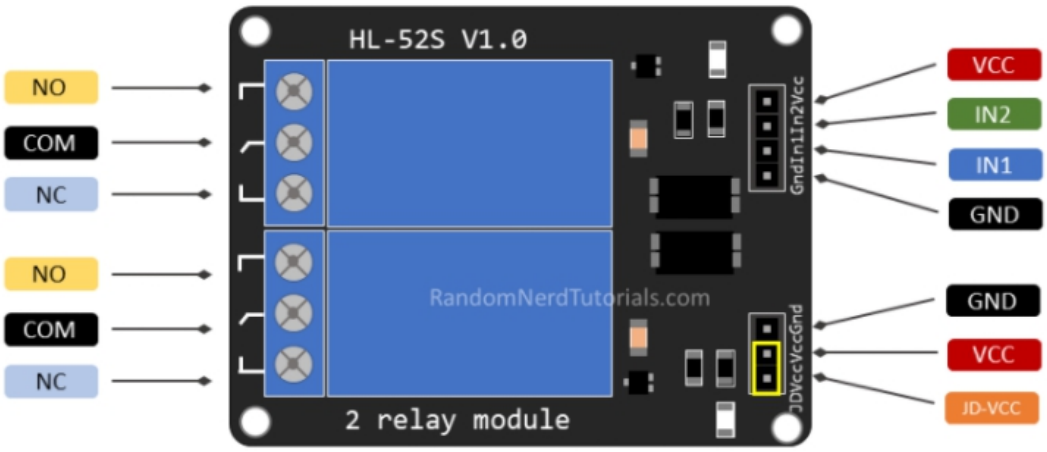


There are relay modules whose electromagnet can be powered by 5V and with 3.3V. Both can be used with the ESP32 – you can either use the VIN pin (that provides 5V) or the 3.3V pin.

Additionally, some come with built-in optocoupler that add an extra "layer" of protection, optically isolating the ESP32 from the relay circuit.

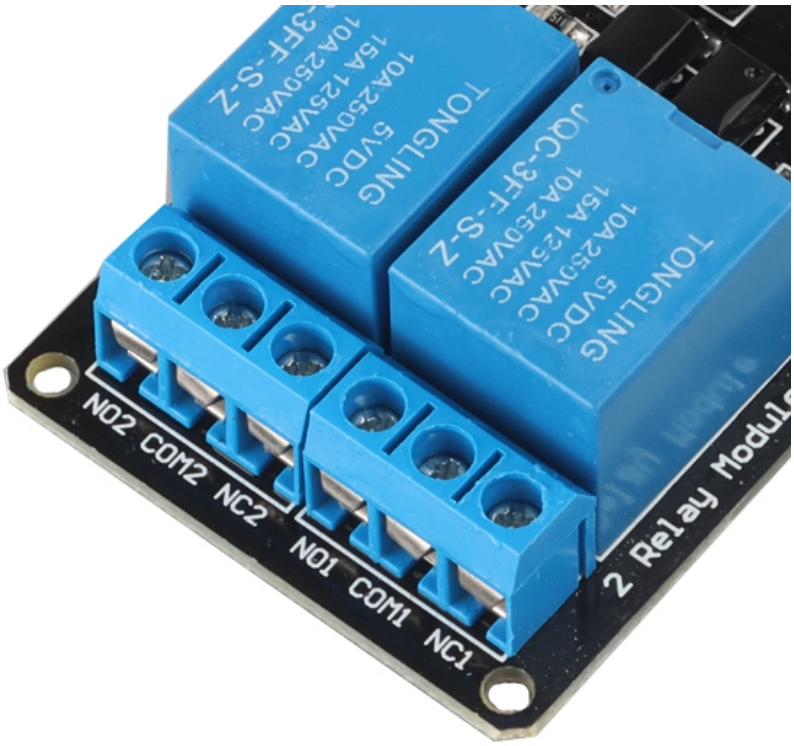
Relay Pinout

For demonstration purposes, let's take a look at the pinout of a 2-channel relay module. Using a relay module with a different number of channels is similar.



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.

