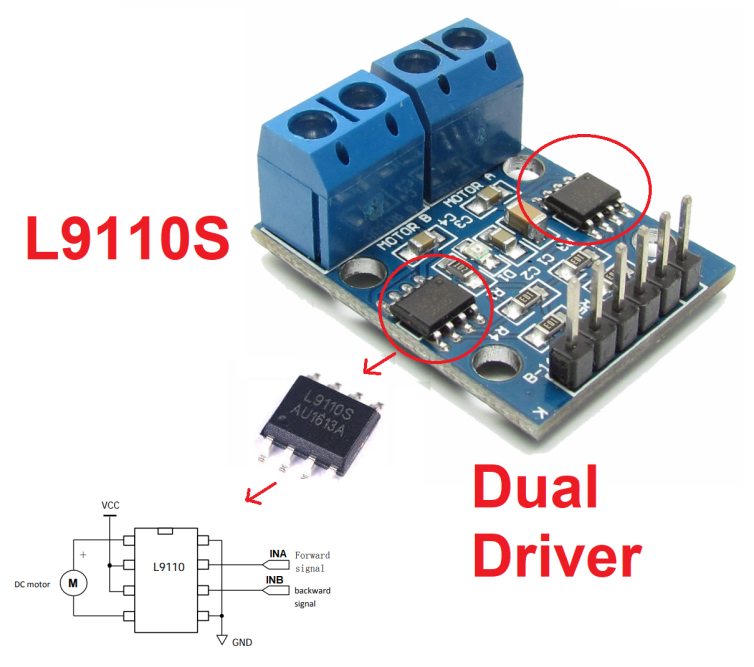
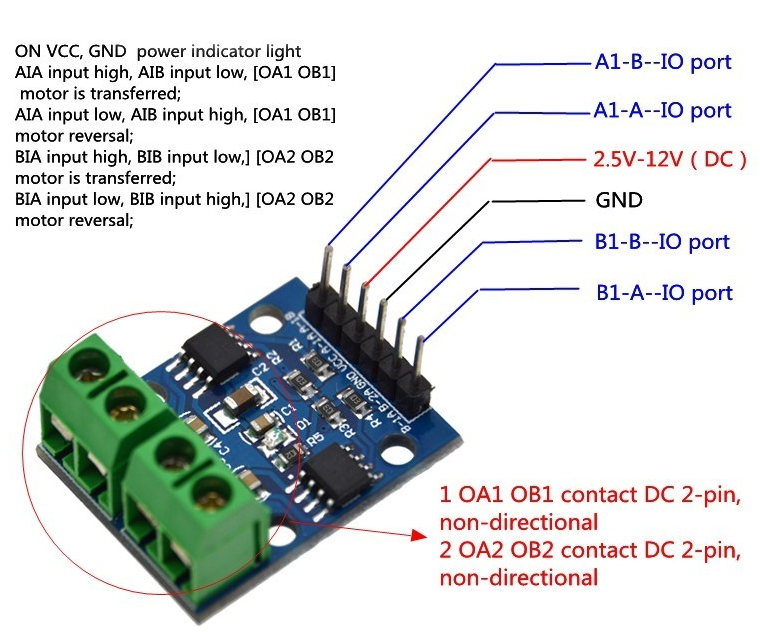
ESP32 and L9110 fan module example

In this example we connect an ESp8266  to a dual L9110 fan module. This is a commonly found, basic low cost module which consists of an L9110 chip and a small motor attached. You need 4 connections between the arduino and the module. VCC, GND , INA and INB. You should use an external power source for Vcc and Gnd

L9110 The ASIC device control and drive motor design two-channel push-pull power amplifier discrete circuits integrated into a monolithic IC, peripheral devices and reduce the cost, improve the reliability of the whole. This chip has two TTL / CMOS compatible with the level of the input, with good resistance; two output terminals can directly forward and reverse movement of the drive motor, it has a large current driving capability, each channel through 750 ~ 800mA of continuous current, peak current capability up to 1.5 ~ 2.0A; while it has a low output saturation voltage; built-in clamp diode reverse the impact of the current release inductive load it in the drive relays, DC motors, stepper motor or switch power tube use on safe and reliable. L9110 is widely used in toy car motor drives, stepper motor drive and switching power tube circuit.

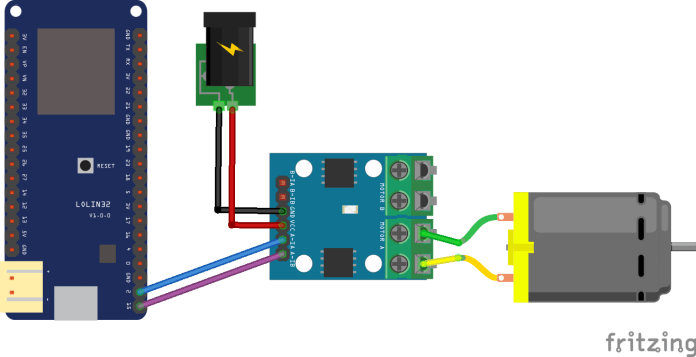
* Motor Voltage: 2.5 ~12V
* Motor channels: 2
* Max Continuous Current per Channel: 800mA
* Size: 31mm x 22mm x 12mm

this is a picture of a typical module



Lets look at how to connect the ESP32 to the module

**Layout**



**Code**

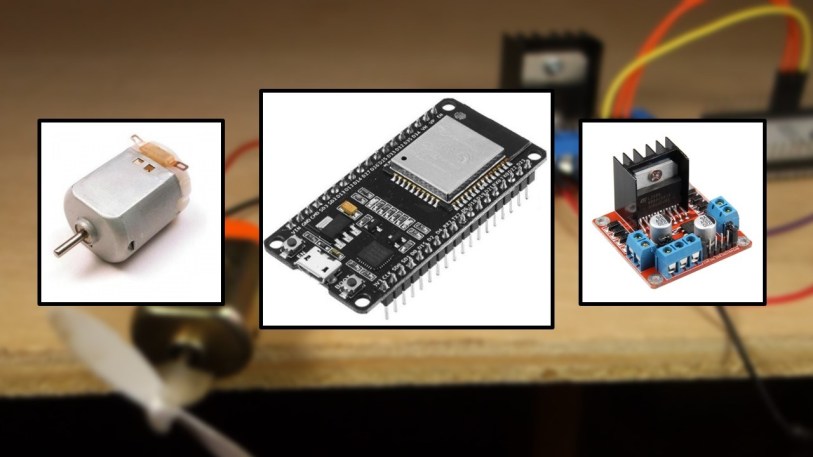
No libraries needed in this example, fairly basic example this one, upload the sketch and the fan will just run in one direction at one speed. if you have one of the dual modules you can have 2 motors fitted and technically move in any direction

int INA = 2;  
int INB = 15;  
  
void setup(){  
pinMode(INA,OUTPUT);  
pinMode(INB,OUTPUT);  
}

void loop(){  
digitalWrite(INA,LOW);  
digitalWrite(INB,HIGH);  
delay(1000);  
}

**ESP32 with DC Motor and L298N Motor Driver – Control Speed and Direction**

This tutorial shows how to control the direction and speed of a DC motor using an ESP32 and the L298N Motor Driver. First, we’ll take a quick look on how the L298N motor driver works. Then, we’ll show you an example on how to control the speed and direction of a DC motor using the ESP32 with Arduino IDE and the L298N motor driver.



**Note**: there are many ways to control a DC motor. We’ll be using the L298N motor driver. This tutorial is also compatible with similar motor driver modules.

To better understand with this tutorial, you may want to take a look at the following posts:

* [Getting Started with ESP32 Dev Module](https://randomnerdtutorials.com/getting-started-with-esp32/)
* [Installing the ESP32 Board in Arduino IDE (Windows instructions)](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/)
* [Installing the ESP32 Board in Arduino IDE (Mac and Linux instructions)](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-mac-and-linux-instructions/)
* [ESP32 Web Server – Arduino IDE](https://randomnerdtutorials.com/esp32-web-server-arduino-ide/)

**Parts Required**

To complete this tutorial you need the following parts:



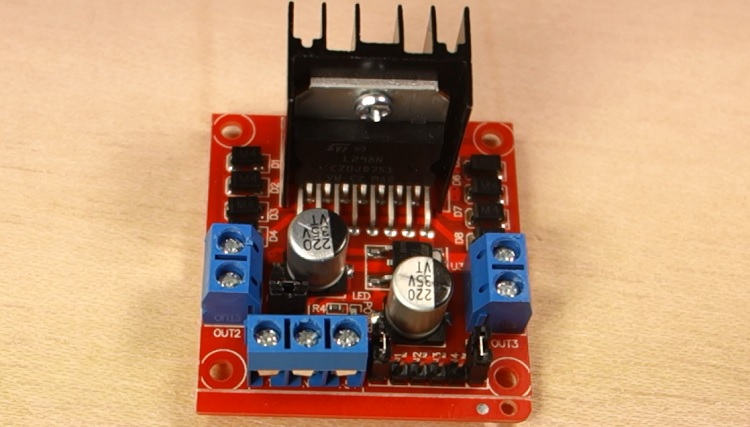
* [ESP32 DOIT DEVKIT V1 Board](https://makeradvisor.com/tools/esp32-dev-board-wi-fi-bluetooth/) – [read ESP32 Development Boards Review and Comparison](https://makeradvisor.com/esp32-development-boards-review-comparison/)
* [DC motor](https://makeradvisor.com/tools/mini-dc-motor/)
* [L298N motor driver](https://makeradvisor.com/tools/l298n-motor-driver/)
* Power source: [4x 1.5 AA batteries](https://amzn.to/2HKmjW4) or [Bench power supply](https://makeradvisor.com/best-bench-power-supply/)
* [2x 100nF ceramic capacitors](https://makeradvisor.com/tools/ceramic-capacitors-kit/) (optional)
* [1x SPDT slide switch](https://makeradvisor.com/tools/spdt-slide-switch/) (optional)
* [Jumper wires](https://makeradvisor.com/tools/jumper-wires-kit-120-pieces/)

**Introducing the L298N Motor Driver**

There are many ways to control a DC motor. The method we’ll use here is suitable for most hobbyist motors, that require 6V or 12V to operate.

