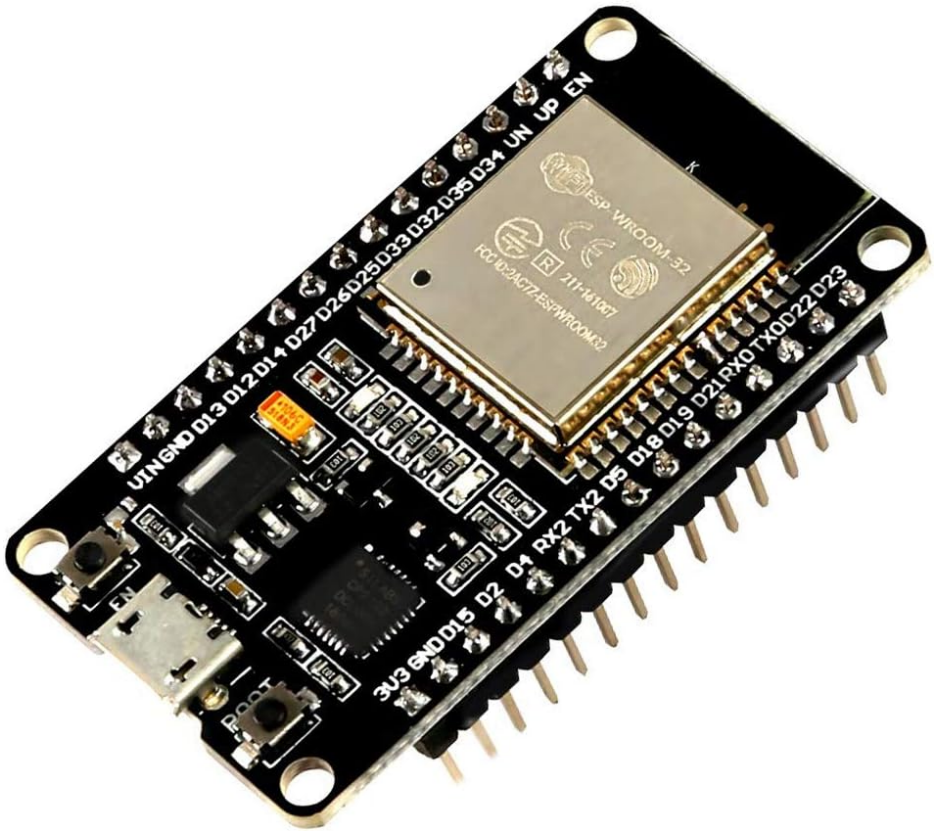


Microcontroller (ESP-32)