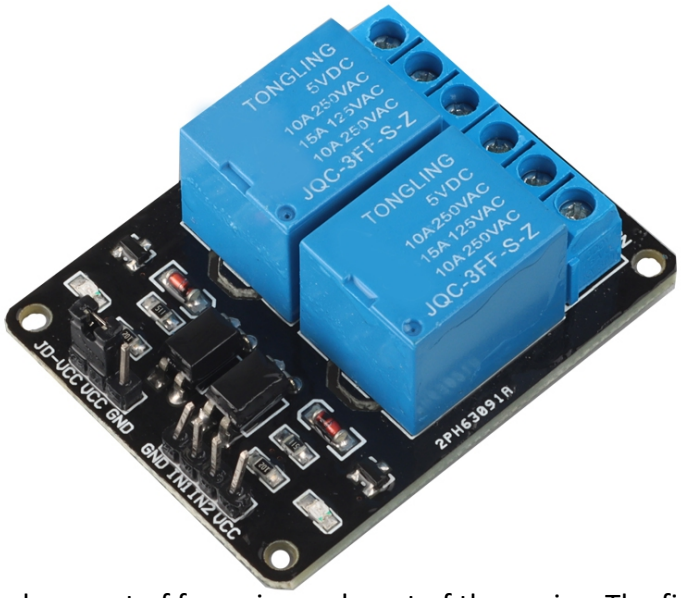
Mains Voltage Connections



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



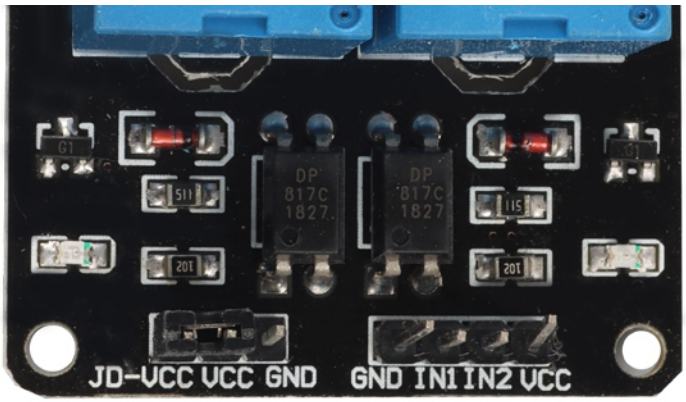
The low-voltage side has a set of four pins and a set of three pins. The first set consists of VCC and GND to power up the module, and input 1 (IN1) and input 2 (IN2) to control the bottom and top relays, respectively.

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:



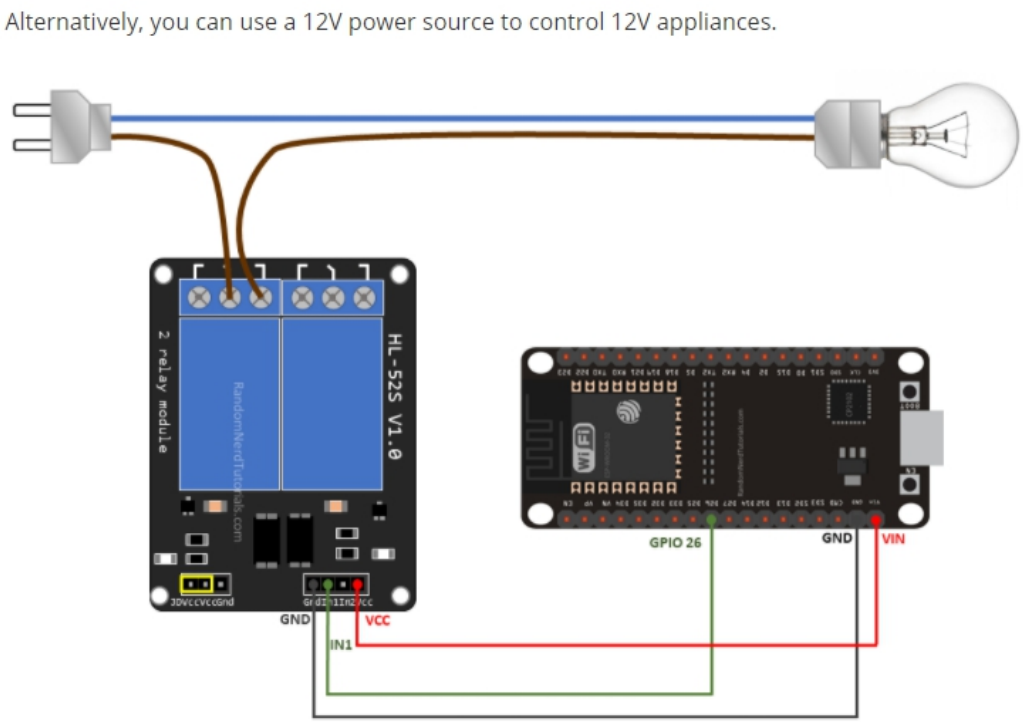
The second set of pins consists of GND, VCC, and JD-VCC pins. The JD-VCC pin powers the electromagnet of the relay. Notice that the module has a jumper cap connecting the VCC and JD-VCC pins; the one shown here is yellow, but yours may be a different color.