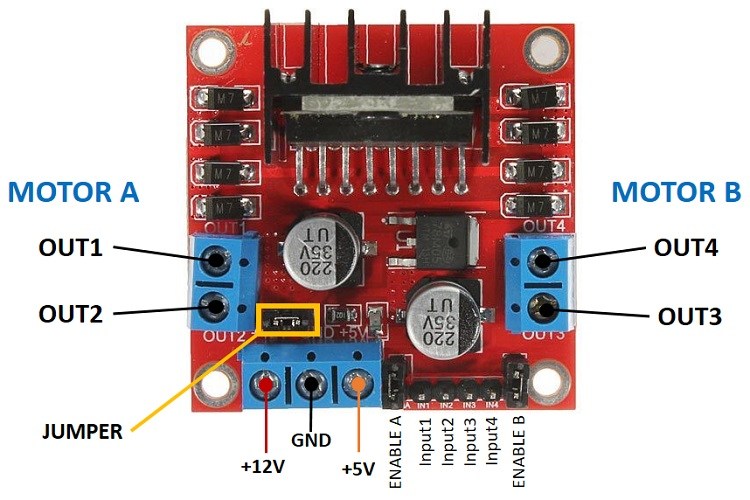
We’re going to use the L298N motor driver that can handle up to 3A at 35V. Additionally, it allows us to drive two DC motors simultaneously, which is perfect to build a robot.

The L298N motor driver is shown in the following figure:



**L298N Motor Driver pinout**

Let’s take a look at the L298N motor driver pinout and see how it works.



The motor driver has a two terminal block in each side for each motor. OUT1 and OUT2 at the left and OUT3 and OUT4 at the right.

* **OUT1**: DC motor A + terminal
* **OUT2**: DC motor A – terminal
* **OUT3**: DC motor B + terminal
* **OUT4**: DC motor B – terminal

At the bottom you have a three terminal block with +12V, **GND**, and +5V.

The +12V terminal block is used to power up the motors. The +5V terminal is used to power up the L298N chip. However, if the jumper is in place, the chip is powered using the motor’s power supply and you don’t need to supply 5V through the +5V terminal.

**Note**: if you supply more than 12V, you need to remove the jumper and supply 5V to the +5V terminal.

It’s important to note that despite the +12V terminal name, with the setup we’ll use here (with the jumper in place) you can supply any voltage between 6V and 12V. In this tutorial will be using 4 AA 1.5V batteries that combined output approximately 6V, but you can use any other suitable power supply. For example, you can use a [bench power supply](https://makeradvisor.com/best-bench-power-supply/) to test this tutorial.

**In summary:**

* **+12V**: The +12V terminal is where you should connect your power supply
* **GND**: power supply GND
* **+5V**: provide 5V if jumper is removed. Acts as a 5V output if jumper is in place
* **Jumper**: jumper in place – uses the motors power supply to power up the chip. Jumper removed: you need to provide 5V to the +5V terminal. If you supply more than 12V, you should remove the jumper

At the bottom right you have four input pins and two enable terminals. The input pins are used to control the direction of your DC motors, and the enable pins are used to control the speed of each motor.

* **IN1:**Input 1 for Motor A
* **IN2**: Input 2 for Motor A
* **IN3**: Input 1 for Motor B
* **IN4**: Input 2 for Motor B
* **EN1**: Enable pin for Motor A
* **EN2**: Enable pin for Motor B

There are jumper caps on the enable pins by default. You need to remove those jumper caps to control the speed of your motors.

**Control DC motors with the L298N**

Now that you’re familiar with the L298N Motor Driver, let’s see how to use it to control your DC motors.

**Enable pins**

The enable pins are like an ON and OFF switch for your motors. For example:

* If you send a **HIGH signal** to the enable 1 pin, motor A is ready to be controlled and at the maximum speed;
* If you send a**LOW signal** to the enable 1 pin, motor A turns off;
* If you send a **PWM signal**, you can control the speed of the motor. The motor speed is proportional to the duty cycle. However, note that for small duty cycles, the motors might not spin, and make a continuous buzz sound.

| **SIGNAL ON THE ENABLE PIN** | **MOTOR STATE** |
| --- | --- |
| HIGH | Motor enabled |
| LOW | Motor not enabled |
| PWM | Motor enabled: speed proportional to duty cycle |

**Input pins**

The input pins control the direction the motors are spinning. Input 1 and input 2 control motor A, and input 3 and 4 control motor B.

* If you apply LOW to input1 and HIGH to input 2, the motor will spin forward;
* If you apply power the other way around: HIGH to input 1 and LOW to input 2, the motor will rotate backwards. Motor B can be controlled using the same method but applying HIGH or LOW to input 3 and input 4.

**Controlling 2 DC Motors – ideal to build a robot**

If you want to [build a robot car](https://randomnerdtutorials.com/build-robot-car-chassis-kit-arduino/) using 2 DC motors, these should be rotating in specific directions to make the robot go left, right, forward or backwards.

For example, if you want your robot to move forward, both motors should be rotating forward. To make it go backwards, both should be rotating backwards.

To turn the robot in one direction, you need to spin the opposite motor faster. For example, to make the robot turn right, enable the motor at the left, and disable the motor at the right. The following table shows the input pins’ state combinations for the robot directions.

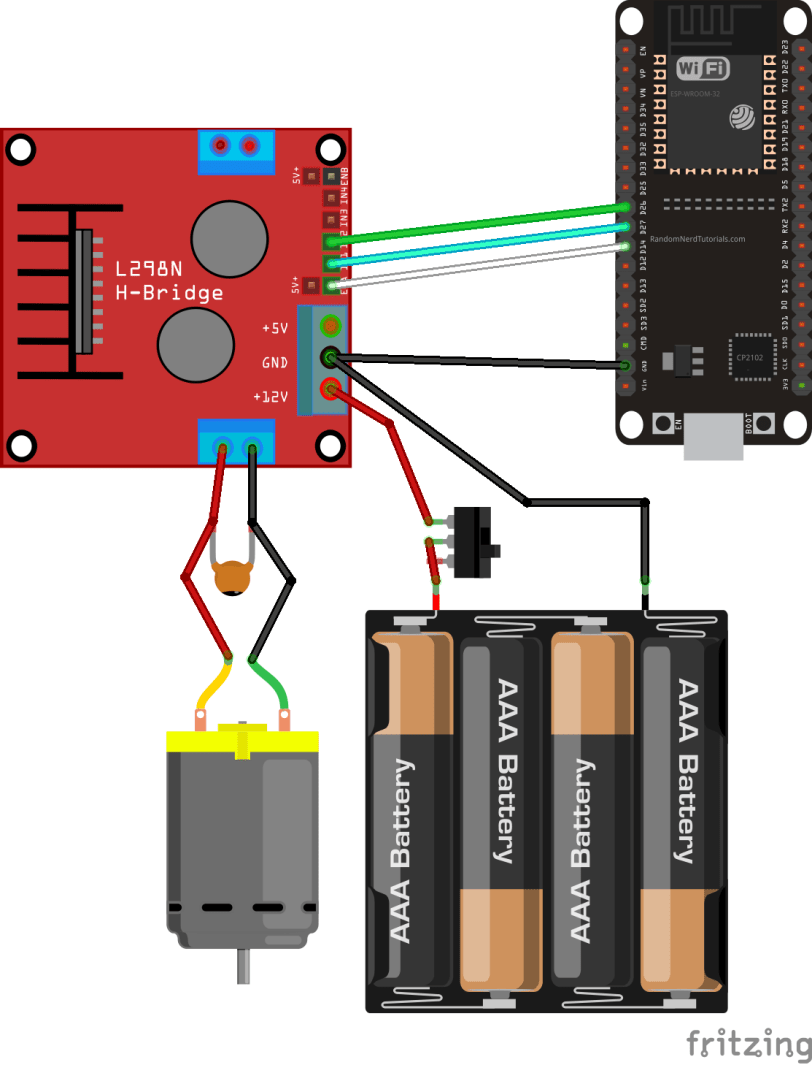
| **DIRECTION** | **INPUT 1** | **INPUT 2** | **INPUT 3** | **INPUT 4** |
| --- | --- | --- | --- | --- |
| Forward | 0 | 1 | 0 | 1 |
| Backward | 1 | 0 | 1 | 0 |
| Right | 0 | 1 | 0 | 0 |
| Left | 0 | 0 | 0 | 1 |
| Stop | 0 | 0 | 0 | 0 |

**Control DC Motor with ESP32 – Speed and Direction**

Now that you know how to control a DC motor with the L298N motor driver, let’s build a simple example to control the speed and direction of one DC motor.

**Schematic**

The motor we’ll control is connected to the motor A output pins, so we need to wire the ENABLEA, INPUT1 and INPUT2 pins of the motor driver to the ESP32. Follow the next schematic diagram to wire the DC motor and the L298N motor driver to the ESP32.

[](https://i1.wp.com/randomnerdtutorials.com/wp-content/uploads/2018/05/ESP32_1_DC_Motor_bb.png?ssl=1)

The DC motor requires a big jump in current to move, so the motors should be powered using an external power source from the ESP32. As an example, we’re using 4AA batteries, but you can use any other suitable power supply. In this configuration, you can use a power supply with 6V to 12V.

The switch between the battery holder and the motor driver is optional, but it is very handy to cut and apply power. This way you don’t need to constantly connect and then disconnect the wires to save power.

We recommend soldering a 0.1uF ceramic capacitor to the positive and negative terminals of the DC motor, as shown in the diagram to help smooth out any voltage spikes. (Note: the motors also work without the capacitor.)

**Preparing the Arduino IDE**

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

* [**Windows** instructions – ESP32 Board in Arduino IDE](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/)
* [**Mac and Linux** instructions – ESP32 Board in Arduino IDE](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-mac-and-linux-instructions/)

After making sure you have the ESP32 add-on installed, you can continue this tutorial.

**Uploading code**

The following code controls the speed and direction of the DC motor. This code is not useful in the real world, this is just a simple example to better understand how to control the speed and direction of a DC motor with the ESP32.