Jubair Ahammad Akter

Department of Computer Science and Engineering
University of Dhaka

February 24, 2026

Contents

1 Introduction to ESP32 Microcontroller

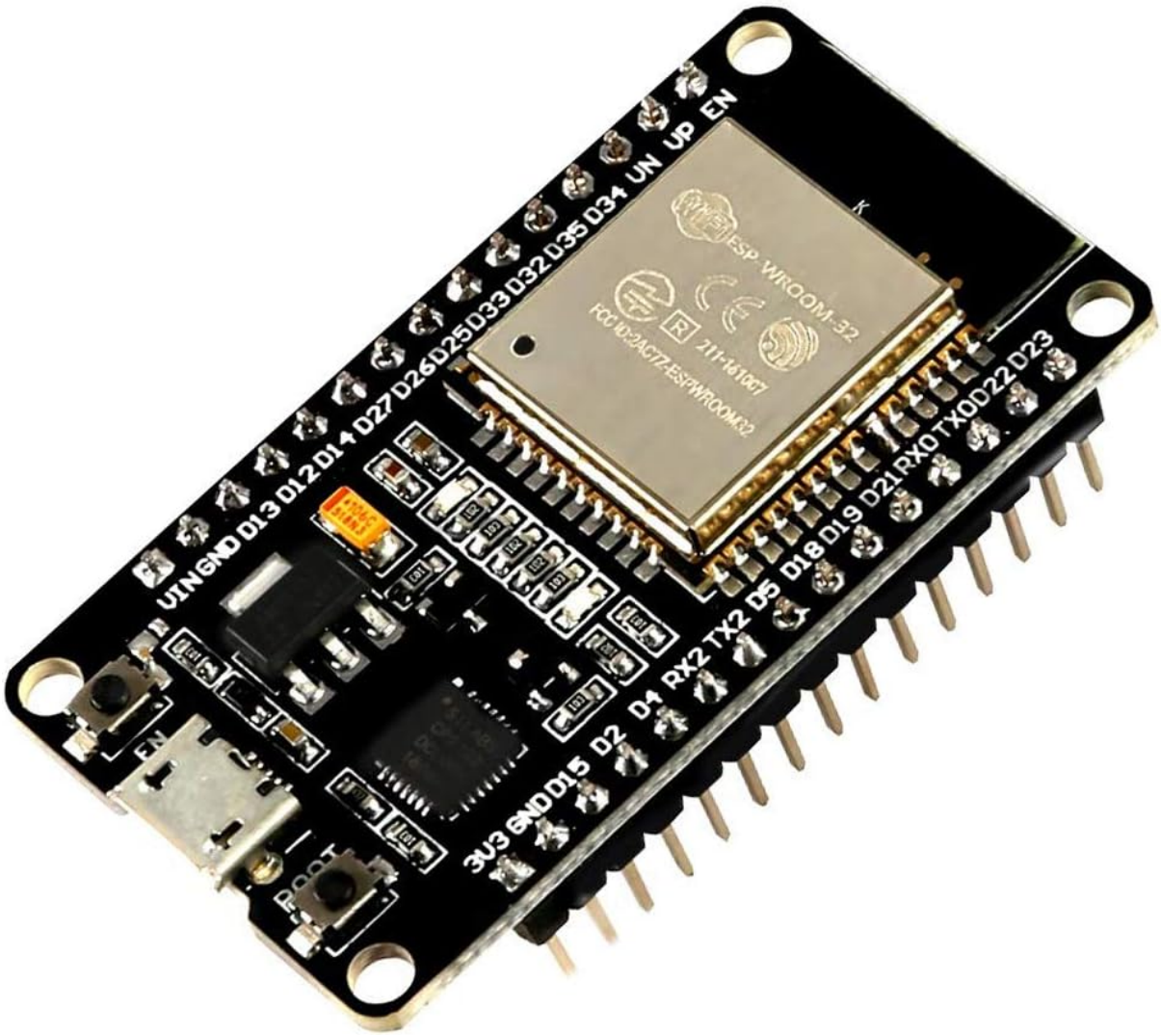


Figure 1: ESP32 Development Board

The **ESP32** is a powerful, low-cost, and versatile microcontroller designed by Espressif Systems. It is the successor to the ESP8266 and features built-in **Wi-Fi** and **Bluetooth** capabilities, making it ideal for IoT (Internet of Things) applications.

1.1 Key Features

- Dual-core 32-bit processor (Tensilica Xtensa LX6)
- Clock speed up to 240 MHz
- 520 KB SRAM, up to 16 MB Flash memory
- 34 GPIO pins with multiple functions

- 18-channel ADC (12-bit), 2-channel DAC
- UART, SPI, I2C, PWM, and other communication protocols
- Built-in Wi-Fi and Bluetooth (Classic + BLE)

1.2 ESP32 Development Boards

Common variants include:

- ESP32-WROOM-32 (most popular)
- ESP32 DevKit v1 (NodeMCU-32S)
- ESP32-S2, ESP32-C3, ESP32-S3

1.3 Pin Diagram and Board Photo

Below are placeholders where you can insert images:

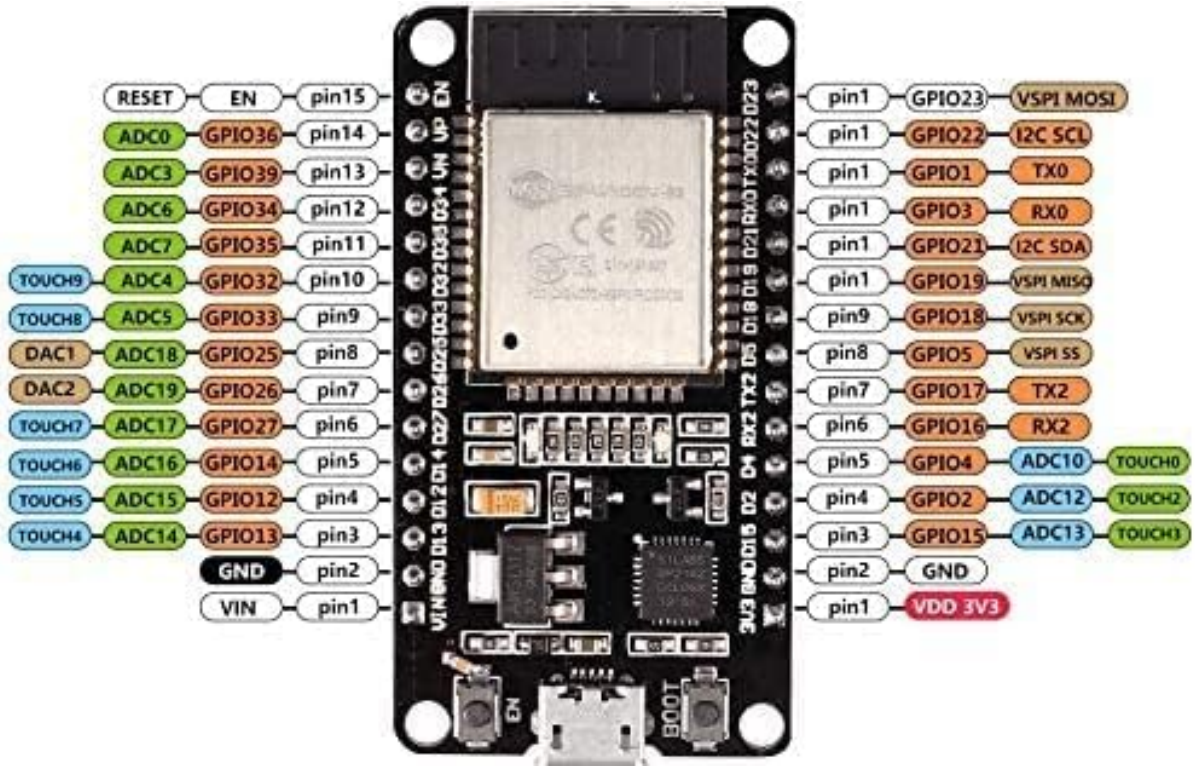


Figure 2: ESP32 Pin Diagram

1.4 Extra

Figure 3: ESP32 Pinout Diagram

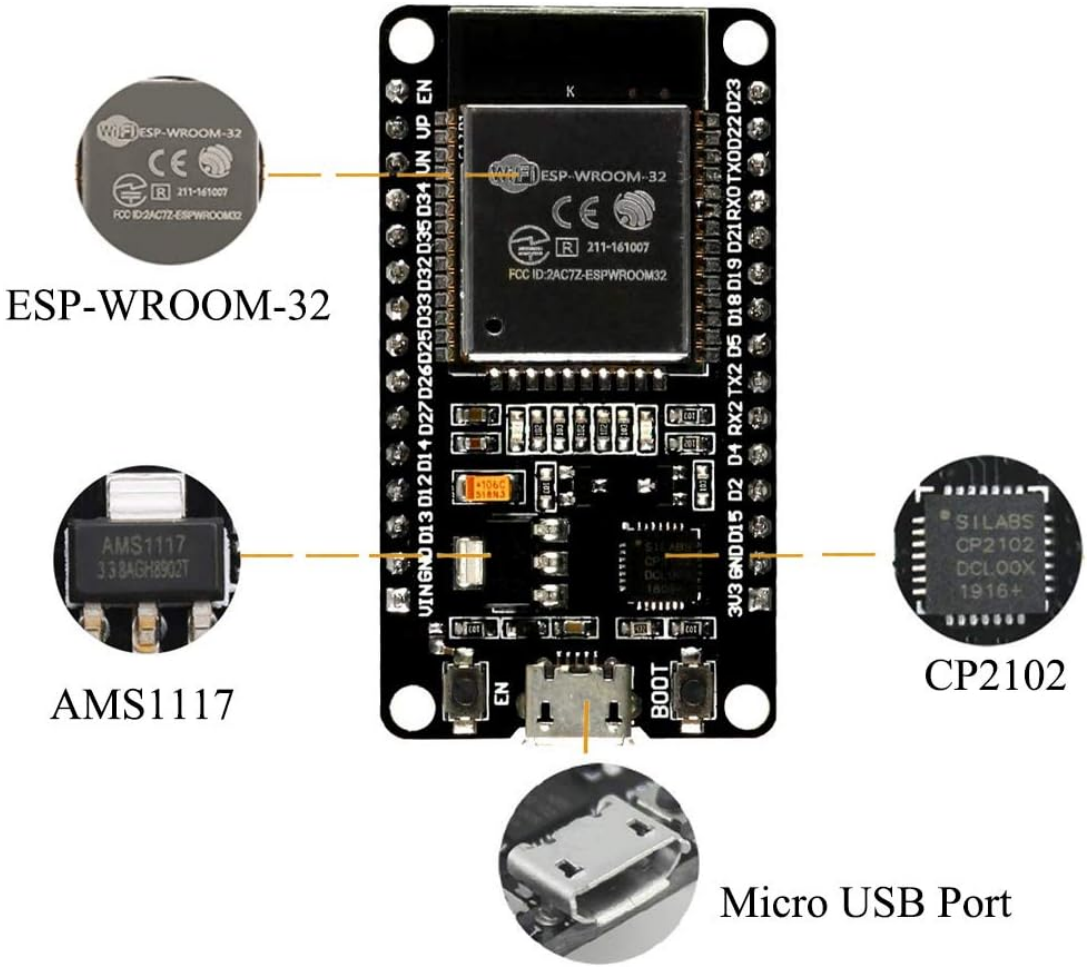


Figure 4: ESP32 Zoomed

2 Setting Up ESP32 on Arduino IDE

2.1 Step 1: Install Arduino IDE

You can install Arduino IDE in two ways — from the official website or directly through the Ubuntu terminal.

Option A: Download from Official Website

Download the latest Arduino IDE from: https://www.arduino.cc/en/software

Option B: Install via Ubuntu Terminal

Open a terminal and run the following commands one by one:

Ubuntu Terminal (Commands)

```
# Update package list
sudo apt update

# Install dependencies
sudo apt install wget tar python3 -y

# Download the latest Arduino IDE (adjust version if needed)
wget
   https://downloads.arduino.cc/arduino-ide_2.3.2_Linux_64bit.AppImage
   -O arduino-ide.AppImage

# Make the AppImage executable
chmod +x arduino-ide.AppImage

# (Optional) Move it to /usr/local/bin for easy access
sudo mv arduino-ide.AppImage /usr/local/bin/arduino-ide

# Run the Arduino IDE
arduino-ide
```

After this, Arduino IDE will open. You can also create a desktop shortcut if desired.

2.2 Step 2: Add ESP32 Board Manager URL

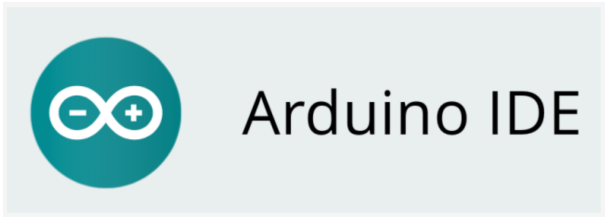
In the Arduino IDE, go to:

1. **File → Preferences**
2. In the **Additional Board Manager URLs** box, paste:

   `https://dl.espressif.com/dl/package_esp32_index.json`

2.3 Step 3: Install the ESP32 Board Package

1. Go to **Tools → Board → Boards Manager**.
2. Search for **ESP32**.



3. Click **Install**.

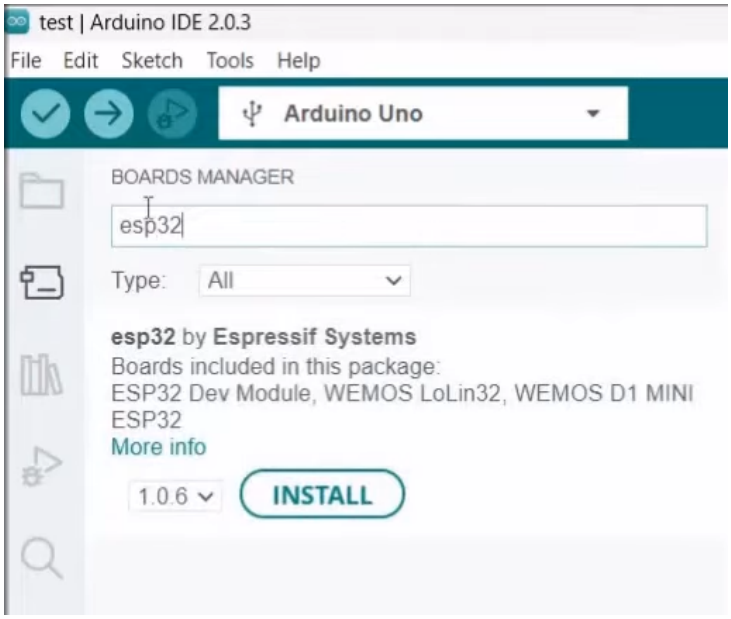


Figure 5: Step:3

2.4 Step 4: Select Board and Port

- Board: `ESP32 Dev Module`
- Port: The COM port detected after connecting your ESP32 via USB. Ensure that **C-type Data cable** (Not Charging cable) is connected with device(PC) and ESP32 microcontroller.

2.5 Step 5: Verify and Upload a Sketch

Click the **Verify** button to compile the code, and then click → **Upload** to send the program to the ESP32.

2.6 UI

Figure 6: Ardino IDE UI_1

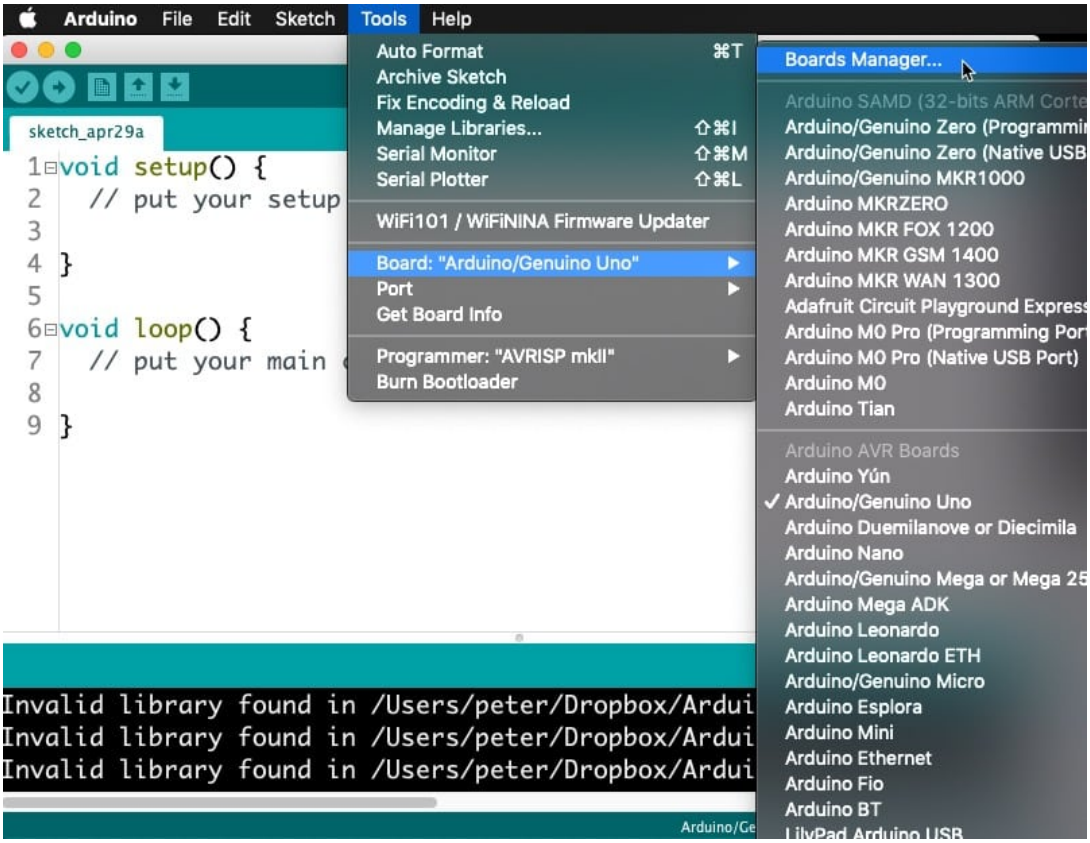


Figure 7: Ardino IDE UI_2

3 Understanding Arduino IDE Code Terms

In Arduino IDE, every program (called a sketch) is composed of two essential functions: `setup()` and `loop()`. These are automatically recognized by the Arduino compiler and determine how your program runs on the ESP32.

3.1 How setup() and loop() Work

- **setup():** This function runs **only once** when the ESP32 powers on or resets. It is used to initialize serial communication, configure pin modes, start sensors, Wi-Fi, etc.
- **loop():** After setup() finishes, the **loop() function runs repeatedly forever**. All continuous tasks such as blinking LEDs, reading sensors, and checking inputs happen here.

```
void setup() {
  // runs once at startup
}

void loop() {
  // runs repeatedly
}
```

3.2 Common Functions (Frequently Used)

- **Serial.begin(baud_rate)** – Starts serial communication.
- **Serial.print() / Serial.println()** – Prints text/values to the Serial Monitor.
- **pinMode(pin, mode)** – Sets a pin as INPUT, OUTPUT, INPUT_PULLUP.
- **digitalWrite(pin, value)** – Outputs HIGH or LOW to a pin.
- **digitalRead(pin)** – Reads HIGH/LOW from a digital pin.
- **delay(ms)** – Pauses the program for given milliseconds.
- **analogRead(pin)** – Reads analog input (0–4095 for ESP32 ADC).
- **analogWrite(pin, value)** – PWM output (0–255) for dimming LEDs or motor speed.
- **millis()** – Returns system uptime in milliseconds (used instead of delay).
- **ledcWrite(channel, dutyCycle)** – ESP32 PWM output using LED Controller.

- **attachInterrupt()** – Used to trigger code when an input pin changes.

These functions form the core of most ESP32 practical tasks.

3.3 Example of Basic Structure

```
int LED_BUILTIN = 13; // GPIO pin number

void setup() {
  Serial.begin(115200);  // Start serial communication
  pinMode(LED_BUILTIN, OUTPUT);    // Set GPIO 13 as output
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // Turn LED on
  delay(1000);             // Wait for 1 second
  digitalWrite(LED_BUILTIN, LOW);  // Turn LED off
  delay(1000);             // Wait for 1 second
}
```

3.4 Line-by-Line Explanation of the Code

Inside setup()

- `Serial.begin(115200);` Initializes serial communication at **115200 baud rate**. This enables printing messages to the Serial Monitor.
- `pinMode(LED_BUILTIN, OUTPUT);` Configures **GPIO 13** as an **OUTPUT pin**. If we set the LED_BUILTIN = 2, then this will allow us to control an LED connected to pin 2.

Inside loop()

- `digitalWrite(LED_BUILTIN, HIGH);` Sends 3.3V (or HIGH signal) to pin 13, turning the built-in LED **ON**.
- `delay(1000);` Waits for **1 second** (1000 ms). During this time, the program is paused.
- `digitalWrite(LED_BUILTIN, LOW);` Sends 0V (LOW signal) to pin 13, turning the built-in LED **OFF**.
- `delay(1000);` Again waits for **1 second**, keeping the LED OFF.

After finishing all lines in `loop()`, the function restarts from the top, causing the built-in LED to blink ON and OFF continuously.

4 Basic LED Blink Using D2 Pin

4.1 Circuit Diagram

- Just connect the microcontroller and PC with data cable. After

4.2 Code

```
// Basic Blink on D2
int led = 2; // D2 pin

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH); // LED ON
  delay(1000);             // wait 1 second
  digitalWrite(led, LOW);  // LED OFF
  delay(1000);             // wait 1 second
}
```

4.3 Explanation

- `pinMode(2, OUTPUT)` Carefully view the upper portion of Blue light, there is written D2. If you set D2 then you can view the output from here, otherwise you need extra equipments.

4.4 Photo / Diagram

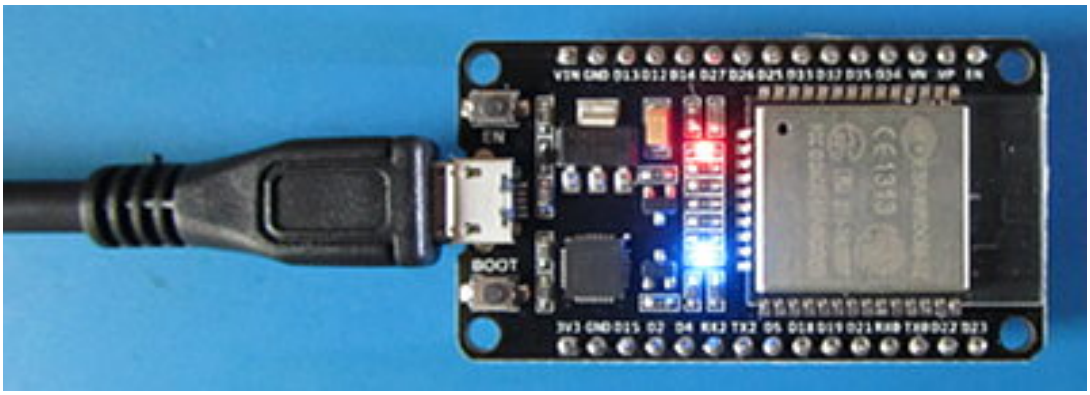


Figure 8: Blink with PIN D2

5 Blink with LED, Resistor and Buzzer

5.1 Why We Use a Resistor with an LED

- ESP32 GPIO pins can supply only **12–20mA** safely.
- Without a resistor, the LED can draw excessive current and may burn out or damage the GPIO pin.
- A **100 resistor** limits the current to a safe level.

5.2 If Not Using D2

If you change the pin number in the code to **4** and connect your LED (with the resistor) to **D4** instead of D2, then the small blue onboard LED on D2 will stop blinking. Your external LED on D4 will blink instead. Always be **careful** about the LED orientation:

- **Anode (+)** → GPIO pin
- **Cathode (−)** → GND through the resistor

5.3 Safe Recommendation

- For beginners, always use stable GPIO pins: **2, 4, 5, 18, 19, 21, 22, 23**.
- Avoid boot-related pins (**0, 2, 12, 15**) unless you know what you are doing.
- Avoid SPI pins (**13, 14, 18, 19**) if you plan to use SPI devices later.

5.4 If You Want to Attach a Buzzer

For an **active buzzer**:

- Connect **buzzer + (positive)** to the GPIO output pin (D2 or D4).
- Connect **buzzer − (negative)** to GND.

Do NOT connect the buzzer directly to 3V3 and GPIO at the same time — that can create back-powering and damage the board.

5.5 Code

Completely same as the old, but you should have to be careful about the output pin number just.

5.6 Photo / Diagram

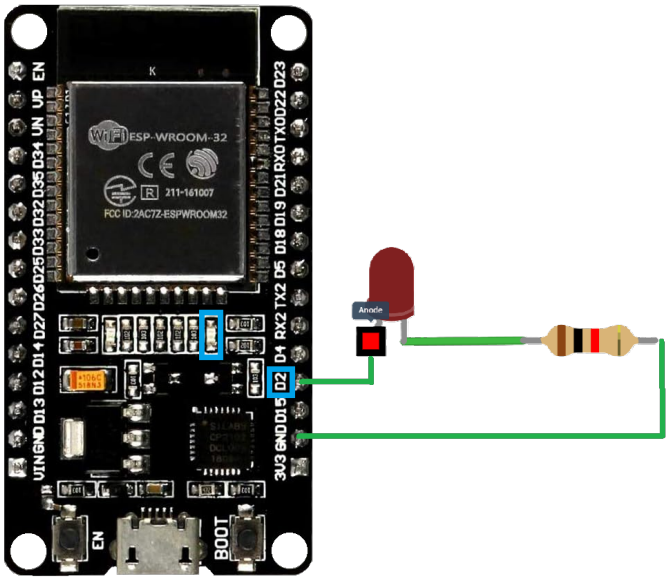


Figure 9: Blink with PIN D2 (LED + Resistor)

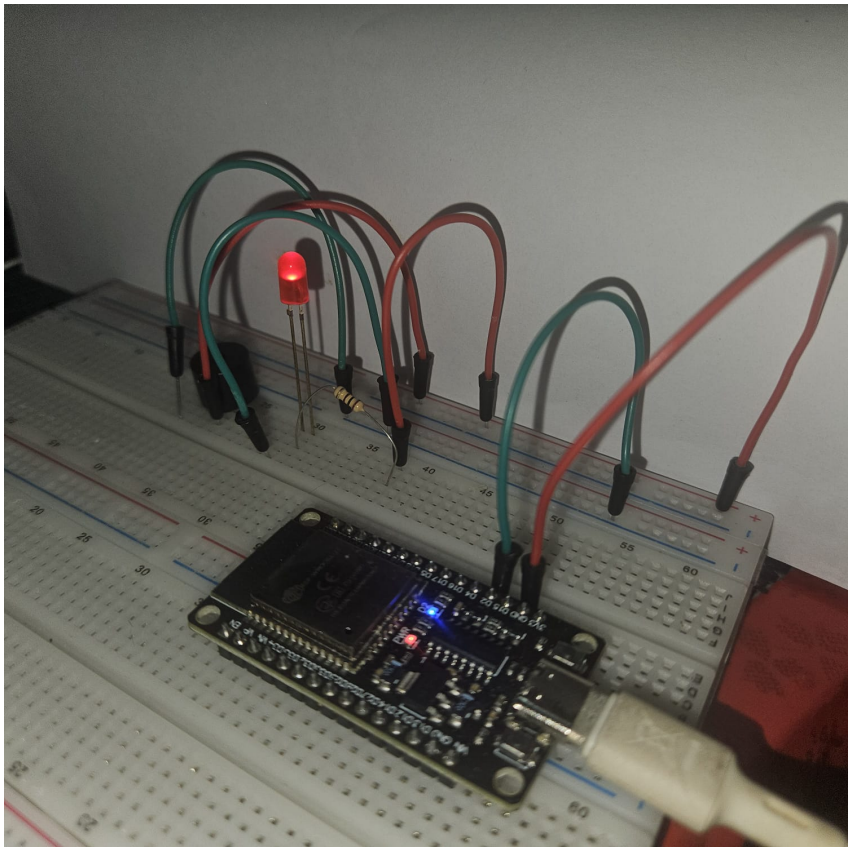


Figure 10: Anode + (GREEN wire) + D2 , Cathode + (RED wire) + GND

6 LED Control Using Push Button (INPUT_PULLUP)

6.1 Connection Setup

To control an LED using a push button on the ESP32, make the following connections:

- For the LED + Registor:
  - Connect the **LED anode (+)** to GPIO **4** (D4).
  - Connect the **LED cathode (−)** to **GND** through a 100 ohm resistor.
- For the push button:
  - Connect one side of the button to **GPIO 2** (D2).
  - Connect the opposite side of the button to **3V3**.

6.2 Code

```
// LED + Button Control Using INPUT_PULLUP

int ledPin = 4;      // LED connected to GPIO 4
int buttonPin = 2;   // Button connected to GPIO 2

void setup() {
  pinMode(ledPin, OUTPUT);           // LED as output
  pinMode(buttonPin, INPUT_PULLUP); // Button as input with
     internal pull-up
  // Button wiring: one side     GND, other side     D2
}

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

  // Button pressed = LOW (because of pull-up)
  if (buttonState == HIGH) {
    digitalWrite(ledPin, HIGH);  // LED ON
  } else {
    digitalWrite(ledPin, LOW);   // LED OFF
  }
}
```

6.3 Important Notes

- Ensure you use the **opposite legs** of the push button (cross-pair), not the same side.
- Always check LED orientation:
  - Long leg = **Anode** (+)
  - Short leg = **Cathode** (–)

6.4 Why We Need a Pull-Down Resistor on the Button Input

When a button is connected between the GPIO pin and **3V3**, the input pin must have a defined LOW state when the button is not pressed. Without a pull-down resistor, the pin is left **floating**, meaning:

- The GPIO pin may randomly read HIGH or LOW.
- Small electrical noise can trigger false button presses.
- The program behaves unpredictably, causing unstable LED or buzzer operation.

A **10k pull-down resistor** solves this issue by:

- Holding the GPIO pin at a stable LOW (0V) when the button is not pressed.
- Allowing the pin to go HIGH only when the button connects it to 3V3.
- Ensuring clean and reliable digital input readings.

In short, a pull-down resistor ensures the input pin is never floating and provides a clear, stable logic level for detecting button presses.

6.5 Photo / Diagram

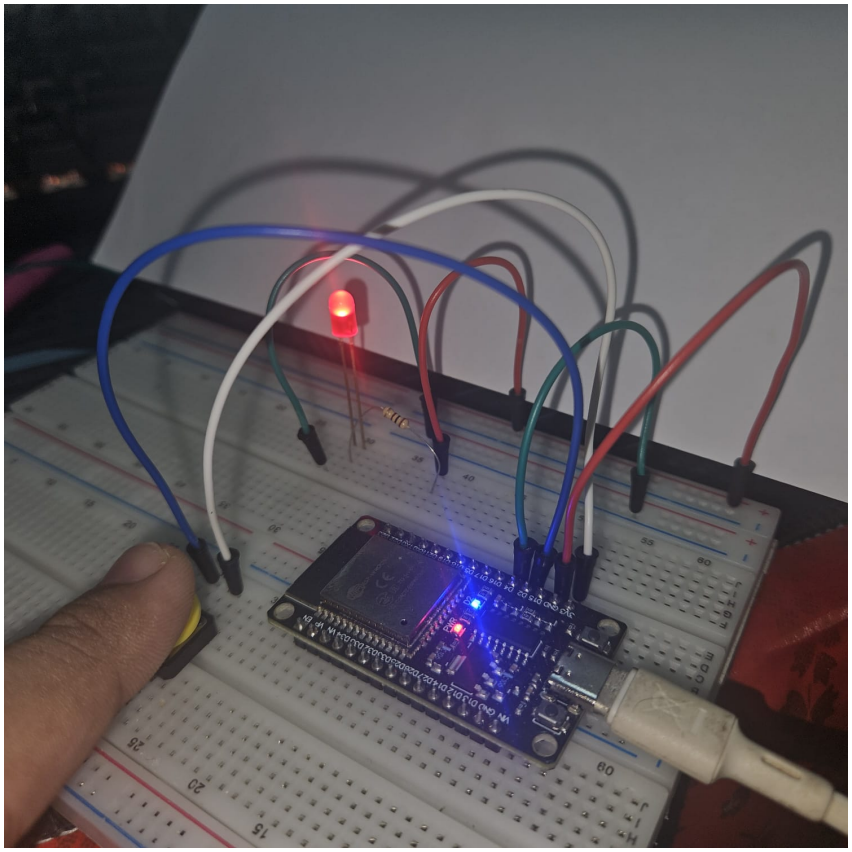


Figure 11: Button Controlling LED on GPIO 4 Using INPUT_PULLUP

7 External Interrupt Controlled LED Toggle

7.1 Code

```
// Button on GPIO 2, LED on GPIO 4
const int buttonPin = 2;
const int ledPin    = 4;

void IRAM_ATTR toggleISR() {
  digitalWrite(ledPin, !digitalRead(ledPin));
}

void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, LOW);  // initialize LED off

  attachInterrupt(digitalPinToInterrupt(buttonPin),
                  toggleISR,
                  FALLING);
}

void loop() {
  // Nothing here     LED toggling handled in ISR
}
```

7.2 Line-by-Line Explanation

- `const int buttonPin = 2;` Defines the constant `buttonPin` to be GPIO 2 where the push-button is connected.
- `const int ledPin = 4;` Defines the constant `ledPin` to be GPIO 4 where the LED is connected.
- `void IRAM_ATTR toggleISR()` Declares the interrupt service routine (ISR) function named `toggleISR`. The macro `IRAM_ATTR` ensures the function is placed in fast instruction RAM on the ESP32, which is important for reliable interrupt handling. :contentReference[oaicite:0]index=0
- `digitalWrite(ledPin, !digitalRead(ledPin));` Inside the ISR: reads the current state of the LED pin, inverts it (using `!`), and writes the inverted state back. This toggles the LED each time the ISR is called.

- `void setup()` The setup function runs once at start.
- `pinMode(buttonPin, INPUT_PULLUP);` Configures the button pin as an input with the internal pull-up resistor enabled. This means the pin will normally read HIGH, and will go LOW when the button is pressed (if wired to GND).
- `pinMode(ledPin, OUTPUT);` Configures the LED pin as an output.
- `digitalWrite(ledPin, LOW);` Initializes the LED to be OFF at startup.
- `attachInterrupt(digitalPinToInterrupt(buttonPin), toggleISR, FALLING);` Attaches an external interrupt to `buttonPin`. When a FALLING edge (HIGH→LOW) is detected on the button pin, the function `toggleISR` will be executed. Because the input uses pull-up, the falling edge corresponds to the button being pressed. :contentReference[oaicite:1]index=1
- `void loop()` The main loop is empty so that the microcontroller is free to perform other tasks while the ISR handles the LED toggling.

8 Stepper Motor Control (28BYJ-48 + ULN2003 Driver)

8.1 Overview

In this experiment, we control the **28BYJ-48 (5V) stepper motor** using the **ULN2003 driver board** and an **ESP32**. The motor rotates **clockwise** and then **counterclockwise** using a **half-step sequence** for smoother motion.

8.2 Required Components

- ESP32 Development Board
- 28BYJ-48 Stepper Motor (5V)
- ULN2003 Driver Module
- External 5V Power Supply (recommended)
- Jumper wires

8.3 Wiring / Connections

- **ULN2003 IN1** → ESP32 GPIO **13**
- **ULN2003 IN2** → ESP32 GPIO **14**
- **ULN2003 IN3** → ESP32 GPIO **12**
- **ULN2003 IN4** → ESP32 GPIO **27**
- **ULN2003 VCC** → **5V External Supply**
- **ULN2003 GND** → **GND (External)** and must be **common GND with ESP32**

Important: Do not power the 28BYJ-48 motor directly from ESP32 3.3V pin. Use external 5V.

8.4 Procedure (Steps)

1. Connect the 28BYJ-48 motor to the ULN2003 driver using the provided connector.
2. Connect ULN2003 IN1–IN4 to ESP32 GPIO pins (13, 14, 12, 27).
3. Provide 5V external power to ULN2003 VCC and connect GND.
4. Make sure ESP32 GND and external GND are connected (common ground).

5. Upload the code below in Arduino IDE.

6. The motor will rotate clockwise, pause, then rotate counterclockwise.

8.5 Code (Half-Step Rotation: CW and CCW)

```
// 28BYJ-48 Stepper Motor Control with ESP32 (ULN2003)
// IN1-IN4 are control pins from ESP32 to ULN2003

// Pins for ESP32
const int IN1 = 13;
const int IN2 = 14;
const int IN3 = 12;
const int IN4 = 27;

// Half-step sequence for 28BYJ-48
const int steps = 8;
int seq[8][4] = {
  {1,0,0,0},
  {1,1,0,0},
  {0,1,0,0},
  {0,1,1,0},
  {0,0,1,0},
  {0,0,1,1},
  {0,0,0,1},
  {1,0,0,1}
};

void setup() {
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
}

void loop() {
  // Clockwise rotation: 2048 half-steps ~= 1 revolution
  for (int i = 0; i < steps * 256; i++) { // 8*256 = 2048
    stepCW();
    delay(2); // speed control (smaller = faster)
  }

  delay(500);
```

```
  // Counterclockwise rotation
  for (int i = 0; i < steps * 256; i++) {
    stepCCW();
    delay(2);
  }

  delay(500);
}

// Clockwise step
void stepCW() {
  static int step = 0;
  digitalWrite(IN1, seq[step][0]);
  digitalWrite(IN2, seq[step][1]);
  digitalWrite(IN3, seq[step][2]);
  digitalWrite(IN4, seq[step][3]);
  step = (step + 1) % 8;
}

// Counterclockwise step
void stepCCW() {
  static int step = 0;
  digitalWrite(IN1, seq[step][0]);
  digitalWrite(IN2, seq[step][1]);
  digitalWrite(IN3, seq[step][2]);
  digitalWrite(IN4, seq[step][3]);
  step = (step - 1 + 8) % 8; // fixed line
}
```

8.6 How It Works (Short Explanation)

- Stepper motors rotate by energizing coils in a specific order.
- The array `seq[8][4]` stores the **half-step pattern**.
- `stepCW()` moves forward through the sequence for clockwise motion.
- `stepCCW()` moves backward through the sequence for counterclockwise motion.
- `delay(2)` controls the speed (reduce delay to increase speed).