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.

Wiring a Relay Module to the ESP32

Connect the relay module to the ESP32 as shown in the following diagram. The diagram shows wiring for a 2-channel relay module, wiring a different number of channels is similar.



In this example, we're controlling a lamp. We just want to light up the lamp occasionally, so it is better to use a normally open configuration.

Controlling a Relay Module with the ESP32 – Arduino Sketch

The code to control a relay with the ESP32 is as simple as controlling an LED or any other output. In this example, as we're using a normally open configuration, we need to send a LOW signal to let the current flow, and a HIGH signal to stop the current flow.

The following code will light up your lamp for 10 seconds and turn it off for another 10 seconds.

```
const int relay = 26;

void setup() {
  Serial.begin(115200);
  pinMode(relay, OUTPUT);}
void loop() {
  // Normally Open configuration, send LOW signal to let current flow
  // (if you're usong Normally Closed configuration send HIGH signal)
  digitalWrite(relay, LOW);
  Serial.println("Current Flowing");
  delay(5000);


  // Normally Open configuration, send HIGH signal stop current flow
  // (if you're usong Normally Closed configuration send LOW signal)
  digitalWrite(relay, HIGH);
  Serial.println("Current not Flowing");
  delay(5000);}
```

Control Multiple Relays with ESP32 Web Server

In this section, we've created a web server example that allows you to control as many relays as you want via web server whether they are configured as normally opened or as normally closed. You just need to change a few lines of code to define the number of relays you want to control and the pin assignment.

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

Installing the ESPAsyncWebServer library

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

Code:

```
// Import required libraries

#include "WiFi.h"

#include "ESPAsyncWebServer.h"



// Set to true to define Relay as Normally Open (NO)
```

```
#define RELAY_NO        true

// Set number of relays
#define NUM_RELAYS    5

// Assign each GPIO to a relay
int relayGPIOs[NUM_RELAYS] = {2, 26, 27, 25, 33};

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

const char* PARAM_INPUT_1 = "relay";
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>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <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: 34px}

        .slider:before {position: absolute; content: ""; height: 52px; width: 52px; left:
8px; bottom: 8px; background-color: #fff; -webkit-transition: .4s; transition: .4s;
border-radius: 68px}

        input:checked+.slider {background-color: #2196F3}

        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?relay="+element.id+"&state=1",
true); }

    else { xhr.open("GET", "/update?relay="+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 ="";

    for(int i=1; i<=NUM_RELAYS; i++){

      String relayStateValue = relayState(i);

      buttons+= "<h4>Relay #" + String(i) + " - GPIO " + relayGPIOs[i-1] +
"</h4><label class=\"switch\"><input type=\"checkbox\"
onchange=\"toggleCheckbox(this)\" id=\"" + String(i) + "\" "+ relayStateValue
+"><span class=\"slider\"></span></label>";

    }

    return buttons;

  }

  return String();

}


String relayState(int numRelay){

  if(RELAY_NO){

    if(digitalRead(relayGPIOs[numRelay-1])){

      return "";

    }
```

```
        else {
            return "checked";
        }
    }
    else {
        if(digitalRead(relayGPIOs[numRelay-1])){
            return "checked";
        }
        else {
            return "";
        }
    }
    return "";
}

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

    // Set all relays to off when the program starts - if set to Normally Open (NO), the relay is off when you set the relay to HIGH
    for(int i=1; i<=NUM_RELAYS; i++){
        pinMode(relayGPIOs[i-1], OUTPUT);
        if(RELAY_NO){
```

```
        digitalWrite(relayGPIOs[i-1], HIGH);
      }
      else{
        digitalWrite(relayGPIOs[i-1], LOW);
      }
    }

    // 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);
    });

    // Send a GET request to
<ESP_IP>/update?relay=<inputMessage>&state=<inputMessage2>
```

```
  server.on("/update", HTTP_GET, [] (AsyncWebServerRequest *request) {
    String inputMessage;
    String inputParam;
    String inputMessage2;
    String inputParam2;
    // GET input1 value on <ESP_IP>/update?relay=<inputMessage>
    if (request->hasParam(PARAM_INPUT_1) &
request->hasParam(PARAM_INPUT_2)) {
      inputMessage = request->getParam(PARAM_INPUT_1)->value();
      inputParam = PARAM_INPUT_1;
      inputMessage2 = request->getParam(PARAM_INPUT_2)->value();
      inputParam2 = PARAM_INPUT_2;
      if(RELAY_NO){
        Serial.print("NO ");
        digitalWrite(relayGPIOs[inputMessage.toInt()-1], !inputMessage2.toInt());
      }
      else{
        Serial.print("NC ");
        digitalWrite(relayGPIOs[inputMessage.toInt()-1], inputMessage2.toInt());
      }
    }
    else {
      inputMessage = "No message sent";
      inputParam = "none";
```

```
        }
        Serial.println(inputMessage + inputMessage2);
        request->send(200, "text/plain", "OK");
    });
    // Start server
    server.begin();
}

void loop() {

}
```