/\*\*\*\*\*\*\*\*\*

Rui Santos

Complete project details at https://randomnerdtutorials.com

\*\*\*\*\*\*\*\*\*/

// Motor A

int motor1Pin1 = 27;

int motor1Pin2 = 26;

int enable1Pin = 14;

// Setting PWM properties

const int freq = 30000;

const int pwmChannel = 0;

const int resolution = 8;

int dutyCycle = 200;

void setup() {

// sets the pins as outputs:

pinMode(motor1Pin1, OUTPUT);

pinMode(motor1Pin2, OUTPUT);

pinMode(enable1Pin, OUTPUT);

// configure LED PWM functionalitites

ledcSetup(pwmChannel, freq, resolution);

// attach the channel to the GPIO to be controlled

ledcAttachPin(enable1Pin, pwmChannel);

Serial.begin(115200);

// testing

Serial.print("Testing DC Motor...");

}

void loop() {

// Move the DC motor forward at maximum speed

Serial.println("Moving Forward");

digitalWrite(motor1Pin1, LOW);

digitalWrite(motor1Pin2, HIGH);

delay(2000);

// Stop the DC motor

Serial.println("Motor stopped");

digitalWrite(motor1Pin1, LOW);

digitalWrite(motor1Pin2, LOW);

delay(1000);

// Move DC motor backwards at maximum speed

Serial.println("Moving Backwards");

digitalWrite(motor1Pin1, HIGH);

digitalWrite(motor1Pin2, LOW);

delay(2000);

// Stop the DC motor

Serial.println("Motor stopped");

digitalWrite(motor1Pin1, LOW);

digitalWrite(motor1Pin2, LOW);

delay(1000);

// Move DC motor forward with increasing speed

digitalWrite(motor1Pin1, HIGH);

digitalWrite(motor1Pin2, LOW);

while (dutyCycle <= 255){

ledcWrite(pwmChannel, dutyCycle);

Serial.print("Forward with duty cycle: ");

Serial.println(dutyCycle);

dutyCycle = dutyCycle + 5;

delay(500);

}

dutyCycle = 200;

}

[View raw code](https://github.com/RuiSantosdotme/Random-Nerd-Tutorials/raw/master/Projects/ESP32/ESP32_DC_Motor.ino)

Upload the code to your ESP32. Make sure you have the right board and COM port selected. Let’s take a look on how the code works.

**Declaring motor pins**

First, you define the GPIOs the motor pins are connected to. In this case, Input 1 for motor A is connected to GPIO 27, the Input 2 to GPIO 26, and the Enable pin to GPIO 14.

int motor1Pin1 = 27;

int motor1Pin2 = 26;

int enable1Pin = 14;

**Setting the PWM  properties to control the speed**

As we’ve seen previously, you can control the DC motor speed by applying a PWM signal to the enable pin of the L298N motor driver. The speed will be proportional to the duty cycle. To use PWM with the ESP32, you need to set the PWM signal properties first.

const int freq = 30000;

const int pwmChannel = 0;

const int resolution = 8;

int dutyCycle = 200;

In this case, we’re generating a signal of 30000 Hz on channel 0 with a 8-bit resolution. We start with a duty cycle of 200 (you can set a duty cycle value from 0 to 255).

For the frequency we’re using, when you apply duty cycles smaller than 200, the motor won’t move and will make a weird buzz sound. So, that’s why we set a duty cycle of 200 at the start.

**Note**: the PWM properties we’re defining here are just an example. The motor works fine with other frequencies.

**setup()**

In the setup(), you start by setting the motor pins as outputs.

pinMode(motor1Pin1, OUTPUT);

pinMode(motor1Pin2, OUTPUT);

pinMode(enable1Pin, OUTPUT);

You need to configure a PWM signal with the properties you’ve defined earlier by using the ledcSetup() function that accepts as arguments, the pwmChannel, the frequency, and the resolution, as follows:

ledcSetup(pwmChannel, 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 enable1Pin GPIO, that corresponds to GPIO 14. The channel that generates the signal is the pwmChannel, that corresponds to channel 0.

ledcAttachPin(enable1Pin, pwmChannel);

**Moving the DC motor forward**

In the loop() is where the motor moves. The code is well comment on what each part of the code does. To move the motor forward, you set input 1 pin to LOW and input 2 pint to HIGH. In this example, the motor speeds forward for 2 seconds (2000 milliseconds).

// Move the DC motor forward at maximum speed

Serial.println("Moving Forward");

digitalWrite(motor1Pin1, LOW);

digitalWrite(motor1Pin2, HIGH);

delay(2000);

**Moving the DC motor backwards**

To move the DC motor backwards you apply power to the motor input pins the other way around. HIGH to input 1 and LOW to input 2.

// Move DC motor backwards at maximum speed

Serial.println("Moving Backwards");

digitalWrite(motor1Pin1, HIGH);

digitalWrite(motor1Pin2, LOW);

delay(2000);

**Stop the DC motor**

To make the DC motor stop, you can either set the enable pin to LOW, or set both input 1 and input 2 pins to LOW. In this example we’re setting both input pins to LOW.

// Stop the DC motor

Serial.println("Motor stopped");

digitalWrite(motor1Pin1, LOW);

digitalWrite(motor1Pin2, LOW);

delay(1000);

**Controlling the DC motor speed**

To control the DC motor speed, we need to change the PWM signal duty cycle. For that you use the ledcWrite() function that accepts as arguments the PWM channel that is generating the signal (not the output GPIO) and the duty cycle, as follows.

ledcWrite(pwmChannel, dutyCycle);

In our example, we have a while loop that increases the duty cycle by 5 in every loop.

// Move DC motor forward with increasing speed

digitalWrite(motor1Pin1, HIGH);

digitalWrite(motor1Pin2, LOW);

while (dutyCycle <= 255){

ledcWrite(pwmChannel, dutyCycle);

Serial.print("Forward with duty cycle: ");

Serial.println(dutyCycle);

dutyCycle = dutyCycle + 5;

delay(500);

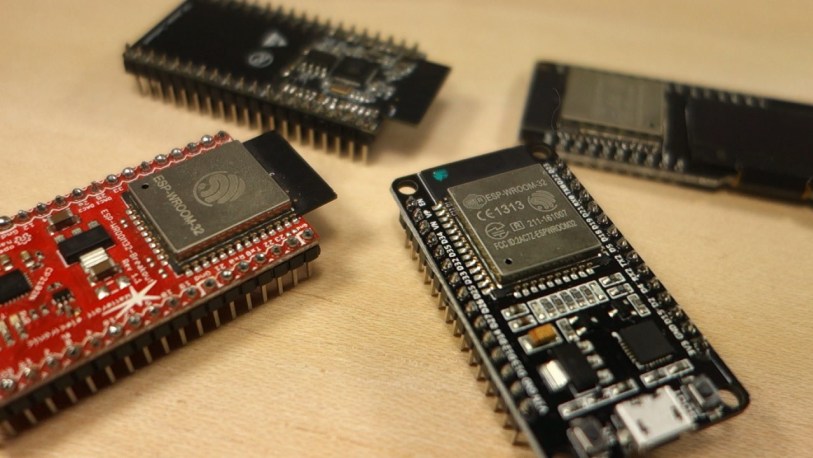
}

When the while condition is no longer true, we set the duty cycle to 200 again.

dutyCycle = 200;

**Getting Started with the ESP32 Board**

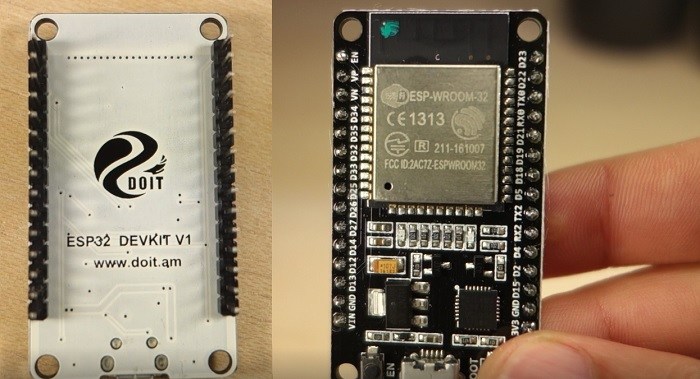
This article is a getting started guide for the ESP32 development board. If you’re familiar with the [ESP8266](https://randomnerdtutorials.com/tutorials-esp8266/), the ESP32 is its sucessor. The ESP32 is loaded with lots of new features. The most relevant: it combines WiFi and Bluetooth wireless capabilities and it’s dual core.



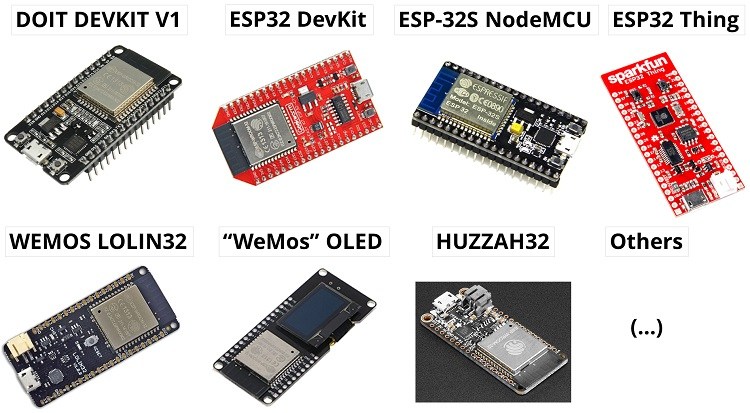
Find the differences between the ESP32 and the ESP8266: [ESP32 vs ESP8266 – Pros and Cons](https://makeradvisor.com/esp32-vs-esp8266/)

**ESP32 DEVKIT DOIT**

In this post, we’ll be using the ESP32 DEVKIT DOIT board as a reference. But the information on this page is also compatible with other ESP32 development boards with the ESP-WROOM-32 chip.



Here’s some examples of ESP32 boards:



**Where to Buy?**

Our ESP32 projects are build using mainly the [ESP32 DEVKIT DOIT board](https://makeradvisor.com/tools/esp32-dev-board-wi-fi-bluetooth/) and that’s the one we recommend getting.