Define Relay Configuration:

Modify the following variable to indicate whether you're using your relays in normally open (NO) or normally closed (NC) configuration. Set the RELAY_NO variable to true for normally open os set to false for normally closed.

```
#define RELAY_NO true
```

Define Number of Relays (Channels)

You can define the number of relays you want to control on the NUM_RELAYS variable. For demonstration purposes, we're setting it to 5.

```
#define NUM_RELAYS 5
```

Define Relays Pin Assignment

In the following array variable you can define the ESP32 GPIOs that will control the relays

```
int relayGPIOs[NUM_RELAYS] = {2, 26, 27, 25, 33};
```

The number of relays set on the NUM_RELAYS variable needs to match the number of GPIOs assigned in the relayGPIOs array.

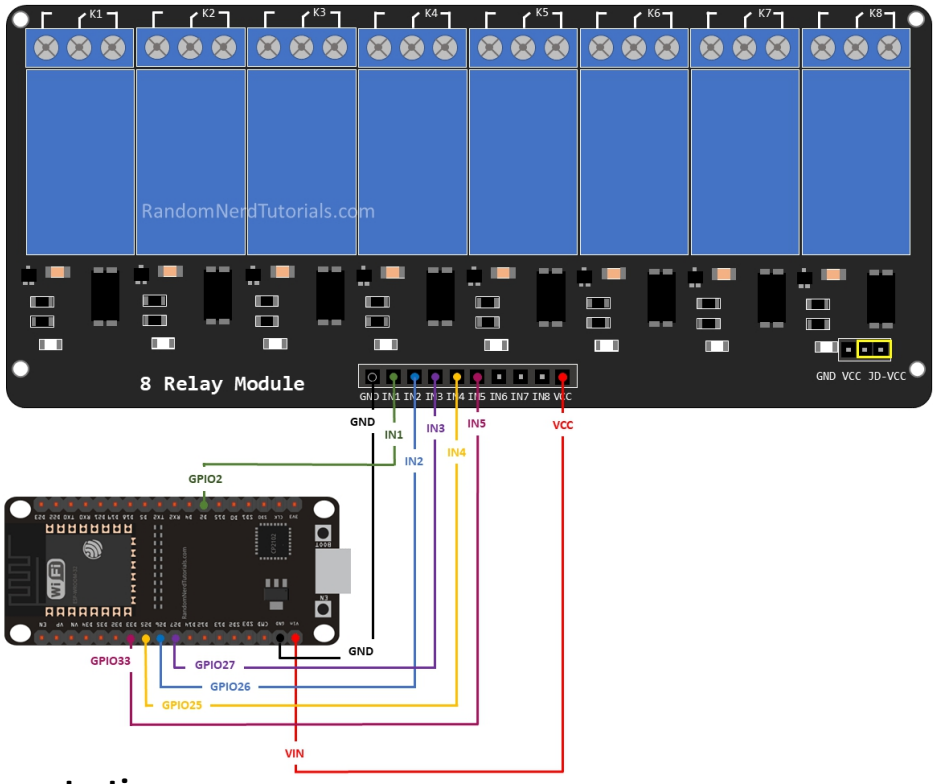
Network Credentials

Insert your network credentials in the following variables.

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

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

Wiring 8 Channel Relay to ESP32

For demonstration purposes, we're controlling 5 relay channels. Wire the ESP32 to the relay module as shown in the next schematic diagram.

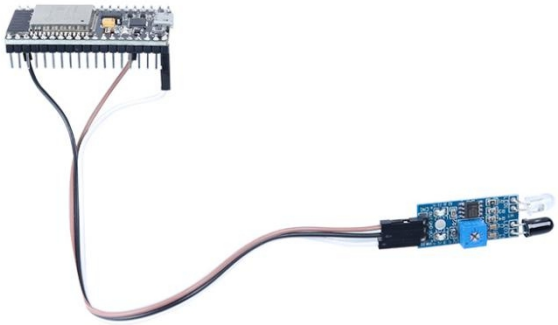


Demonstration

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.

LESSON 7 How to use IR obstacle avoidance sensor on ESP32

In this article, I will write how to use evasive infrared sensor on ESP32.



His sensor can be used to detect objects or obstacles ahead using reflected infrared light.

The sensor has 2 main parts, namely IR transmitter and IR receiver. The infrared transmitter is obligated to emit infrared light. When it hits an object, the infrared light will be reflected. The function of the infrared receiver is to receive infrared reflections.

When the infrared receiver receives the reflected infrared light, the output will be "low". When the infrared receiver does not receive the reflected infrared light, the output will be "high".

There are 2 LED indicators in the sensor. Power indicator light and output indicator light. If the module is powered by current, the power indicator LED will light up. If there is an object in front of the sensor or infrared receiver to receive infrared light reflection, the output indicator LED will light up.

Use jumper wires to connect the IR sensor to the ESP32.

See the picture above or the description about this:

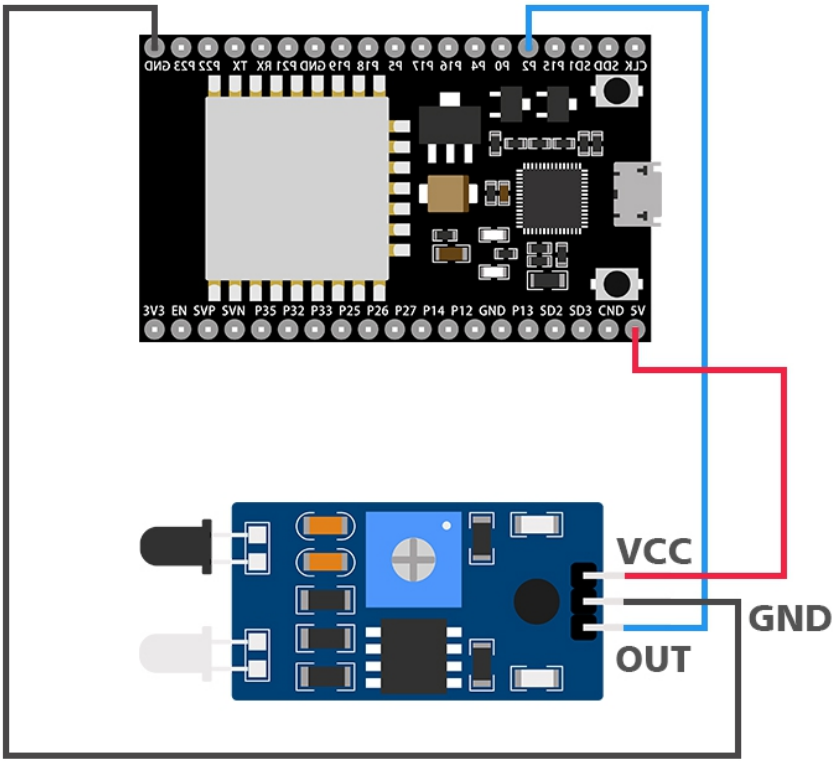
Infrared to ESP32

VCC ==> + 5 volts

Ground ==> ground

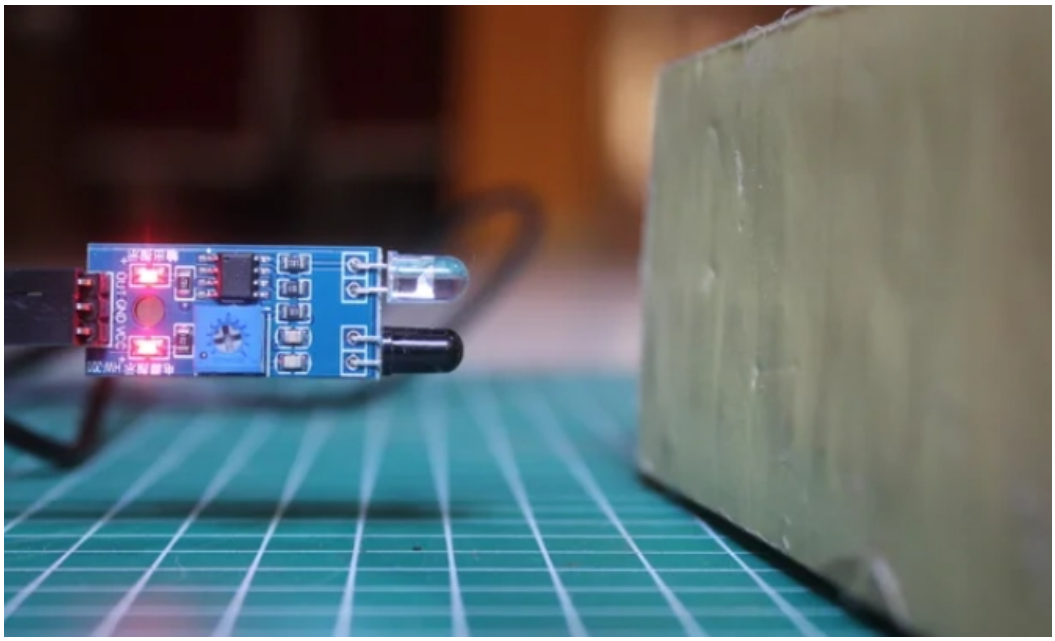
OUT ==> P2

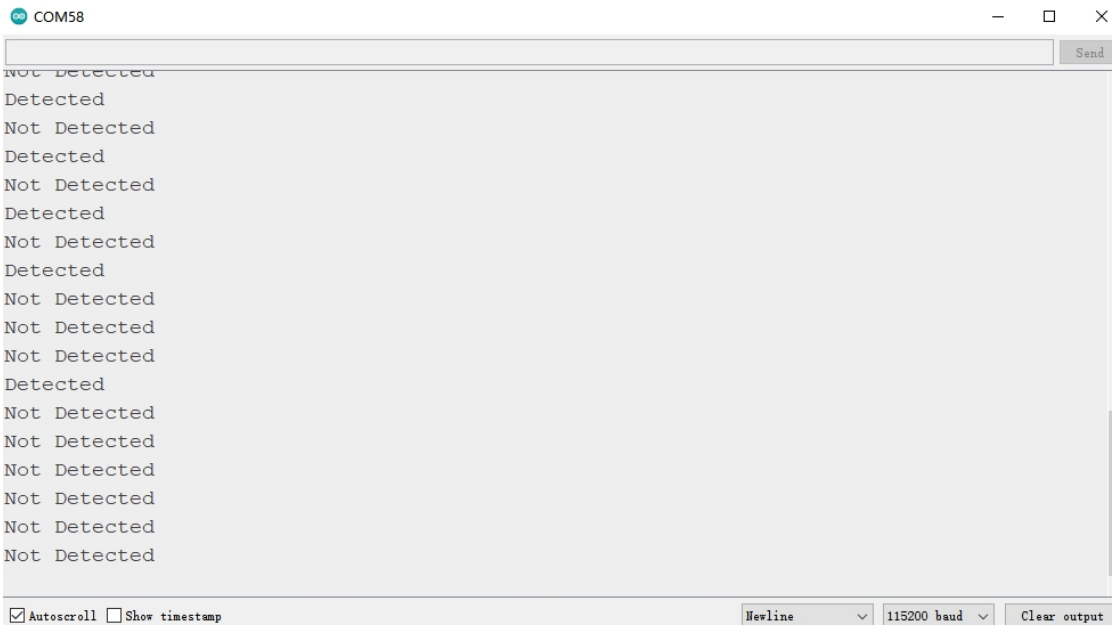
Wiring diagram:



Code:

```
int pinIR = 2;

void setup(){

   Serial.begin(115200);

   pinMode(pinIR, INPUT);

   Serial.println("Detect IR Sensor");

   delay(1000);

}

void loop(){

   int IRstate = digitalRead(pinIR);


   if(IRstate == LOW){

      Serial.println("Detected");


   }

   else if(IRstate == HIGH){

        Serial.println("Not Detected");

   }

   delay(1000);

}
```



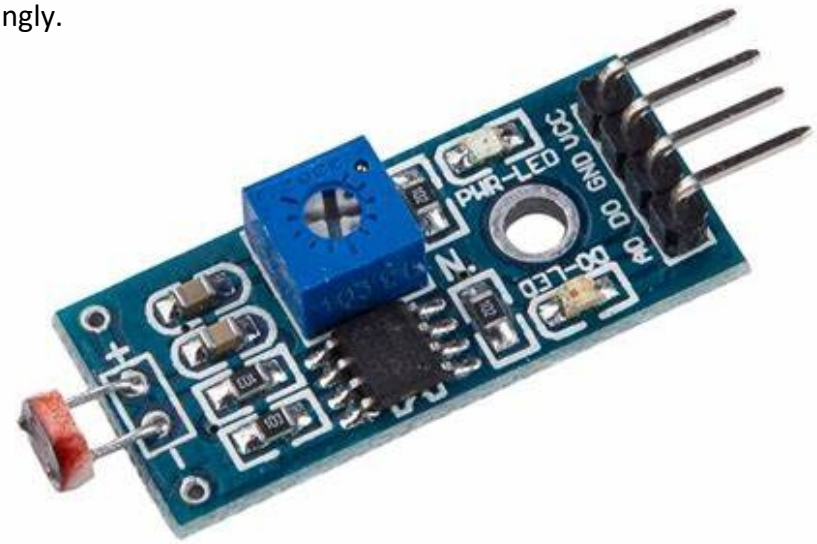


If you place an object in front of the sensor, the serial monitor will say "Detected".

if there is no object in front of the sensor, the monitor serial will say "Not Detected".

Lesson 8 How to use a photoresistor sensor on ESP32

In this lesson, I will show you how to interface ESP32 with a photoresistor (a type of resistor whose resistance varies with the lighting level) and ESP32 to make the LED light automatically. ESP32 is used to analyze the lighting level and turn the LED on or off accordingly.



What are photoresistors?

Photo resistors, also known as light dependent resistors (LDR), are light sensitive devices most often used to indicate the presence or absence of light, or to measure the light intensity. In the dark, their resistance is very high, sometimes up to 1MΩ, but when the LDR sensor is exposed to light, the resistance drops dramatically, even down to a few ohms, depending on the light intensity. LDRs have a sensitivity that varies with the wavelength of the light applied and are nonlinear devices. They are used in many applications but are sometimes made obsolete by other devices such as photodiodes and phototransistors. Some countries have banned LDRs made of lead or cadmium over environmental safety concerns.

Light dependent resistor definition

Photo resistors are light sensitive resistors whose resistance decreases as the intensity of light they are exposed to increases.