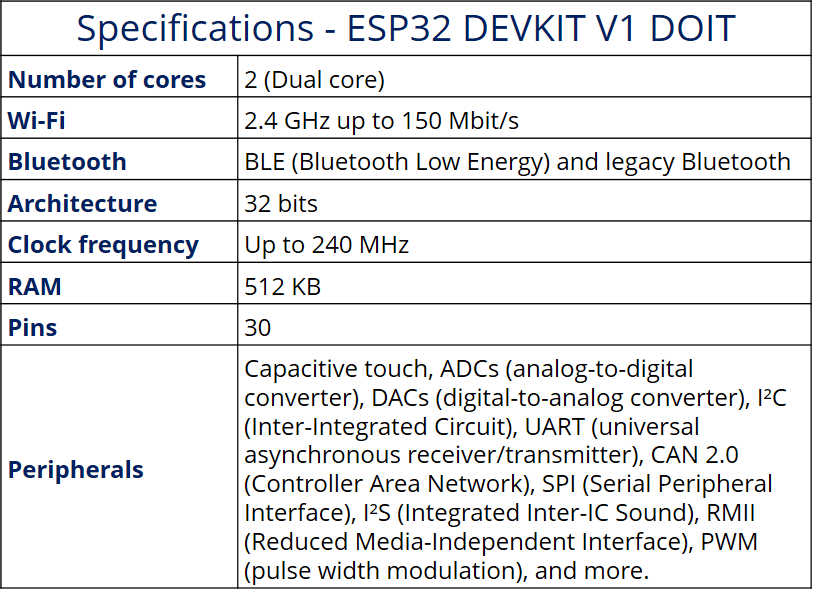
* [ESP32 DEVKIT DOIT board](https://makeradvisor.com/tools/esp32-dev-board-wi-fi-bluetooth/)

You can also read the following article that compares several ESP32 development boards: [ESP32 Development Boards](https://makeradvisor.com/esp32-development-boards-review-comparison/).

**Specifications**

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

* The ESP32 is dual core, this means it has 2 processors.
* It has Wi-Fi and bluetooth built-in.
* It runs 32 bit programs.
* The clock frequency can go up to 240MHz and it has a 512 kB RAM.
* This particular board has 30 or 36 pins, 15 in each row.
* It also has wide variety of peripherals available, like: capacitive touch, ADCs, DACs, UART, SPI, I2C and much more.
* It comes with built-in hall effect sensor and built-in temperature sensor.



To learn more about the ESP32 GPIOs, read our GPIO reference guide: [ESP32 Pinout Reference: Which GPIO pins should you use?](https://randomnerdtutorials.com/esp32-pinout-reference-gpios/)

**Programming Environments**

The ESP32 can be programmed in different programming environments. You can use:

* Arduino IDE
* Espressif IDF (IoT Development Framework)
* [Micropython](https://randomnerdtutorials.com/getting-started-micropython-esp32-esp8266/)
* JavaScript
* LUA
* …

In our projects, we program the ESP32 mainly with Arduino IDE.

**Preparing the ESP32 Board in Arduino IDE**

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

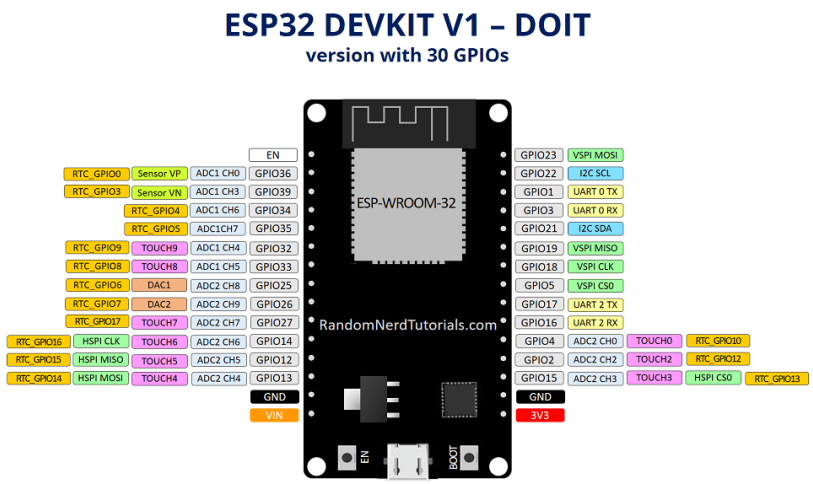
* [**Windows** instructions – Installing the ESP32 Board in Arduino IDE](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/)
* [**Mac and Linux** instructions – Installing the ESP32 Board in Arduino IDE](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-mac-and-linux-instructions/)

**ESP32 Pinout Guide**

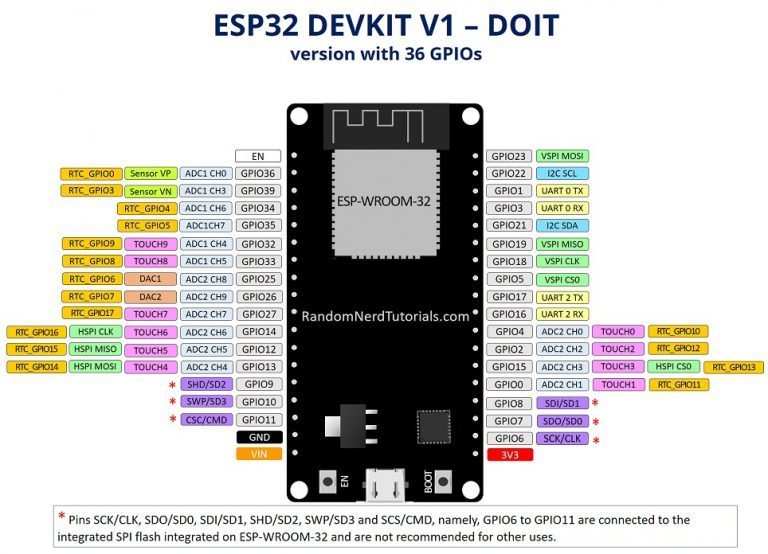
The [ESP32 has more GPIOs](https://randomnerdtutorials.com/esp32-pinout-reference-gpios/) with more functionalities compared to the ESP826.

With the ESP32 you can decide which pins are UART, I2C, or SPI – you just need to set that on the code. This is possible due to the ESP32 chip’s multiplexing feature that allows to assign multiple functions to the same pin. If you don’t set them on the code, the pins will be used as default – as shown in the figure below (the pin location can change depending on the manufacturer).

**Version with 30 GPIOs**



**Version with 36 GPIOs**



You can read our detailed [ESP32 Pinout Reference Guide](https://randomnerdtutorials.com/esp32-pinout-reference-gpios/).

**Upload Code to the ESP32 using Arduino IDE**

To show you how to upload code to your ESP32 board, we’ll build a simple example to blink an LED.

Copy the following code to your Arduino IDE:

/\*

Blink

\*/

// ledPin refers to ESP32 GPIO 23

const int ledPin = 23;

// the setup function runs once when you press reset or power the board

void setup() {

// initialize digital pin ledPin as an output.

pinMode(ledPin, OUTPUT);

}

// the loop function runs over and over again forever

void loop() {

digitalWrite(ledPin, HIGH); // turn the LED on (HIGH is the voltage level)

delay(1000); // wait for a second

digitalWrite(ledPin, LOW); // turn the LED off by making the voltage LOW

delay(1000); // wait for a second

}

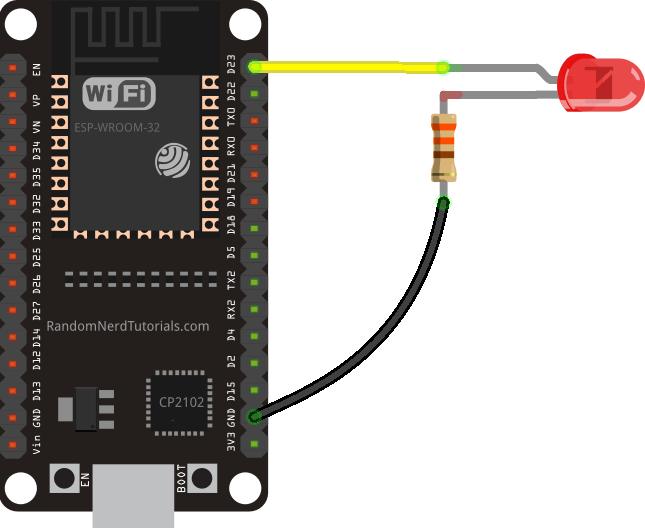
[View raw code](https://github.com/RuiSantosdotme/ESP32-Course/raw/master/code/Blink_LED/Blink_LED.ino)

In this code, we’re controlling an LED connected to GPIO 23.

const int ledPin = 23;

So, connect an LED to your ESP32 by following the next schematic diagram.

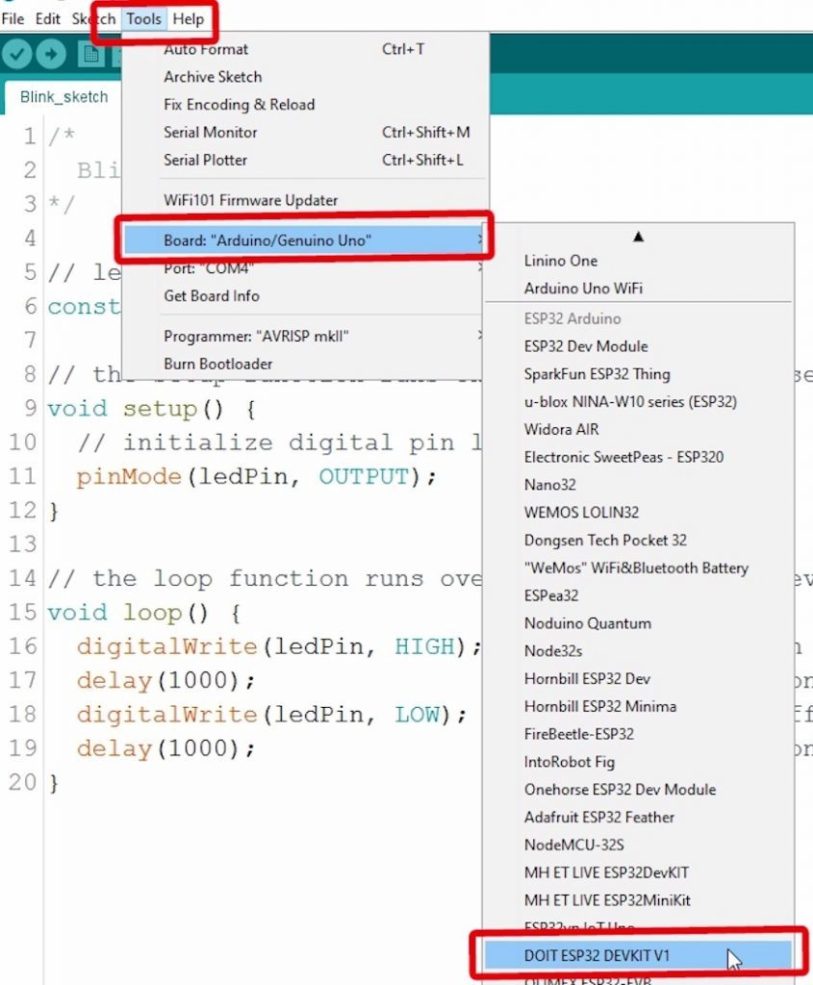
**Important:** always check the pinout for your specific board before building any circuit.



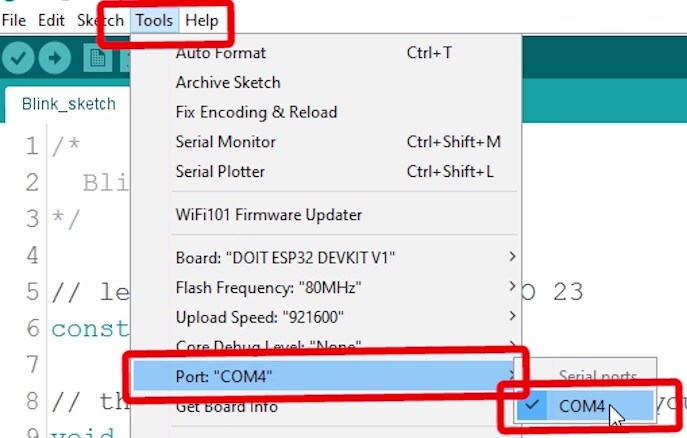
Here’s a list of the parts you need to build this previous circuit:

* [ESP32 DOIT DEVKIT V1 Board](https://makeradvisor.com/tools/esp32-dev-board-wi-fi-bluetooth/)
* [5mm LED](https://makeradvisor.com/tools/3mm-5mm-leds-kit-storage-box/)
* [330 Ohm resistor](https://makeradvisor.com/tools/resistors-kits/)
* [Jumper wires](https://makeradvisor.com/tools/jumper-wires-kit-120-pieces/)
* [Breadboard](https://makeradvisor.com/tools/mb-102-solderless-breadboard-830-points/) (optional)

Plug your ESP32 development board to your computer and follow these next instructions:

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


2) Go to **Tools** > **Port** and select a COM port available.



3) Press the upload button.

https://i0.wp.com/randomnerdtutorials.com/wp-content/uploads/2016/12/arduino-ide-upload-button.png?zoom=1.25&resize=34%2C29&ssl=1

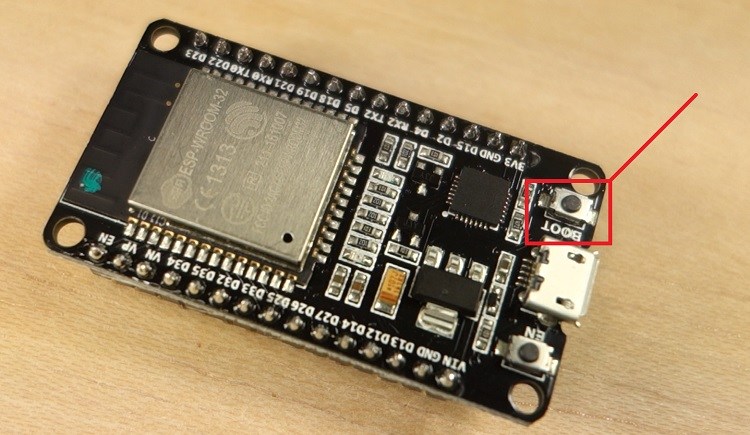
That’s it!

**Note:** If you get the following error when trying to upload code, it means that your ESP32 is not in flashing/uploading mode.

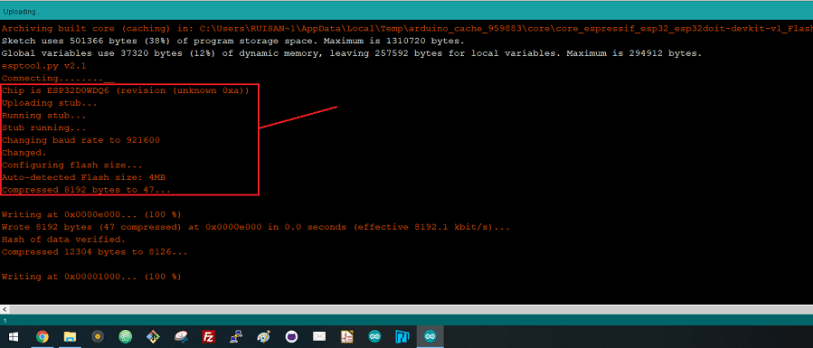
Failed to connect to ESP32: Timed out... Connecting...

To upload code, you need to follow the next steps (make sure you have the right board selected:

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



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



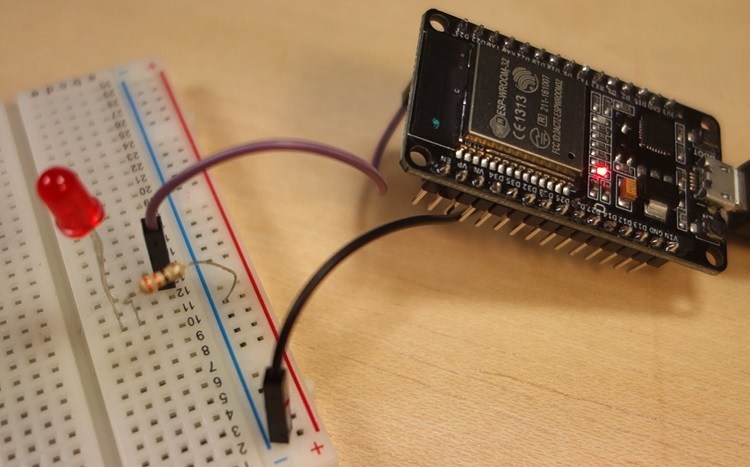
* After that, you should see the “**Done uploading**” message.

That’s it. After uploading the new sketch, you can press the “**ENABLE**” button to restart the ESP32 and run the new uploaded sketch.

**Note:**Learn how to fix the [“Failed to connect to ESP32: Timed out waiting for packet header” error](https://randomnerdtutorials.com/solved-failed-to-connect-to-esp32-timed-out-waiting-for-packet-header/) permanently when trying to upload new code to your ESP32 board once for all.

**Demonstration**

After uploading the code, the LED connected to GPIO 23 should be blinking every other second.



**Wrapping up**

We hope you’ve found this getting started guide useful. The blinking LED is just a simple project to get you started with the ESP32. This is also a great way to learn the procedure you need to do to upload code to your board.

If you like ESP32, we have more than 20 projects with the ESP32 you can find in our repository of ESP32 projects:

* [20+ ESP32 Projects and Tutorials](https://randomnerdtutorials.com/projects-esp32/)

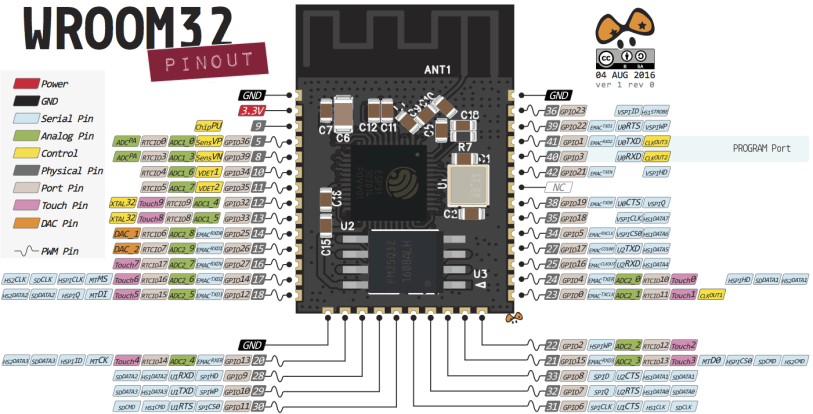
You may also like:

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

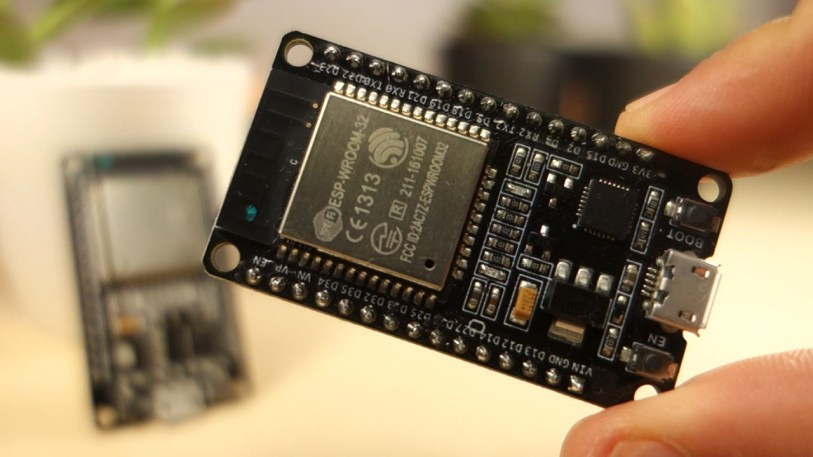
The ESP32 chip comes with 48 pins with multiple functions. Not all pins are exposed in all ESP32 development boards, and there are some pins that cannot be used.

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

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

[](https://i0.wp.com/randomnerdtutorials.com/wp-content/uploads/2018/08/esp32-pinout-chip-ESP-WROOM-32.png?ssl=1)

**Note:**not all GPIOs are accessible in all development boards, but each specific GPIO works in the same way regardless of the development board you’re using. If you’re just getting started with the ESP32, we recommend reading our guide: [Getting Started with the ESP32 Development Board](https://randomnerdtutorials.com/getting-started-with-esp32/).



**ESP32 Peripherals**

The ESP32 peripherals include:

* [18 Analog-to-Digital Converter (ADC) channels](https://randomnerdtutorials.com/esp32-adc-analog-read-arduino-ide/)
* 3 SPI interfaces
* 3 UART interfaces
* [2 I2C interfaces](https://randomnerdtutorials.com/esp32-i2c-communication-arduino-ide/)
* [16 PWM output channels](https://randomnerdtutorials.com/esp32-pwm-arduino-ide/)
* 2 Digital-to-Analog Converters (DAC)
* 2 I2S interfaces
* [10 Capacitive sensing GPIOs](https://randomnerdtutorials.com/esp32-touch-pins-arduino-ide/)

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

Although you can define the pins properties on the software, there are pins assigned by default as shown in the following figure (this is an example for the [ESP32 DEVKIT V1 DOIT board](https://makeradvisor.com/tools/esp32-dev-board-wi-fi-bluetooth/) with 36 pins – the pin location can change depending on the manufacturer).

[](https://i1.wp.com/randomnerdtutorials.com/wp-content/uploads/2018/08/ESP32-DOIT-DEVKIT-V1-Board-Pinout-36-GPIOs-updated.jpg?ssl=1)

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

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

|  |  |  |  |
| --- | --- | --- | --- |
| **GPIO** | **Input** | **Output** | **Notes** |
| **0** | pulled up | OK | outputs PWM signal at boot |
| **1** | TX pin | OK | debug output at boot |
| **2** | OK | OK | connected to on-board LED |
| **3** | OK | RX pin | HIGH at boot |
| **4** | OK | OK |  |
| **5** | OK | OK | outputs PWM signal at boot |
| **6** | x | x | connected to the integrated SPI flash |
| **7** | x | x | connected to the integrated SPI flash |
| **8** | x | x | connected to the integrated SPI flash |
| **9** | x | x | connected to the integrated SPI flash |
| **10** | x | x | connected to the integrated SPI flash |
| **11** | x | x | connected to the integrated SPI flash |
| **12** | OK | OK | boot fail if pulled high |
| **13** | OK | OK |  |
| **14** | OK | OK | outputs PWM signal at boot |
| **15** | OK | OK | outputs PWM signal at boot |
| **16** | OK | OK |  |
| **17** | OK | OK |  |
| **18** | OK | OK |  |
| **19** | OK | OK |  |
| **21** | OK | OK |  |
| **22** | OK | OK |  |
| **23** | OK | OK |  |
| **25** | OK | OK |  |
| **26** | OK | OK |  |
| **27** | OK | OK |  |
| **32** | OK | OK |  |
| **33** | OK | OK |  |
| **34** | OK |  | input only |
| **35** | OK |  | input only |
| **36** | OK |  | input only |
| **39** | OK |  | input only |

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

**Input only pins**

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

* GPIO 34
* GPIO 35
* GPIO 36
* GPIO 39

**SPI flash integrated on the ESP-WROOM-32**

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

* GPIO 6 (SCK/CLK)
* GPIO 7 (SDO/SD0)
* GPIO 8 (SDI/SD1)
* GPIO 9 (SHD/SD2)
* GPIO 10 (SWP/SD3)
* GPIO 11 (CSC/CMD)

**Capacitive touch GPIOs**

The ESP32 has 10 internal capacitive touch sensors. These can sense variations in anything that holds an electrical charge, like the human skin. So they can detect variations induced when touching the GPIOs with a finger. These pins can be easily integrated into capacitive pads, and replace mechanical buttons. The capacitive touch pins can also be used to [wake up the ESP32 from deep sleep](https://randomnerdtutorials.com/esp32-touch-wake-up-deep-sleep/).

Those internal touch sensors are connected to these GPIOs:

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

**Learn how to use the touch pins with Arduino IDE:** [ESP32 Touch Pins with Arduino IDE](https://randomnerdtutorials.com/esp32-touch-pins-arduino-ide/)

**Analog to Digital Converter (ADC)**

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

* ADC1\_CH0 (GPIO 36)
* ADC1\_CH1 (GPIO 37)
* ADC1\_CH2 (GPIO 38)
* ADC1\_CH3 (GPIO 39)
* ADC1\_CH4 (GPIO 32)
* ADC1\_CH5 (GPIO 33)
* ADC1\_CH6 (GPIO 34)
* ADC1\_CH7 (GPIO 35)
* ADC2\_CH0 (GPIO 4)
* ADC2\_CH1 (GPIO 0)
* ADC2\_CH2 (GPIO 2)
* ADC2\_CH3 (GPIO 15)
* ADC2\_CH4 (GPIO 13)
* ADC2\_CH5 (GPIO 12)
* ADC2\_CH6 (GPIO 14)
* ADC2\_CH7 (GPIO 27)
* ADC2\_CH8 (GPIO 25)
* ADC2\_CH9 (GPIO 26)

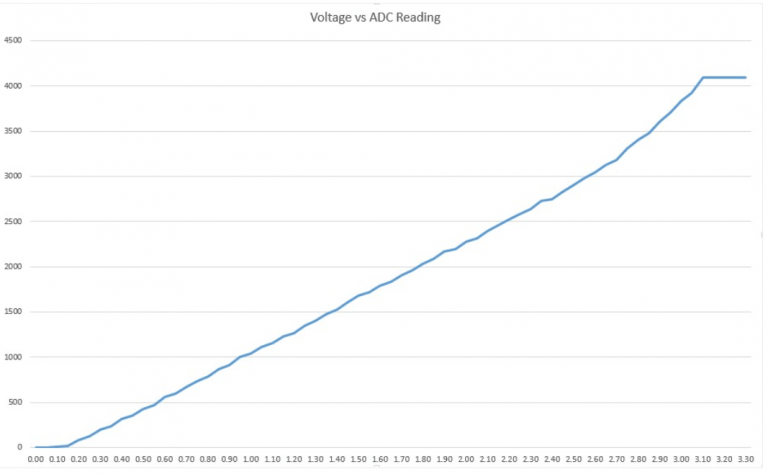
**Learn how to use the ESP32 ADC pins:**

* [ESP32 ADC Pins with Arduino IDE](https://randomnerdtutorials.com/esp32-adc-analog-read-arduino-ide/)
* [ESP32 ADC Pins with MicroPython](https://randomnerdtutorials.com/esp32-esp8266-analog-readings-micropython/)

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

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

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

  
[View source](https://github.com/espressif/arduino-esp32/issues/92)

**Digital to Analog Converter (DAC)**

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

* DAC1 (GPIO25)
* DAC2 (GPIO26)

**RTC GPIOs**

There is RTC GPIO support on the ESP32. The GPIOs routed to the RTC low-power subsystem can be used when the ESP32 is in deep sleep. These RTC GPIOs can be used to wake up the ESP32 from deep sleep when the Ultra Low Power (ULP) co-processor is running. The following GPIOs can be used as an [external wake up source](https://randomnerdtutorials.com/esp32-external-wake-up-deep-sleep/).

* RTC\_GPIO0 (GPIO36)
* RTC\_GPIO3 (GPIO39)
* RTC\_GPIO4 (GPIO34)
* RTC\_GPIO5 (GPIO35)
* RTC\_GPIO6 (GPIO25)
* RTC\_GPIO7 (GPIO26)
* RTC\_GPIO8 (GPIO33)
* RTC\_GPIO9 (GPIO32)
* RTC\_GPIO10 (GPIO4)
* RTC\_GPIO11 (GPIO0)
* RTC\_GPIO12 (GPIO2)
* RTC\_GPIO13 (GPIO15)
* RTC\_GPIO14 (GPIO13)
* RTC\_GPIO15 (GPIO12)
* RTC\_GPIO16 (GPIO14)
* RTC\_GPIO17 (GPIO27)

**Learn how to use the RTC GPIOs to wake up the ESP32 from deep sleep:** [ESP32 Deep Sleep with Arduino IDE and Wake Up Sources](https://randomnerdtutorials.com/esp32-deep-sleep-arduino-ide-wake-up-sources/)

**PWM**

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

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

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

**Learn how to use ESP32 PWM with Arduino IDE:** [ESP32 PWM with Arduino IDE](https://randomnerdtutorials.com/esp32-pwm-arduino-ide/)

**I2C**

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

* GPIO 21 (SDA)
* GPIO 22 (SCL)

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

Wire.begin(SDA, SCL);

**Learn more about I2C communication protocol with the ESP32 using Arduino IDE:**[ESP32 I2C Communication (Set Pins, Multiple Bus Interfaces and Peripherals)](https://randomnerdtutorials.com/esp32-i2c-communication-arduino-ide/)

**SPI**

By default, the pin mapping for SPI is:

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

**Interrupts**

All GPIOs can be configured as interrupts.

**Learn how to use interrupts with the ESP32:**

* [ESP32 interrupts with Arduino IDE](https://randomnerdtutorials.com/esp32-pir-motion-sensor-interrupts-timers/)
* [ESP32 interrupts with MicroPython](https://randomnerdtutorials.com/micropython-interrupts-esp32-esp8266/)

**Strapping Pins**

The ESP32 chip has the following strapping pins:

* GPIO 0
* GPIO 2
* GPIO 4
* GPIO 5 (must be HIGH during boot)
* GPIO 12 (must be LOW during boot)
* GPIO 15 (must be HIGH during boot)

These are used to put the ESP32 into bootloader or flashing mode. On most development boards with built-in USB/Serial, you don’t need to worry about the state of these pins. The board puts the pins in the right state for flashing or boot mode. More information on the [ESP32 Boot Mode Selection can be found here](https://github.com/espressif/esptool/wiki/ESP32-Boot-Mode-Selection).

However, if you have peripherals connected to those pins, you may have trouble trying to upload new code, flashing the ESP32 with new firmware or resetting the board. If you have some peripherals connected to the strapping pins and you are getting trouble uploading code or flashing the ESP32, it may be because those peripherals are preventing the ESP32 to enter the right mode. Read the [Boot Mode Selection documentation](https://github.com/espressif/esptool/wiki/ESP32-Boot-Mode-Selection) to guide you in the right direction. After resetting, flashing, or booting, those pins work as expected.

**Pins HIGH at Boot**

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

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

**Enable (EN)**

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

**GPIO current drawn**

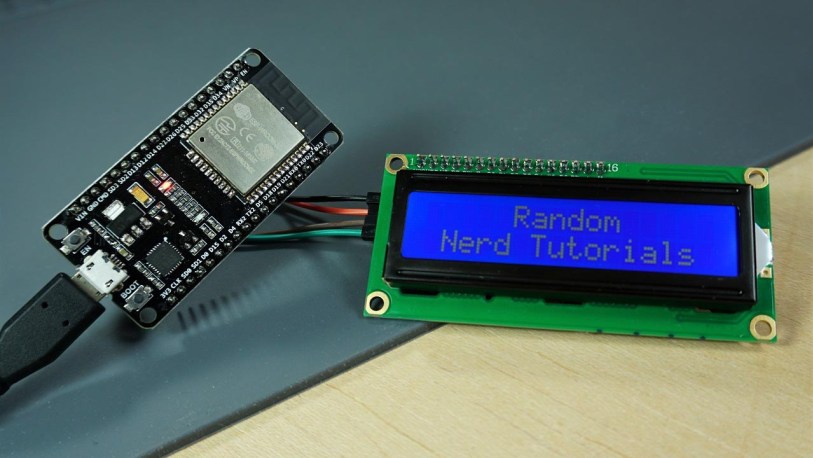
The absolute maximum current drawn per GPIO is 40mA according to the “Recommended Operating Conditions” section in the ESP32 datasheet.

**ESP32 Built-In Hall Effect Sensor**

The ESP32 also features a [built-in hall effect sensor](https://randomnerdtutorials.com/esp32-hall-effect-sensor/) that detects changes in the magnetic field in its surroundings.

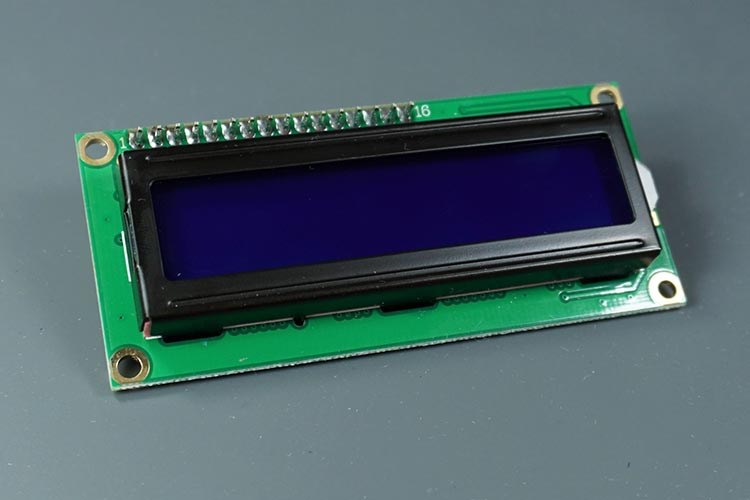
**How to Use I2C LCD with ESP32 on Arduino IDE (ESP8266 compatible)**

This tutorial shows how to use the I2C LCD (Liquid Crystal Display) with the ESP32 using Arduino IDE. We’ll show you how to wire the display, install the library and try sample code to write text on the LCD: static text, and scroll long messages. You can also use this guide with the ESP8266.



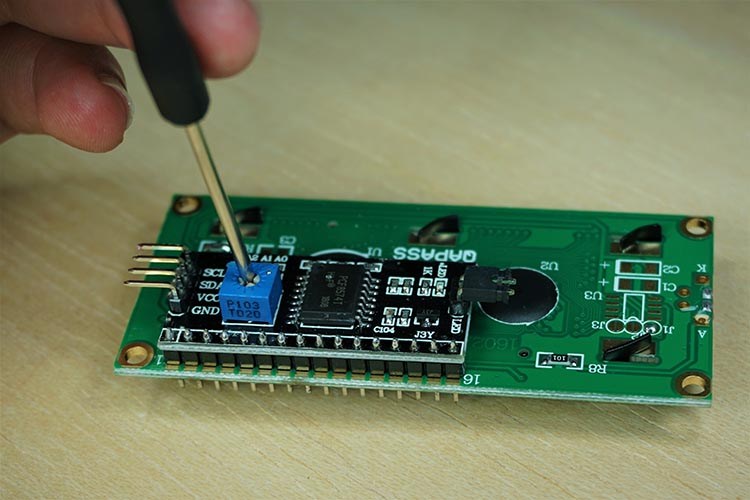
**16×2 I2C Liquid Crystal Display**

For this tutorial we’ll be using a 16×2 I2C LCD display, but LCDs with other sizes should also work.



The advantage of using an I2C LCD is that the wiring is really simple. You just need to wire the SDA and SCL pins.

Additionally, it comes with a built-in potentiometer you can use to adjust the contrast between the background and the characters on the LCD. On a “regular” LCD you need to add a potentiometer to the circuit to adjust the contrast.



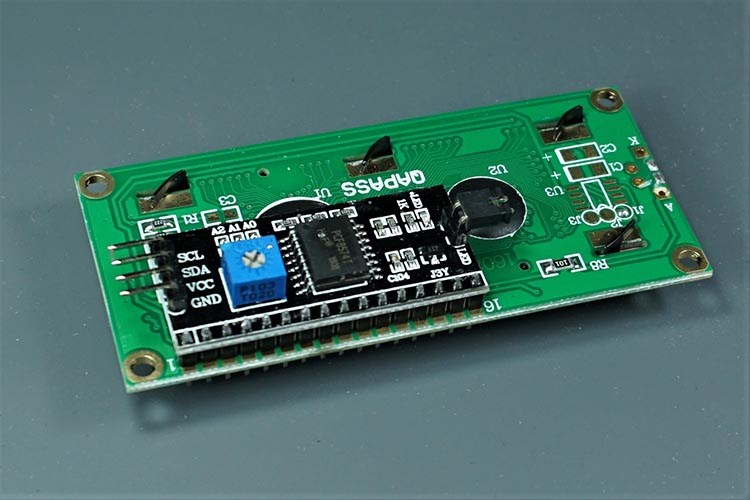
**Parts Required**

To follow this tutorial you need these parts:

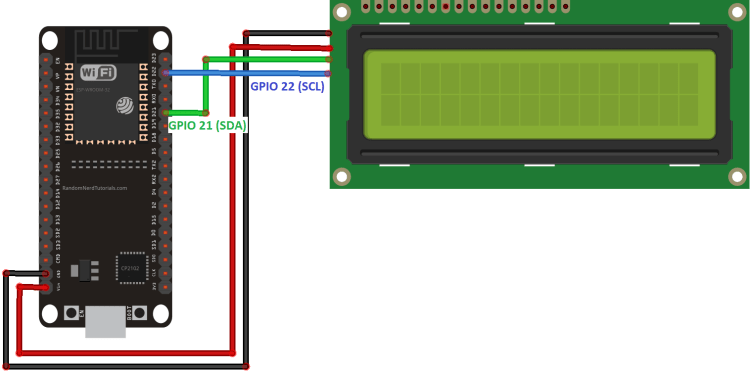
* [ESP32 DOIT DEVKIT V1 Board](https://makeradvisor.com/tools/esp32-dev-board-wi-fi-bluetooth/) – [read ESP32 Development Boards Review and Comparison](https://makeradvisor.com/esp32-development-boards-review-comparison/)
* Optional – [ESP8266 12-E](https://makeradvisor.com/tools/esp8266-esp-12e-nodemcu-wi-fi-development-board/) – read [Best ESP8266 Wi-Fi Development Boards](https://makeradvisor.com/best-esp8266-wi-fi-development-board/)
* [16×2 I2C Liquid Crystal Display (LCD)](https://makeradvisor.com/i2c-lcd-16x2/)
* [Female to female jumper wires](https://makeradvisor.com/tools/jumper-wires-kit-120-pieces/)

**Wiring the LCD to the ESP32**

This display uses I2C communication, which makes wiring really simple.



Wire your LCD to the ESP32 by following the next schematic diagram. We’re using the ESP32 default I2C pins (GPIO 21 and GPIO 22).

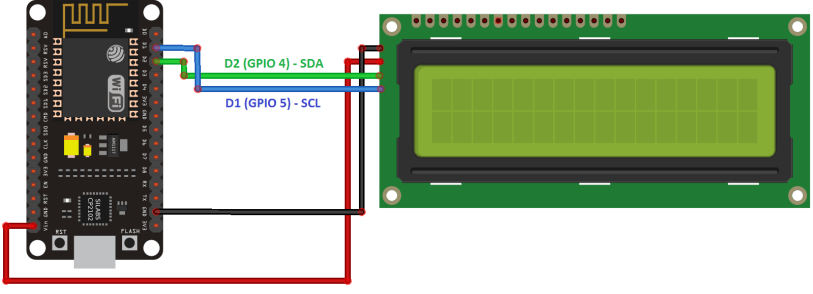


You can also use the following table as a reference.

|  |  |
| --- | --- |
| **I2C LCD** | **ESP32** |
| GND | GND |
| VCC | VIN |
| SDA | GPIO 21 |
| SCL | GPIO 22 |

**Wiring the LCD to the ESP8266**

You can also wire your LCD to the ESP8266 by following the next schematic diagram. We’re using the ESP8266 default I2C pins (GPIO 4 and GPIO 5).



You can also use the following table as a reference.

|  |  |
| --- | --- |
| **I2C LCD** | **ESP8266** |
| GND | GND |
| VCC | VIN |
| SDA | GPIO 4 (D2) |
| SCL | GPIO 5 (D1) |

**Preparing the Arduino IDE**

Before proceeding with the project, you need to install the ESP32 or ESP8266 add-on in the Arduino IDE.

**Arduino IDE with ESP32**

Follow one of the next guides to prepare your Arduino IDE to work with the ESP32:

* [**Windows** instructions – ESP32 Board in Arduino IDE](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/)
* [**Mac and Linux** instructions – ESP32 Board in Arduino IDE](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-mac-and-linux-instructions/)

**Arduino IDE with ESP8266**

To install the ESP8266 add-on in your Arduino IDE, read the following tutorial: [How to Install the ESP8266 Board in Arduino IDE](https://randomnerdtutorials.com/how-to-install-esp8266-board-arduino-ide/).

**Installing the LiquidCrystal\_I2C Library**

There are several libraries that work with the I2C LCD. We’re using [this library by Marco Schwartz](https://github.com/marcoschwartz/LiquidCrystal_I2C). Follow the next steps to install the library:

1. [Click here to download the LiquidCrystal\_I2C library](https://github.com/marcoschwartz/LiquidCrystal_I2C/archive/master.zip). You should have a .zip folder in your Downloads
2. Unzip the *.zip* folder and you should get **LiquidCrystal\_I2C-master**folder
3. Rename your folder from  to**LiquidCrystal\_I2C**
4. Move the **LiquidCrystal\_I2C**folder to your Arduino IDE installation **libraries**folder
5. Finally, re-open your Arduino IDE

**Getting the LCD Address**

Before displaying text on the LCD, you need to find the LCD I2C address. With the LCD properly wired to the ESP32, upload the following I2C Scanner sketch.

/\*\*\*\*\*\*\*\*\*

Rui Santos

Complete project details at https://randomnerdtutorials.com

\*\*\*\*\*\*\*\*\*/

#include <Wire.h>

void setup() {

Wire.begin();

Serial.begin(115200);

Serial.println("\nI2C Scanner");

}

void loop() {

byte error, address;

int nDevices;

Serial.println("Scanning...");

nDevices = 0;

for(address = 1; address < 127; address++ ) {

Wire.beginTransmission(address);

error = Wire.endTransmission();

if (error == 0) {

Serial.print("I2C device found at address 0x");

if (address<16) {

Serial.print("0");

}

Serial.println(address,HEX);

nDevices++;

}

else if (error==4) {

Serial.print("Unknow error at address 0x");

if (address<16) {

Serial.print("0");

}

Serial.println(address,HEX);

}

}

if (nDevices == 0) {

Serial.println("No I2C devices found\n");

}

else {

Serial.println("done\n");

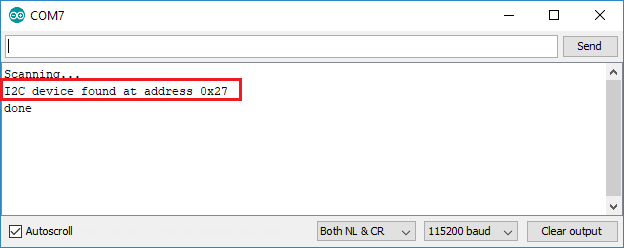
}

delay(5000);

}

[View raw code](https://github.com/RuiSantosdotme/Random-Nerd-Tutorials/raw/master/Projects/LCD_I2C/I2C_Scanner.ino)

After uploading the code, open the Serial Monitor at a baud rate of 115200. Press the ESP32 EN button. The I2C address should be displayed in the Serial Monitor.



In this case the address is **0x27**. If you’re using a similar 16×2 display, you’ll probably get the same address.

**Display Static Text on the LCD**

Displaying static text on the LCD is very simple. All you have to do is select where you want the characters to be displayed on the screen, and then send the message to the display.

Here’s a very simple sketch example that displays “**Hello, World!**“.

/\*\*\*\*\*\*\*\*\*

Rui Santos

Complete project details at https://randomnerdtutorials.com

\*\*\*\*\*\*\*\*\*/

#include <LiquidCrystal\_I2C.h>

// set the LCD number of columns and rows

int lcdColumns = 16;

int lcdRows = 2;

// set LCD address, number of columns and rows

// if you don't know your display address, run an I2C scanner sketch

LiquidCrystal\_I2C lcd(0x27, lcdColumns, lcdRows);

void setup(){

// initialize LCD

lcd.init();

// turn on LCD backlight

lcd.backlight();

}

void loop(){

// set cursor to first column, first row

lcd.setCursor(0, 0);

// print message

lcd.print("Hello, World!");

delay(1000);

// clears the display to print new message

lcd.clear();

// set cursor to first column, second row

lcd.setCursor(0,1);

lcd.print("Hello, World!");

delay(1000);

lcd.clear();

}

[View raw code](https://github.com/RuiSantosdotme/Random-Nerd-Tutorials/raw/master/Projects/LCD_I2C/I2C_LCD_Display_Text.ino)

It displays the message in the first row, and then in the second row.



In this simple sketch we show you the most useful and important functions from the LiquidCrystal\_I2C library. So, let’s take a quick look at how the code works.

**How the code works**

First, you need to include theLiquidCrystal\_I2C library.

#include <LiquidCrystal\_I2C.h>

The next two lines set the number of columns and rows of your LCD display. If you’re using a display with another size, you should modify those variables.

int lcdColumns = 16;

int lcdRows = 2;

Then, you need to set the display address, the number of columns and number of rows. You should use the display address you’ve found in the previous step.

LiquidCrystal\_I2C lcd(0x27, lcdColumns, lcdRows);

In the setup(), first initialize the display with the init() method.

lcd.init();

Then, turn on the LCD backlight, so that you’re able to read the characters on the display.

lcd.backlight();

To display a message on the screen, first you need to set the cursor to where you want your message to be written. The following line sets the cursor to the first column, first row.

lcd.setCursor(0, 0);

**Note**: 0 corresponds to the first column, 1 to the second column, and so on…

Then, you can finally print your message on the display using the print() method.

lcd.print("Hello, World!");

Wait one second, and then clean the display with the clear() method.

lcd.clear();

After that, set the cursor to a new position: first column, second row.

lcd.setCursor(0,1);

Then, the process is repeated.

So, here’s a summary of the functions to manipulate and write on the display:

* lcd.init(): initializes the display
* lcd.backlight(): turns the LCD backlight on
* lcd.setCursor(int column, int row): sets the cursor to the specified column and row
* lcd.print(String message): displays the message on the display
* lcd.clear(): clears the display

This example works well to display static text no longer than 16 characters.

**Display Scrolling Text on the LCD**

Scrolling text on the LCD is specially useful when you want to display messages longer than 16 characters. The library comes with built-in functions that allows you to scroll text. However, many people experience problems with those functions because:

* The function scrolls text on both rows. So, you can’t have a fixed row and a scrolling row;
* It doesn’t work properly if you try to display messages longer than 16 characters.

So, we’ve created a sample sketch with a function you can use in your projects to scroll longer messages.

The following sketch displays a static message in the first row and a scrolling message longer than 16 characters in the second row.

/\*\*\*\*\*\*\*\*\*

Rui Santos

Complete project details at https://randomnerdtutorials.com

\*\*\*\*\*\*\*\*\*/

#include <LiquidCrystal\_I2C.h>

// set the LCD number of columns and rows

int lcdColumns = 16;

int lcdRows = 2;

// set LCD address, number of columns and rows

// if you don't know your display address, run an I2C scanner sketch

LiquidCrystal\_I2C lcd(0x27, lcdColumns, lcdRows);

String messageStatic = "Static message";

String messageToScroll = "This is a scrolling message with more than 16 characters";

// Function to scroll text

// The function acepts the following arguments:

// row: row number where the text will be displayed

// message: message to scroll

// delayTime: delay between each character shifting

// lcdColumns: number of columns of your LCD

void scrollText(int row, String message, int delayTime, int lcdColumns) {

for (int i=0; i < lcdColumns; i++) {

message = " " + message;

}

message = message + " ";

for (int pos = 0; pos < message.length(); pos++) {

lcd.setCursor(0, row);

lcd.print(message.substring(pos, pos + lcdColumns));

delay(delayTime);

}

}

void setup(){

// initialize LCD

lcd.init();

// turn on LCD backlight

lcd.backlight();

}

void loop(){

// set cursor to first column, first row

lcd.setCursor(0, 0);

// print static message

lcd.print(messageStatic);

// print scrolling message

scrollText(1, messageToScroll, 250, lcdColumns);

}

[View raw code](https://github.com/RuiSantosdotme/Random-Nerd-Tutorials/raw/master/Projects/LCD_I2C/I2C_LCD_Example.ino)

After reading the previous section, you should be familiar on how this sketch works, so we’ll just take a look at the newly created function: scrollText()

void scrollText(int row, String message, int delayTime, int lcdColumns) {

for (int i=0; i < lcdColumns; i++) {

message = " " + message;

}

message = message + " ";

for (int pos = 0; pos < message.length(); pos++) {

lcd.setCursor(0, row);

lcd.print(message.substring(pos, pos + lcdColumns));

delay(delayTime);

}

}

To use this function you should pass four arguments:

* row: row number where the text will be display
* message: message to scroll
* delayTime: delay between each character shifting. Higher delay times will result in slower text shifting, and lower delay times will result in faster text shifting.
* lcdColumns: number of columns of your LCD

In our code, here’s how we use the scrollText() function:

scrollText(1, messageToScroll, 250, lcdColumns);

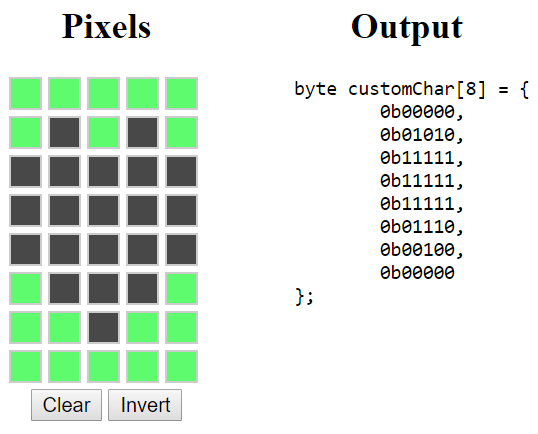
The messageToScroll variable is displayed in the second row (1 corresponds to the second row), with a delay time of 250 ms (the GIF image is speed up 1.5x).



**Display Custom Characters**

In a 16×2 LCD there are 32 blocks where you can display characters. Each block is made out of 5×8 tiny pixels. You can display custom characters by defining the state of each tiny pixel. For that, you can create a byte variable to hold  the state of each pixel.

To create your custom character, you can go [here](https://omerk.github.io/lcdchargen/) to generate the byte variable for your character. For example, a heart:



Copy the byte variable to your code (before the setup()). You can call it heart:

byte heart[8] = {

0b00000,

0b01010,

0b11111,

0b11111,

0b11111,

0b01110,

0b00100,

0b00000

};

Then, in the setup(), create a custom character using the createChar() function. This function accepts as arguments a location to allocate the char and the char variable as follows:

lcd.createChar(0, heart);

Then, in the loop(), set the cursor to where you want the character to be displayed:

lcd.setCursor(0, 0);

Use the write() method to display the character. Pass the location where the character is allocated, as follows:

lcd.write(0);