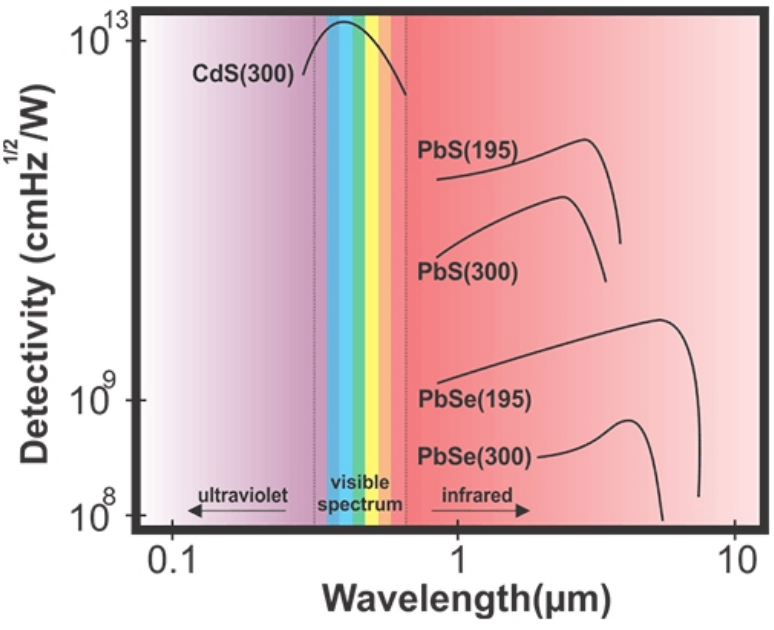
Characteristics

Types of photo resistors and working mechanisms

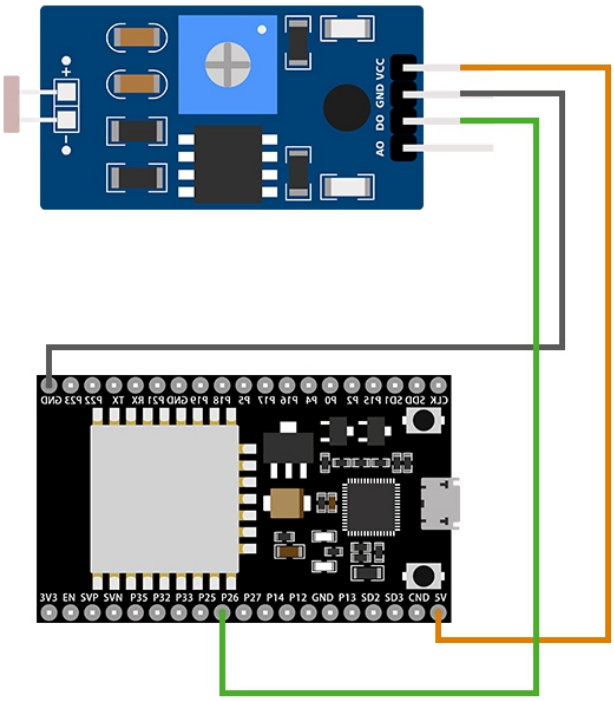
Based on the materials used, photo resistors can be divided into two types; intrinsic and extrinsic. Intrinsic photo resistors use undoped materials such as silicon or germanium. Photons that fall on the device excite electrons from the valence band to the conduction band, and the result of this process are more free electrons in the material, which can carry current, and therefore less resistance. Extrinsic photo resistors are made of materials doped with impurities, also called dopants. The dopants create a new energy band above the existing valence band, populated by electrons. These electrons need less energy to make the transition to the conduction band thanks to the smaller energy gap. The result is a device sensitive to different wavelengths of light. Regardless, both types will exhibit a decrease in resistance when illuminated. The higher the light intensity, the larger the resistance drop is. Therefore, the resistance of LDRs is an inverse, nonlinear function of light intensity.

Wavelength dependency

The sensitivity of a photo resistor varies with the light wavelength. If the wavelength is outside a certain range, it will not affect the resistance of the device at all. It can be said that the LDR is not sensitive in that light wavelength range. Different materials have different unique spectral response curves of wavelength versus sensitivity. Extrinsic light dependent resistors are generally designed for longer wavelengths of light, with a tendency towards the infrared (IR). When working in the IR range, care must be taken to avoid heat buildup, which could affect measurements by changing the resistance of the device due to thermal effects. The figure shown here represents the spectral response of photoconductive detectors made of different materials, with the operating temperature expressed in K and written in the parentheses.



Wiring diagram:



Code：

```
//cosntants for the pins where sensors are plugged into.
const int sensorPin = 26;
const int ledPin =2;
//Set up some global variables for the light level an initial value.
int lightInit;    // initial value
int lightVal;      // light reading
void setup()
{
   // We'll set up the LED pin to be an output.
   pinMode(ledPin, OUTPUT);
   lightInit = analogRead(sensorPin);
   //we will take a single reading from the light sensor and store it in the lightCal
//variable. This will give us a prelinary value to compare against in the loop
}
void loop()
{
   lightVal = analogRead(sensorPin); // read the current light levels
   //if lightVal is less than our initial reading withing a threshold then it is dark.
   if(lightVal - lightInit <    50)
   {
         digitalWrite (ledPin, HIGH); // turn on light
```

```
  }
  //otherwise, it is bright
  else
  {
    digitalWrite (ledPin, LOW); // turn off light
  }
}
```

Experimental diagram: