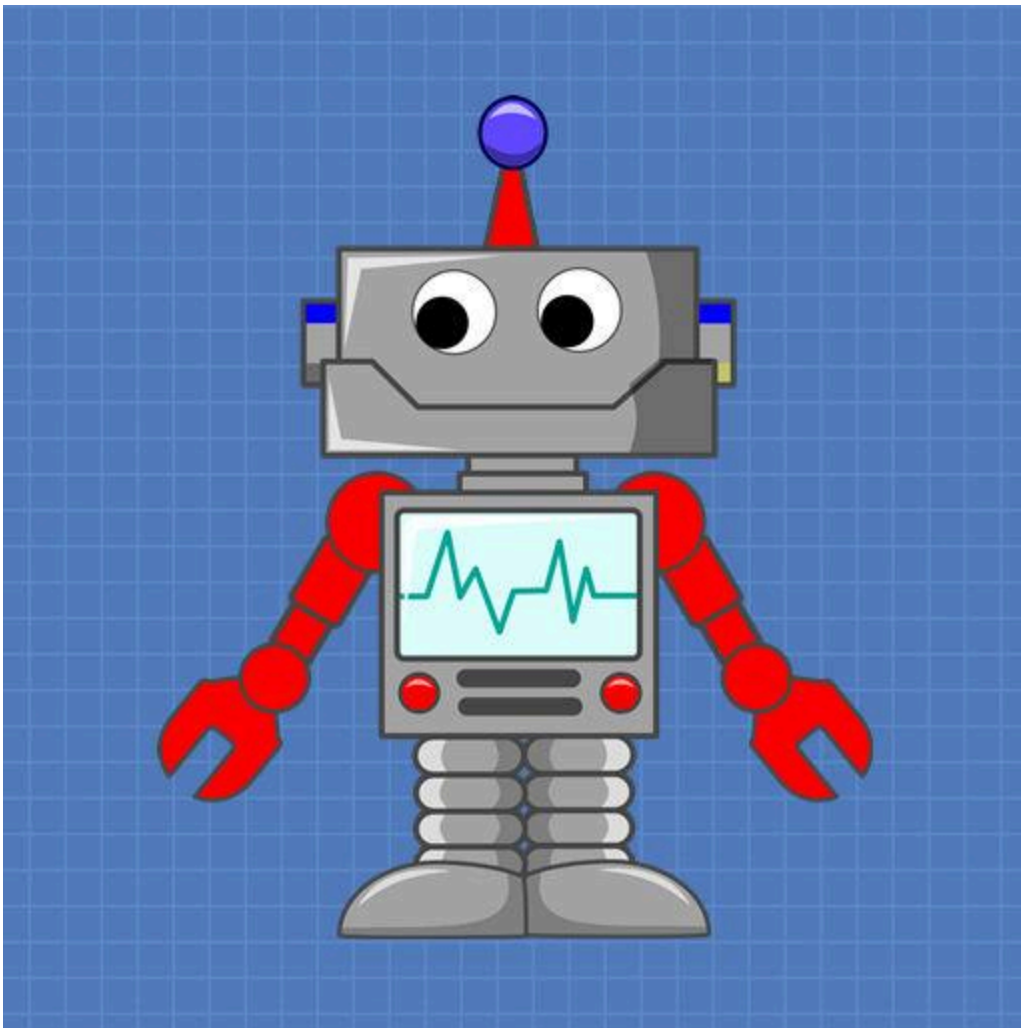


I2C Tricks and Tips with ESP32

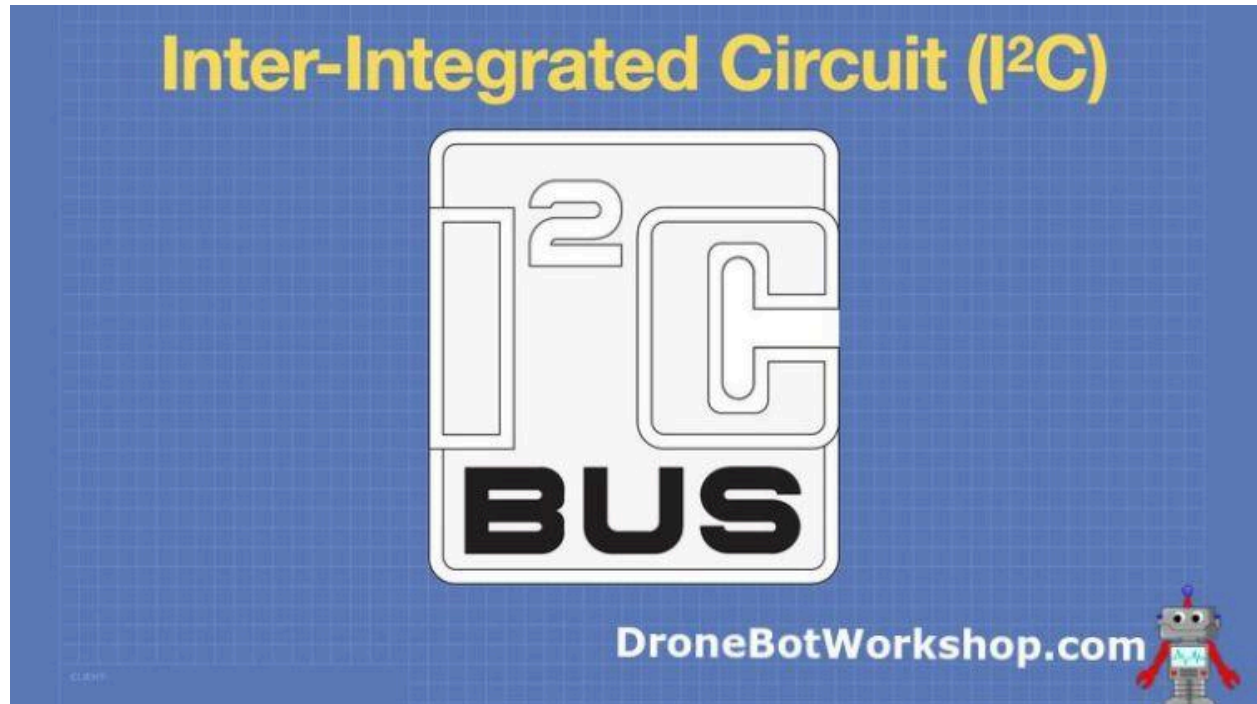


DroneBot Workshop Tutorial

<https://dronebotworkshop.com>

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

Today, we will take a deep dive into using I²C with the ESP32 microcontroller. We'll examine the I²C implementation in the ESP32 and see how we can use it as a Controller, Peripheral, and with multiple I²C buses.



Introduction

The I²C (Inter-Integrated Circuit) bus is one of the most versatile and popular communication protocols used by microcontrollers. Whether you're connecting sensors, displays, or other peripherals to your ESP32, understanding how to use I²C effectively can significantly expand your project possibilities.

<https://dronebotworkshop.com>



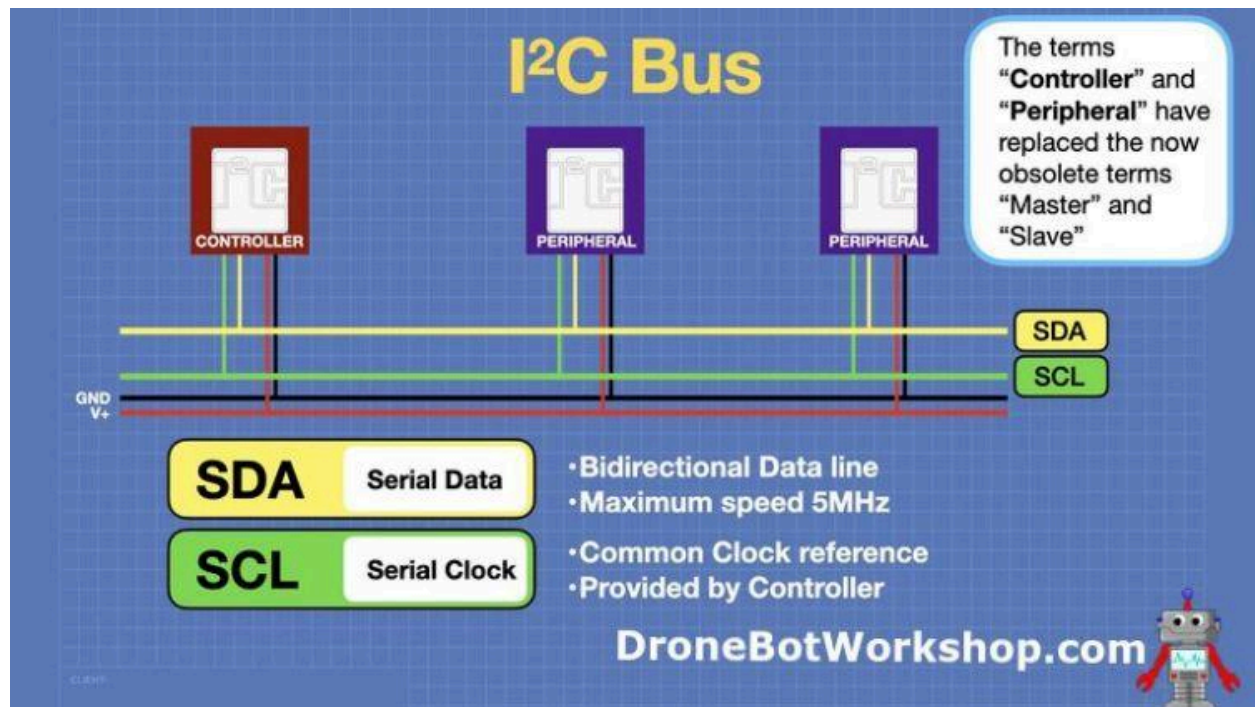
In this article and accompanying video, we'll explore the ESP32's I²C capabilities from the ground up. We'll start with the fundamentals of the I²C protocol, then dive into the ESP32's specific implementation. You'll learn how to connect and communicate with I²C devices, utilize multiple I²C buses simultaneously, and even configure your ESP32 to act as an I²C peripheral device. That last task will allow you to start designing your own custom I²C Peripherals.

Whether you're a student, hobbyist, or an experienced maker, you'll find something useful in this practical guide to I²C on the ESP32. Let's get started!

The I²C Bus

The **Inter-Integrated Circuit (I²C or I2C) bus** is a simple, low-speed, short-distance communication protocol developed by Philips (now NXP) in 1982. It is widely used for connecting sensors, displays, EEPROMs, ADCs/DACs, real-time clocks, and other peripherals to microcontrollers such as Arduino and ESP32. Unlike UART or SPI, I²C uses only **two wires** for communication:

- **SDA (Serial Data line)** – carries the data.
- **SCL (Serial Clock line)** – carries the clock signal.



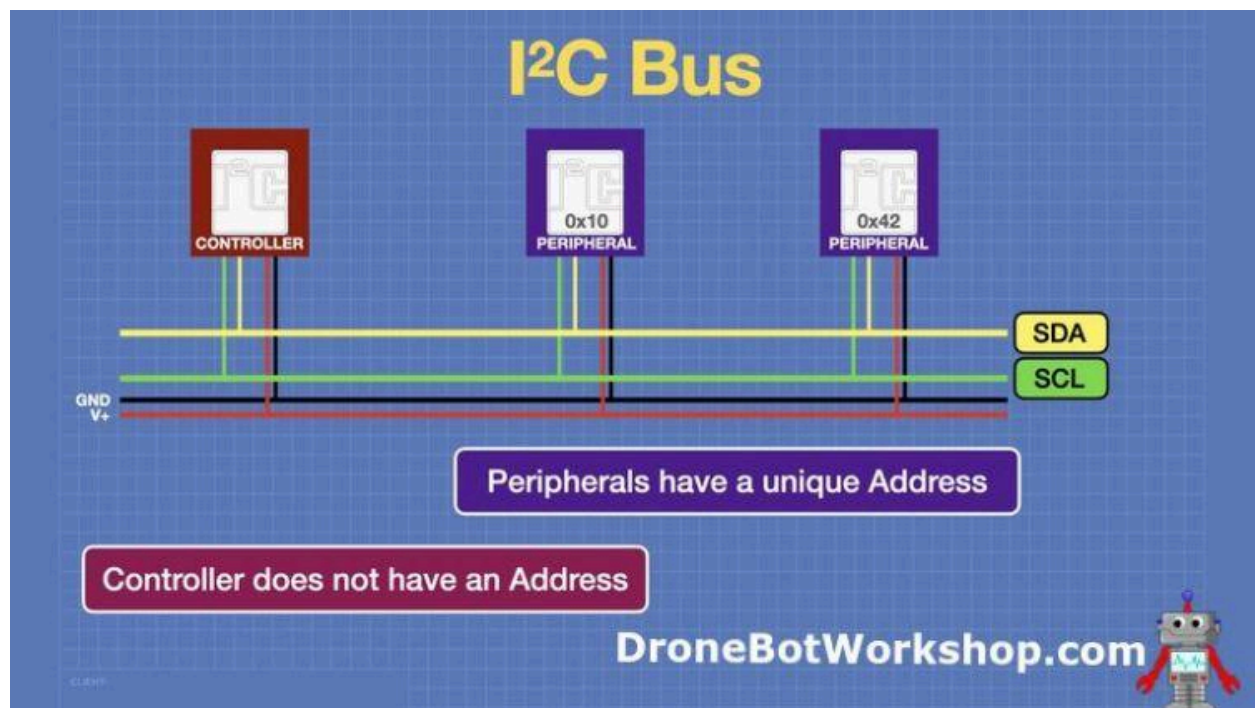
The bus also included wires for VCC (3.3 or 5 volts) and Ground.

I²C devices can be divided into two categories:

- **Controller** – This is the host of the I²C bus. It provides the clock signal on the SCL line and initiates all communications. The Controller was formerly called the "Master".

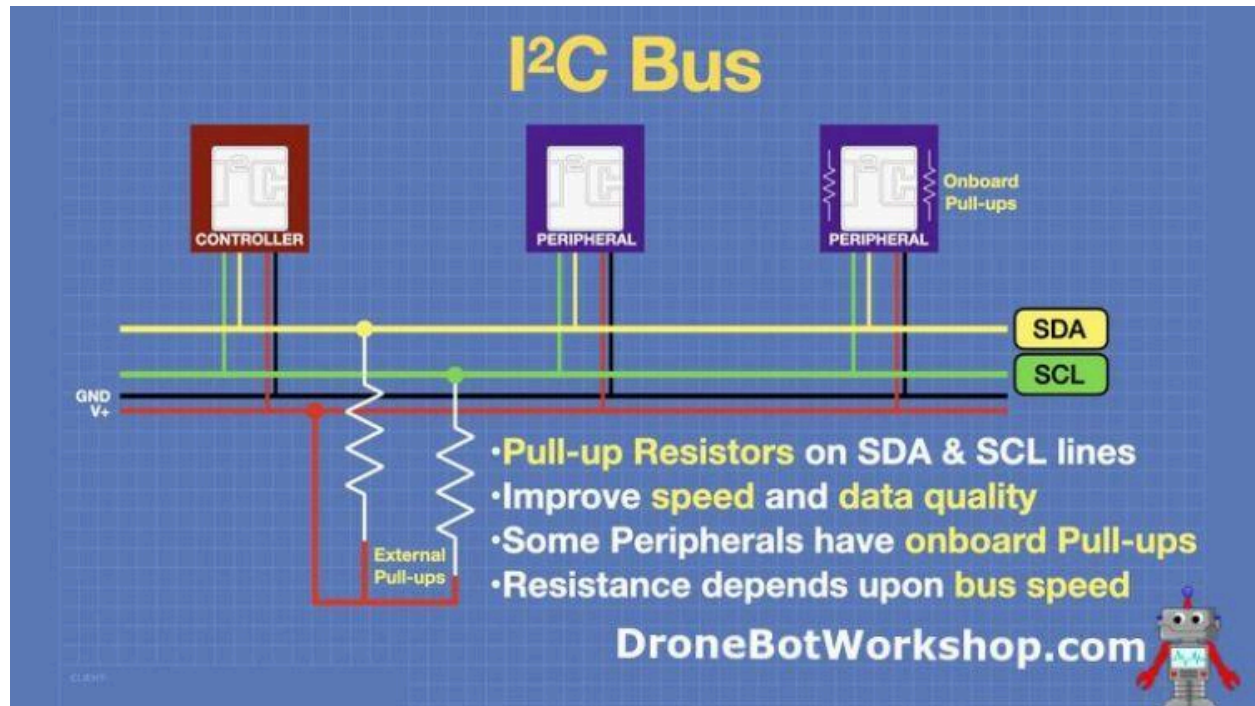
- **Peripheral** – These are the “clients” of the bus. They respond to communications from the Controller. Peripherals can be input or output devices, and examples include sensors and displays. The Peripheral was formerly referred to as a “Slave” device.

Each Peripheral on the I²C bus has a unique 7-bit address (there are also 10-bit address I²C devices). The Controller uses this address to talk to specific peripherals. It is possible to have multiple controllers, but only one controller can be active at a time. Multiple controllers are not common, and we won't be implementing this type of configuration today.

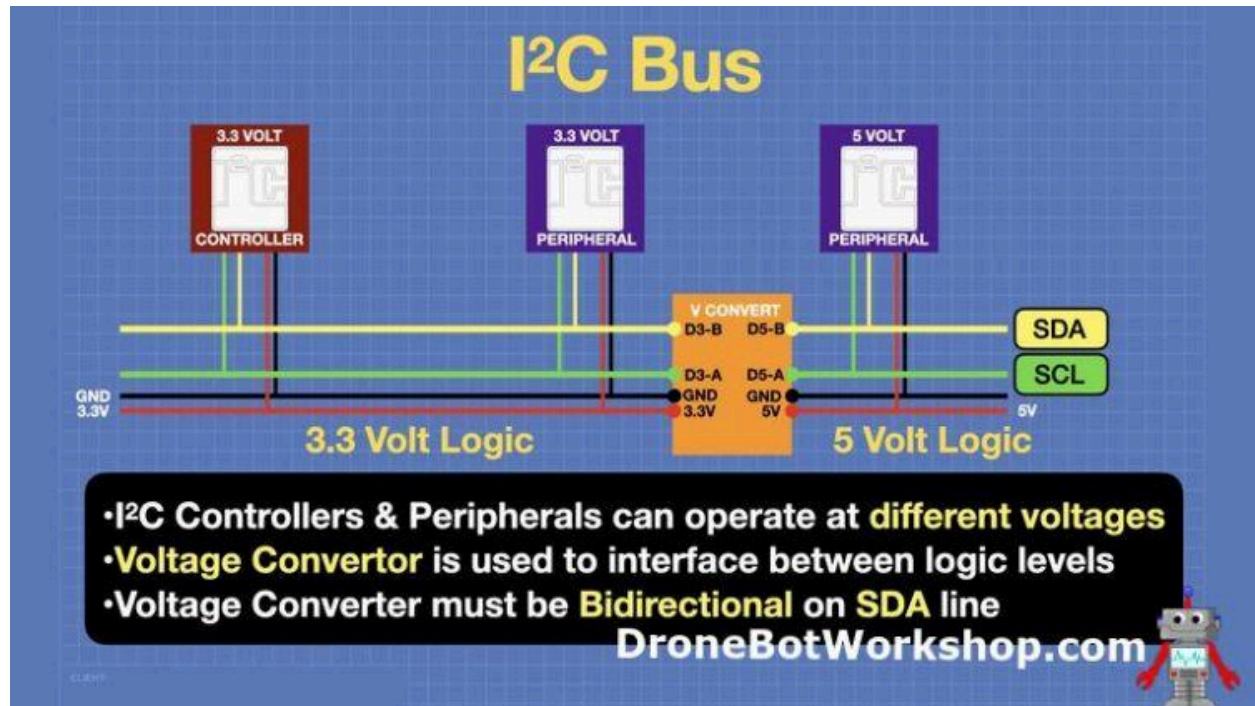


The Controller includes the intended peripheral's address in the first byte of every I²C transaction. No two peripherals on the same bus should share an address, or bus conflicts will occur. In theory, up to 127 peripherals can exist on the same bus; however, in real life, the number of devices usually doesn't exceed a dozen.

Data is sent in 8-bit packets, and every byte transferred is followed by an acknowledgment bit from the receiving device. Only the Controller can initiate a conversation, preventing conflicts between peripherals. The Controller is also the source of the clock signal on the SCL line, and all bus traffic is synchronized to this clock.



The bus uses open-drain outputs with pull-up resistors on both SDA and SCL lines, typically 4.7k to 10k. Many I2C peripherals have built-in pull-up resistors, and care should be taken to keep the total resistance of all pull-ups (wired in parallel) to over 2.2k. Most designs aim for a 4.7k pull-up resistor.



The I²C bus can operate at different logic voltage levels, and is commonly implemented with either a 3.3-volt or 5-volt power supply. If you need to mix peripherals and controllers that use different logic voltages, you'll need to use a Voltage Converter module. It must be bidirectional, as data on the SDA line flows in both directions.

ESP32 I²C Basics

The ESP32 family includes robust I²C support with hardware controllers that handle the low-level protocol details. Most ESP32 variants feature **two built-in hardware I²C controllers** (sometimes called “I²C ports”):

- **I²C Controller 0 (I2C0 (I2C_EXT0))**
- **I²C Controller 1 (I2C1 (I2C_EXT1))**

These controllers are part of the chip’s peripheral system and can operate simultaneously, allowing for flexible multi-bus configurations.

ESP32 I²C Implementation

The hardware controllers can be configured independently to operate in either **Controller mode** (issuing the clock and initiating transactions) or **Peripheral mode** (responding to requests from another I²C controller on the bus). This dual capability allows you to use an ESP32 both as a system “host” talking to sensors, and as a smart peripheral that another microcontroller can control. We will be configuring an ESP32 as a Peripheral later in this article. Note that you will need to program using the ESP-IDF to use the second controller in Peripheral mode.

Unlike many microcontrollers with fixed I²C pins, the ESP32 lets you map SDA and SCL to almost any available GPIO. In some iterations (such as the ESP32-C6 that we will be using today), the second I²C bus is mapped to fixed pins.

ESP32-C6 DevKit

The **ESP32-C6-DevKitC-1** is Espressif's official development board for the ESP32-C6, a powerful Wi-Fi 6 and Bluetooth 5 (LE) enabled RISC-V microcontroller.

This board is designed for prototyping and testing applications that take advantage of the ESP32-C6's next-generation connectivity features. It includes a USB-to-UART bridge for programming, a 5V-to-3.3V regulator, a reset button, and a boot button, making it ready for use with the Arduino IDE, ESP-IDF, or other frameworks.



The DevKit has the following specifications:

- **Module:** ESP32-C6-WROOM-1 or WROOM-1U (external antenna)
- **Flash Memory:** 8 MB SPI flash
- **Wireless Protocols:**
 - Wi-Fi 6 (802.11ax, 2.4 GHz)
 - Bluetooth 5 (LE)
 - IEEE 802.15.4 (Zigbee 3.0, Thread 1.3)

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

- **Processor:** Single-core RISC-V CPU
- **USB Ports:**
 - USB-to-UART bridge (up to 3 Mbps)
 - Native USB 2.0 (12 Mbps, full-speed)
- **Power Supply Options:**
 - USB Type-C ports
 - 5V and 3.3V pin headers
- **GPIO Access:** Most GPIOs broken out to pin headers
- **RGB LED:** Addressable, connected to GPIO8
- **Buttons:**
 - Boot (for flashing)
 - Reset
- **Current Measurement:** J5 jumper for inline ammeter connection

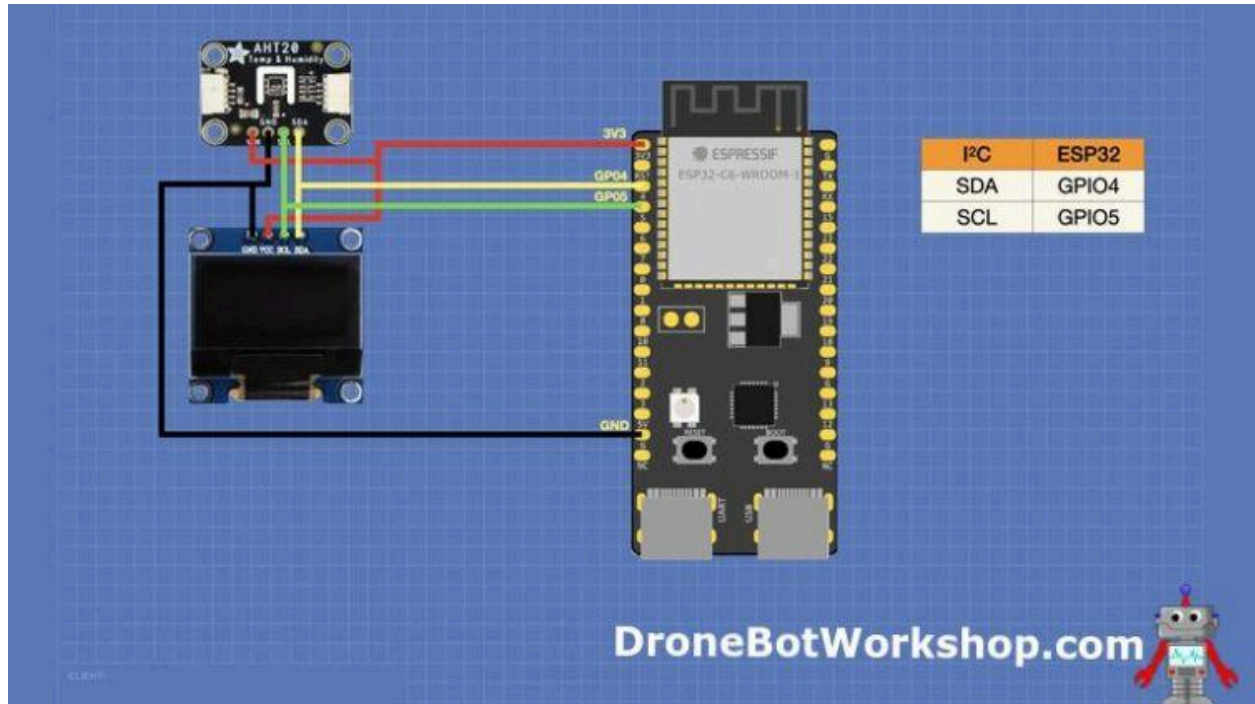
With support for protocols like Matter and Zigbee, as well as the standard WiFi and Bluetooth, this inexpensive module is one that I'll be using for many projects going forward.

ESP32 I²C Demo Hookup

We will be doing a few experiments using the ESP32-C6 DevKit and the following I²C peripherals:

- Adafruit AHT20 Temperature and Humidity Sensor
- SSD1306 OLED Display (we will be using two of these eventually)

Here is how we will be hooking everything up:

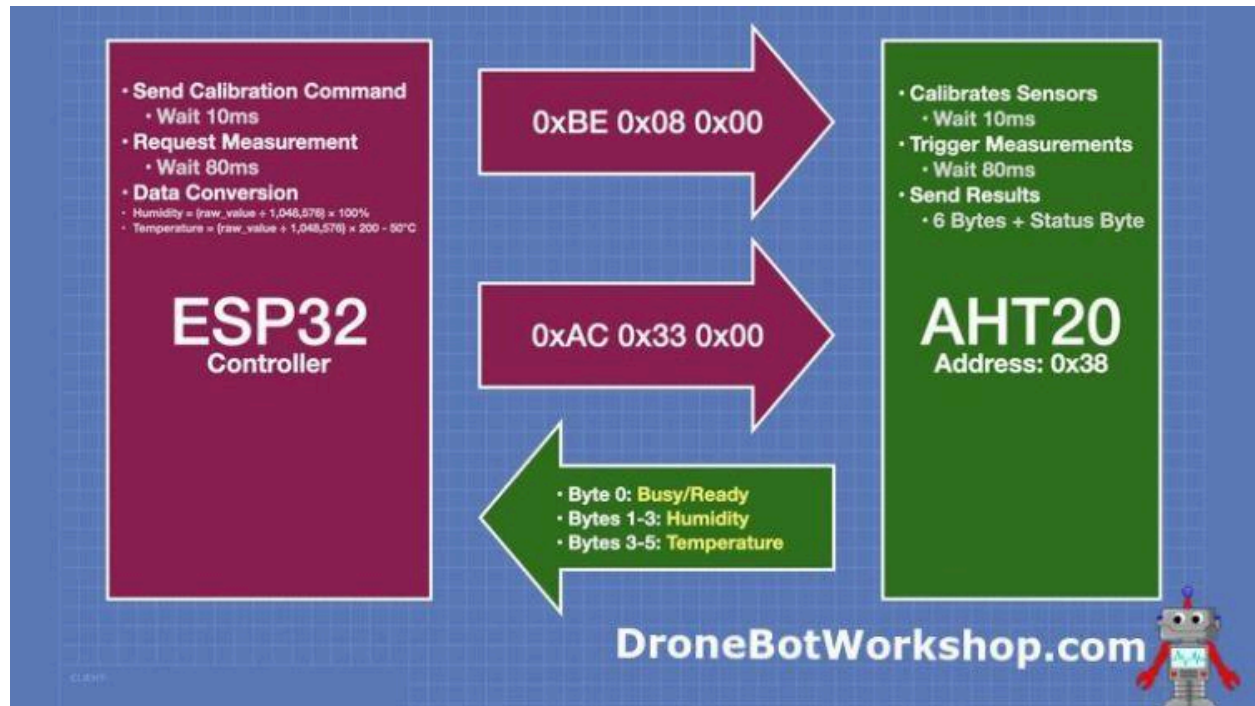


If you don't have the same ESP32 module as the one I am using, you can substitute another one, as just about any ESP32 will work. If GPIO pins 4 and 5 are not available, you can use different ones and change the pin designations in the sketches to match.

ESP32 I²C Demo – Basic Wire Code for AHT20

We will begin by creating a sketch to display temperature and humidity readings from the AHT20 on the serial monitor. We will be ignoring the OLED for now.

Instead of using a library for the AHT20, we are just going to use the Wire library. This will show you how to communicate with an I²C peripheral without a dedicated library. The code implements the complete AHT20 communication protocol from scratch, providing excellent insight into low-level I²C operations.



The program follows the AHT20's required initialization sequence: performing a soft reset, sending calibration commands, and then entering a continuous measurement loop. Each measurement cycle involves triggering a reading, waiting for the sensor to complete its internal conversion process, and then retrieving six bytes of raw data.

Here is the sketch:

```
/*  
Wire Library Demo  
esp32-aht20-wire.ino  
Demonstrates use of Wire Library for I2C  
Read Temperature & Humidity from AHT20  
AHT20 I2C address: 0x38  
Sequence:  
1. Soft reset  
2. Trigger measurement  
3. Wait for busy flag to clear  
4. Read 6 bytes and convert to values  
  
DroneBot Workshop 2025  
https://dronebotworkshop.com  
*/  
  
// Include Wire Library for I2C  
#include <Wire.h>  
  
// Define Pins  
const uint8_t SDA_PIN = 4;  
const uint8_t SCL_PIN = 5;  
const uint8_t AHT20_ADDR = 0x38;  
  
void setup() {  
  
    // Start Serial Monitor  
    Serial.begin(115200);  
}
```



```
    delay(200);

    // Start I2C with custom SDA/SCL pins
    Wire.begin(SDA_PIN, SCL_PIN);

    Serial.println("Initializing AHT20...");

    // 1. Soft reset
    Wire.beginTransmission(AHT20_ADDR);
    Wire.write(0xBA); // Soft reset command
    Wire.endTransmission();
    delay(20); // Small delay after reset

    // 2. Initialization / calibration command
    Wire.beginTransmission(AHT20_ADDR);
    Wire.write(0xBE); // Init command
    Wire.write(0x08);
    Wire.write(0x00);
    Wire.endTransmission();
    delay(10);
}

void loop() {
    // 3. Trigger measurement (temperature + humidity)
    Wire.beginTransmission(AHT20_ADDR);
    Wire.write(0xAC); // Trigger measurement command
    Wire.write(0x33);
    Wire.write(0x00);
```

```
Wire.endTransmission();

delay(80); // Wait typical measurement time (~80ms)

// 4. Read 6 bytes of data
Wire.requestFrom(AHT20_ADDR, (uint8_t)6);
if (Wire.available() == 6) {
    uint8_t data[6];

    for (int i = 0; i < 6; i++) {
        data[i] = Wire.read();
    }

    // Byte0 bit7 = busy flag (should be 0 now)
    if (data[0] & 0x80) {
        Serial.println("Sensor busy, try again");
        delay(500);
        return;
    }

    // Combine humidity bytes (20-bit value)
    uint32_t rawHumidity = ((uint32_t)data[1] << 12) | ((uint32_t)data[2] << 4) |
        ((uint32_t)data[3] >> 4);

    // Combine temperature bytes (20-bit value)
    uint32_t rawTemp = (((uint32_t)data[3] & 0x0F) << 16) | ((uint32_t)data[4] << 8) |
        ((uint32_t)data[5]);

    // Convert to human-readable
    float humidity = (rawHumidity * 100.0) / 1048576.0; // 2^20 = 1048576
```

```
float temperature = (rawTemp * 200.0 / 1048576.0) - 50.0;

// Output results
Serial.print("Humidity: ");
Serial.print(humidity, 1);
Serial.print(" % | Temp: ");
Serial.print(temperature, 1);
Serial.println(" C");
} else {
    Serial.println("Read error");
}

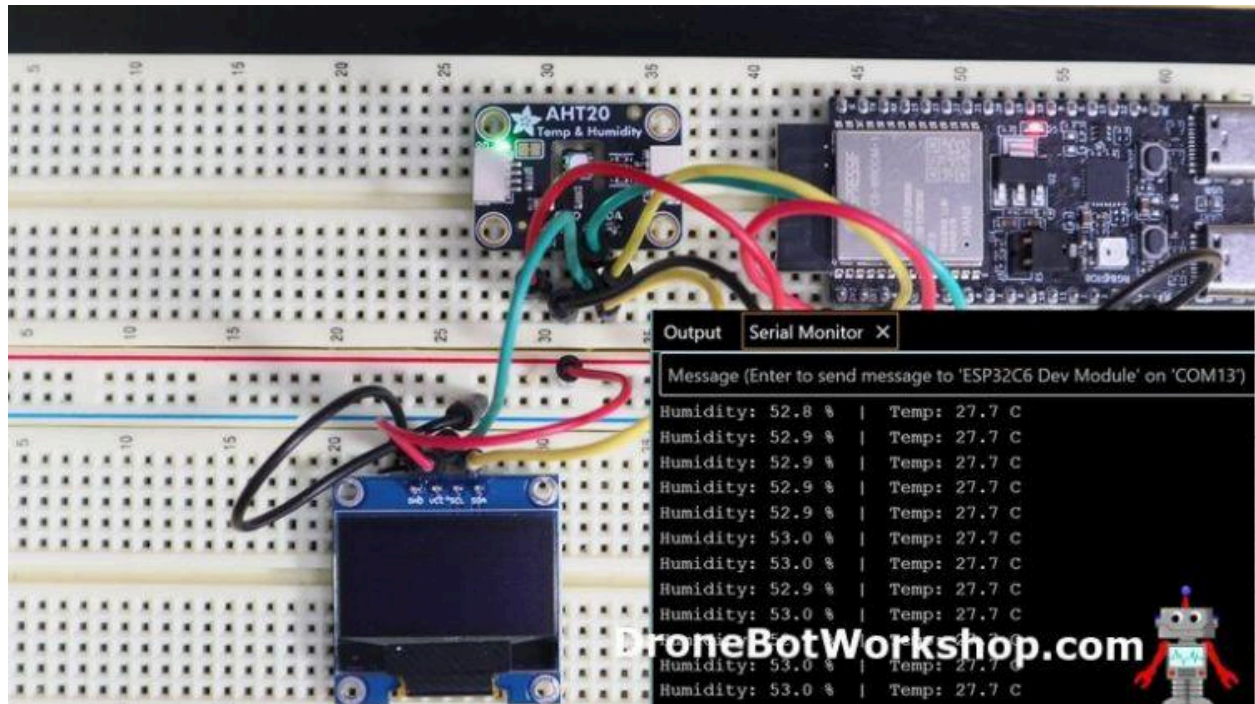
delay(2000); // Wait before next reading
}
```

In the **setup() function**, the sketch initializes serial communication for output and starts the I²C bus using **custom SDA and SCL pins (GPIO 4 and 5)**. It then sends a **soft reset command (0xBA)** to the AHT20, followed by an **initialization/calibration command (0xBE with parameters 0x08, 0x00)**. These steps ensure the sensor starts in a clean state and is ready for measurements.

Inside the **loop() function**, the ESP32 sends the **trigger measurement command (0xAC, 0x33, 0x00)** to the sensor, which starts a combined temperature and humidity measurement. After waiting about 80 ms for the measurement to complete, the ESP32 requests 6 data bytes from the sensor. The sketch checks the **busy flag** in the first byte to confirm the sensor is ready, then extracts the raw 20-bit humidity and temperature values by combining bits from multiple bytes.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

These raw values are converted into **percentage humidity** and **degrees Celsius** using formulas from the AHT20 datasheet. Finally, the results are printed to the serial monitor every two seconds.



Load the sketch and observe the output on the serial monitor. If you don't get any output, double-check your wiring.

<https://dronebotworkshop.com>

ESP32 I²C Demo – AHT20 with OLED Display

Now we will add the OLED display to the picture. We'll simplify our coding by using three libraries:

- Adafruit_AHTX0 – Library for the AHT20 temperature and humidity sensor.
- Adafruit_GFX – Graphics library, used with the display.
- Adafruit_SSD1306 – Library for the SSD1306 OLED display.

All of these libraries can be installed using the Library Manager in the Arduino IDE.

Here is the sketch we will be using:


```
/*  
  ESP32 I2C Demo  
  
  esp32-i2c-th-oled.ino  
  
  Uses ESP32, OLED Display and Temperature/Humidity Sensor  
  
  Demonstrates operation of I2C bus on ESP32  
  
  
  DroneBot Workshop 2025  
  
  https://dronebotworkshop.com  
*/  
  
  
#include <Wire.h>  
#include <Adafruit_AHTX0.h>  
#include <Adafruit_GFX.h>  
#include <Adafruit_SSD1306.h>  
  
  
// I2C pins for ESP32-C6  
const int SDA_PIN = 4;  
const int SCL_PIN = 5;  
  
  
// OLED dimensions  
#define SCREEN_WIDTH 128  
#define SCREEN_HEIGHT 64  
  
  
// Objects  
Adafruit_AHTX0 aht;  
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);  
  
  
void setup() {
```

```
Serial.begin(115200);

delay(200);

Wire.begin(SDA_PIN, SCL_PIN);

// Initialize OLED
if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println("SSD1306 OLED not found");
    for (;;)
}

display.clearDisplay();

// Initialize AHT20
if (!aht.begin()) {
    Serial.println("AHT20 not found");
    display.setTextSize(1);
    display.setTextColor(SSD1306_WHITE);
    display.setCursor(0, 0);
    display.println("AHT20 not found");
    display.display();
    for (;;)
}

// Draw static title in yellow section
display.setTextSize(1);
display.setTextColor(SSD1306_WHITE);
display.setCursor(10, 0); // Top yellow band
display.println("Temp/Humid Demo");
```

```
display.display();

delay(1000);
}

void loop() {

  sensors_event_t humidity, temp;
  aht.getEvent(&humidity, &temp);

  // Debug to Serial

  Serial.print("Temp: ");
  Serial.print(temp.temperature, 1);
  Serial.print(" C Humidity: ");
  Serial.print(humidity.relative_humidity, 1);
  Serial.println(" %");

  // Clear only the blue section (y = 16 to 63)
  display.fillRect(0, 16, SCREEN_WIDTH, SCREEN_HEIGHT - 16, SSD1306_BLACK);

  // Large temp in blue area
  display.setTextSize(2);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 20); // Starting in blue section
  display.print(temp.temperature, 1);
  display.cp437(true);
  display.print(" C");

  // Large humidity in blue area
  display.setCursor(0, 45);
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

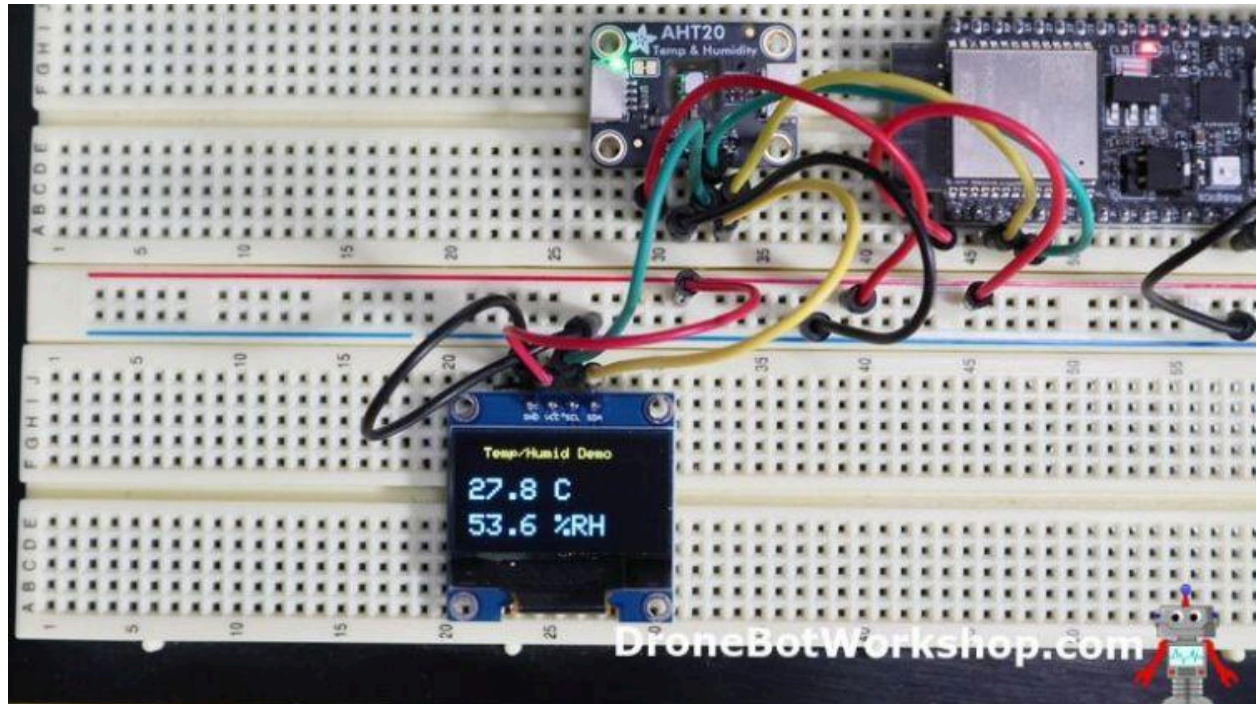
```
display.print(humidity.relative_humidity, 1);  
  
display.print(" %RH");  
  
display.display();  
  
delay(2000);  
}
```

In **setup()**, the code starts the serial monitor, then brings up the I²C bus on custom pins SDA=GPIO4 and SCL=GPIO5 with `Wire.begin(SDA_PIN, SCL_PIN)`. It then initializes each Peripheral in turn. If the OLED or the sensor isn't detected at its address, the sketch prints a message and stops.

In the **loop()** function, the ESP32, acting as the Controller, requests the latest temperature and humidity data from the AHT20 Peripheral using the `aht.getEvent()` function. Once the data is received, the code sends it to the Serial Monitor for debugging and then formats it to be displayed on the OLED screen. It clears the previous readings from the display and sends the new values, which the OLED Peripheral then shows on its screen.

This cycle repeats every two seconds.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>



Load the code and observe the OLED display. You should see the temperature and humidity displayed, along with a header. On most OLEDs, the header will be a different color than the main text area; my header is yellow with blue text.

<https://dronebotworkshop.com>

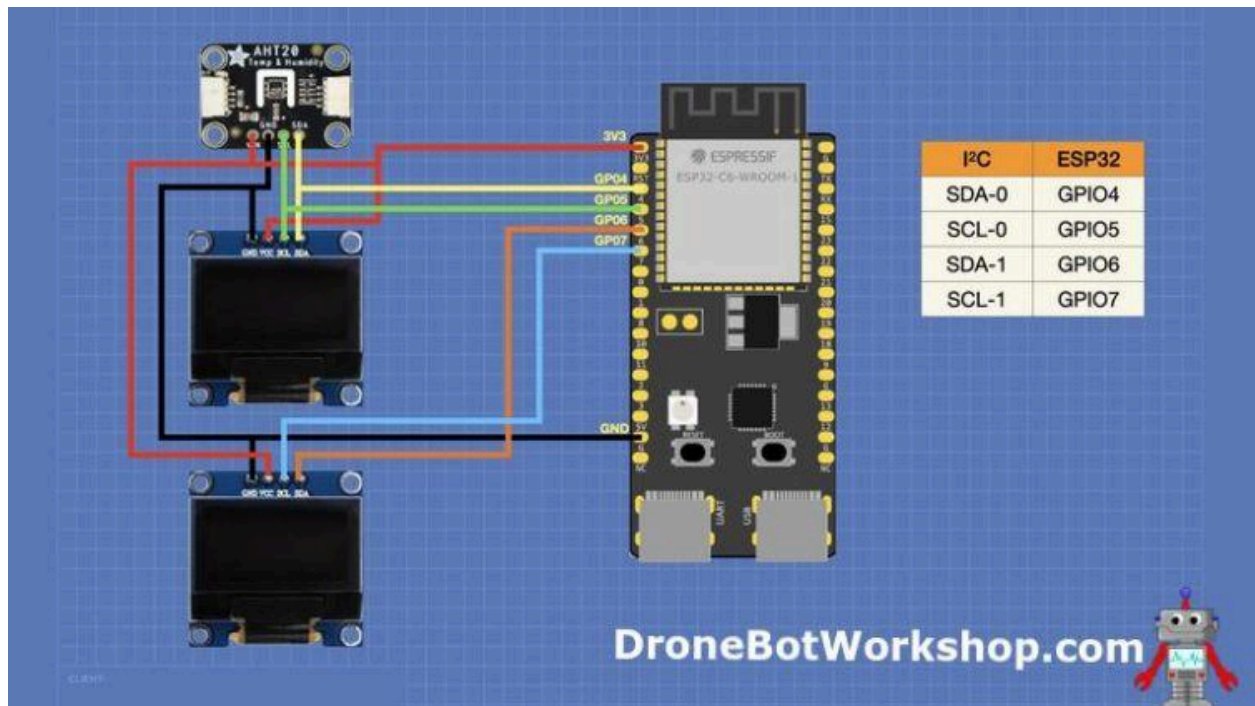
Multiple I²C Buses

The ESP32 has two hardware I²C controllers, each one capable of operating as either a Controller or a Peripheral. Note that limitations in ESP32 Boards Manager 3 prevent the second bus from being used as a Peripheral; you would need the ESP-IDF development platform to do this.

One great use of the second I²C bus is to resolve a problem when two peripherals have the same address, and neither one can be changed. We will simulate this situation with two OLED displays, each with the address of 0x3C. While this situation would normally cause a conflict, we can put one display on the second I²C bus to avoid this.

ESP32 I²C Multiple Bus Demo Hookup

Here is our hookup. Note that this is the same hookup we've already made; we are just adding a second OLED.



Note that GPIO6 and GPIO7 are the default pins for the second I²C controller. If you are using a different ESP32, you will need to determine the second controller's default pins, as the Arduino IDE won't allow you to change them.

ESP32 I²C Multiple Bus Demo Code

Here is the sketch that we will be using to utilize both I²C buses, each one driving an identical OLED display.

```
/*  
  ESP32 I2C Dual I2C Bus Demo  
  
  esp32-dual-i2c.ino  
  
  Demonstrates using both I2C buses in ESP32  
  
  Uses ESP32, 2 OLED displays and temperature/humidity sensor  
  
  
  DroneBot Workshop 2025  
  
  https://dronebotworkshop.com  
*/  
  
  
#include <Wire.h>  
#include <Adafruit_GFX.h>  
#include <Adafruit_SSD1306.h>  
#include <Adafruit_AHTX0.h>  
  
  
// OLED Display configuration  
  
#define SCREEN_WIDTH 128  
#define SCREEN_HEIGHT 64  
#define OLED_RESET -1  
#define SCREEN_ADDRESS 0x3C // BOTH displays have the SAME address!  
  
  
// I2C Bus 0 pins (for temperature display and sensor) - Main I2C  
  
#define I2C0_SDA 4  
#define I2C0_SCL 5  
  
  
// I2C Bus 1 pins (for humidity display) - LP I2C (FIXED pins only!)  
  
#define I2C1_SDA 6  
#define I2C1_SCL 7
```

```
// Create TwoWire objects for each bus

TwoWire I2C_Bus0 = TwoWire(0);

TwoWire I2C_Bus1 = TwoWire(1);


// Create peripheral objects - NOTE: Both OLEDs use same address!

Adafruit_SSD1306 tempDisplay(SCREEN_WIDTH, SCREEN_HEIGHT, &I2C_Bus0, OLED_RESET);
Adafruit_SSD1306 humidityDisplay(SCREEN_WIDTH, SCREEN_HEIGHT, &I2C_Bus1, OLED_RESET);
Adafruit_AHTX0 aht;


void setup() {

  Serial.begin(115200);


  // Initialize I2C buses with specific pins

  I2C_Bus0.begin(I2C0_SDA, I2C0_SCL, 100000);
  I2C_Bus1.begin(I2C1_SDA, I2C1_SCL, 100000);


  // Initialize temperature display on Bus 0 (same address as humidity display!)
  if (!tempDisplay.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println(F("Temperature display allocation failed"));
    for (;;)
      ;
  }


  // Initialize humidity display on Bus 1 (same address - but different bus!)
  if (!humidityDisplay.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println(F("Humidity display allocation failed"));
    for (;;)
      ;
  }
}
```

```
    ;  
}  
  
// Initialize AHT20 sensor on Bus 0 (with temperature display)  
if (!aht.begin(&I2C_Bus0)) {  
    Serial.println("Could not find AHT20 sensor on Bus 0!");  
    while (1) delay(10);  
}  
  
// Initialize displays  
setupDisplay(tempDisplay, "Temperature", "Bus 0");  
setupDisplay(humidityDisplay, "Humidity", "Bus 1");  
  
Serial.println("Multiple I2C Bus Demo initialized");  
Serial.println("Both OLED displays have address 0x3C - but on separate buses!");  
  
// Scan both buses for verification  
scanAllBuses();  
}  
  
void setupDisplay(Adafruit_SSD1306& display, const char* title, const char* bus) {  
    display.clearDisplay();  
    display.setTextSize(1);  
    display.setTextColor(SSD1306_WHITE);  
    display.setCursor(0, 0);  
    display.println(title);  
    display.println(bus);  
    display.println("Initializing...");  
}
```



```
    display.display();

    delay(1000);
}

void loop() {

    // Read sensor data

    sensors_event_t humidity, temp;
    aht.getEvent(&humidity, &temp);

    // Update temperature display (Bus 0)

    updateTemperatureDisplay(temp.temperature);

    // Update humidity display (Bus 1)

    updateHumidityDisplay(humidity.relative_humidity);

    // Print to serial monitor

    Serial.print("Temp: ");
    Serial.print(temp.temperature, 1);
    Serial.print("°C, Humidity: ");
    Serial.print(humidity.relative_humidity, 1);
    Serial.println("%");

    delay(2000);
}

void updateTemperatureDisplay(float temperature) {

    tempDisplay.clearDisplay();

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

```
tempDisplay.setTextSize(1);  
tempDisplay.println("TEMPERATURE");  
tempDisplay.println("I2C Bus 0");  
tempDisplay.print("Address: 0x");  
tempDisplay.println(SCREEN_ADDRESS, HEX);  
tempDisplay.println();  
  
tempDisplay.setTextSize(2);  
tempDisplay.print(temperature, 1);  
tempDisplay.println(" C");  
  
tempDisplay.display();  
}  
  
void updateHumidityDisplay(float humidity) {  
    humidityDisplay.clearDisplay();  
    humidityDisplay.setCursor(0, 0);  
    humidityDisplay.setTextSize(1);  
    humidityDisplay.println("HUMIDITY");  
    humidityDisplay.println("I2C Bus 1");  
    humidityDisplay.print("Address: 0x");  
    humidityDisplay.println(SCREEN_ADDRESS, HEX);  
    humidityDisplay.println();  
  
    humidityDisplay.setTextSize(2);  
    humidityDisplay.print(humidity, 1);  
    humidityDisplay.println(" %");  
}
```

```
    humidityDisplay.display();
}

// Function to scan devices on both buses
void scanAllBuses() {
    Serial.println("Scanning I2C Bus 0:");
    scanI2CBus(I2C_Bus0);

    Serial.println("Scanning I2C Bus 1:");
    scanI2CBus(I2C_Bus1);
}

void scanI2CBus(TwoWire& wire) {
    byte error, address;
    int 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.print(address, HEX);
            Serial.println(" !");

            nDevices++;
        }
    }
}
```

```
if (nDevices == 0) {  
    Serial.println("No I2C devices found");  
}  
  
Serial.println();  
}
```

The sketch configures the ESP32 to act as a Controller on two separate I²C buses simultaneously. Instead of using the default Wire object, the code creates two distinct TwoWire objects: I2C_Bus0 and I2C_Bus1. Each object is assigned to one of the ESP32's physical hardware controllers (0 and 1) and is connected to its own unique set of SDA/SCL pins.

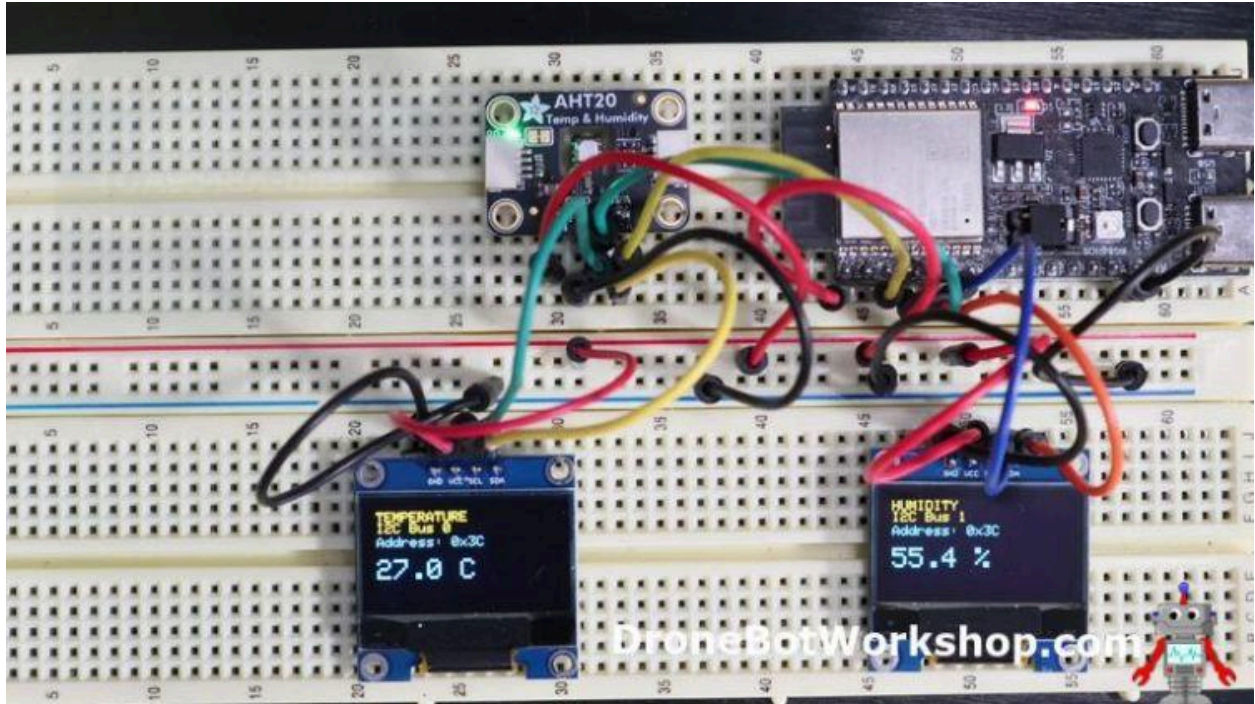
The code then initializes three Peripheral devices:

- An AHT20 sensor and the first OLED display (tempDisplay) are initialized on Bus 0.
- The second OLED display (humidityDisplay) is initialized on Bus 1.

In the **loop()**, the Controller first requests data from the AHT20 sensor on Bus 0. It then sends the temperature value to the OLED on Bus 0 and the humidity value to the OLED on Bus 1, updating each display with its specific information.

The setup() function also includes a handy I²C scanner that checks both buses to verify which peripherals are connected to each one.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>



Load the code to the ESP32 and observe the results. You should see temperature displayed on one OLED and humidity on the other. The I²C address of each OLED (0x3C) is also shown.

<https://dronebotworkshop.com>

ESP32 as an I²C Peripheral

This demonstration code transforms the ESP32-S3 into a custom I²C peripheral device, creating what is essentially a “smart potentiometer sensor” that can be queried by any I²C controller. The code showcases the ESP32’s ability to operate as a peripheral rather than the typical controller role we’ve seen in previous examples.

The implementation uses a [Seeeduino XIAO ESP32-S3](#) with a potentiometer connected to analog pin A0. The ESP32 continuously reads the potentiometer value, applies sophisticated averaging to reduce noise, and maps the 12-bit ADC reading to an 8-bit value (0-255) for efficient I²C transmission. When an I²C controller requests data from address 0x2A, the ESP32 responds with the current potentiometer value as a single byte.

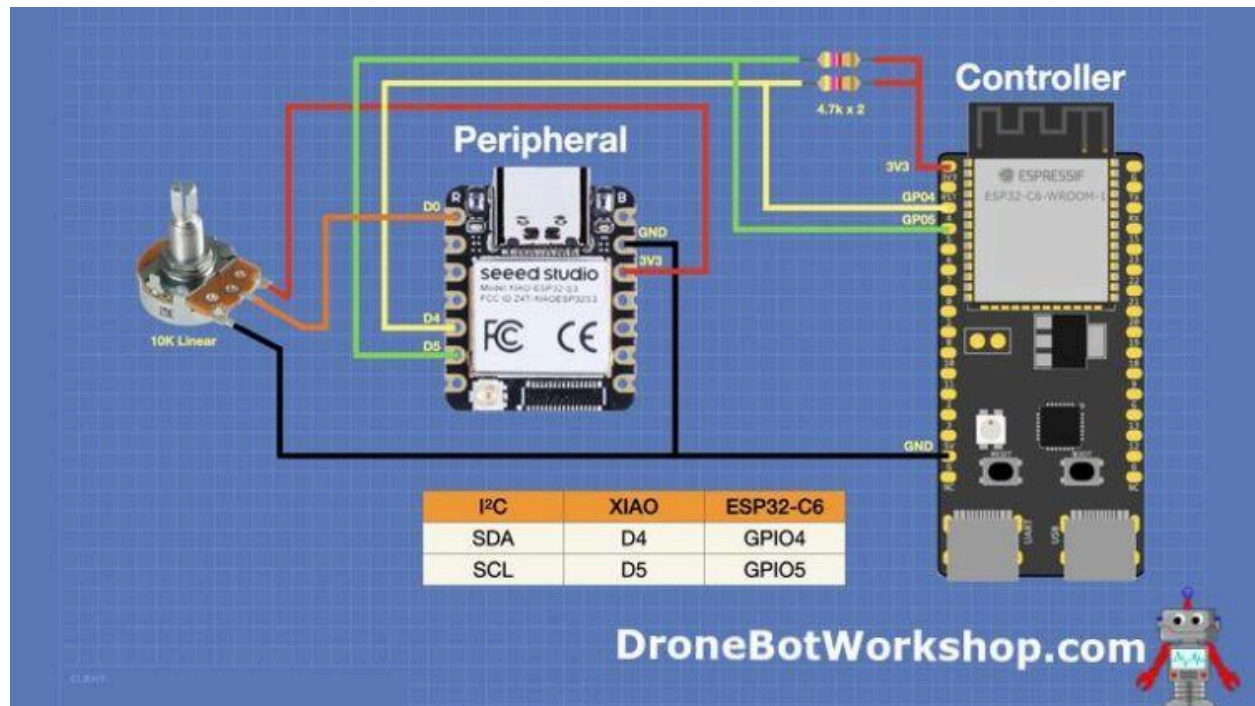
We are using the XIAO as a peripheral for two reasons:

- It is small, which is good for a DIY peripheral.
- The ESP32-C6 cannot be programmed as a peripheral using the Arduino IDE (it can with the ESP-IDF).

One restriction of the Wire Library is that a Peripheral must use the default I²C pins.

ESP32 I²C Peripheral Demo Hookup

Here is the hookup for our demo. Note that we are still using an ESP32-C[^] DevKit as the Controller. Actually, any ESP32 would probably work as a controller.



Also note the use of the pull-up resistors, which were not required in previous circuits, as the peripherals (AHT20 & OLED) both have built-in ones.

ESP32 I²C Peripheral Demo Code – Peripheral

Here is the sketch that we will load onto the Seeeduino XIAO:


```
/*  
  ESP32S3 I2C Perepheral Demo  
  esp32S3-perepheral.ino  
  Demonstrates use of ESP32 as I2C perepheral  
  Uses Seeeduino XIAO ESP32S3  
  Reads value of potentiometer (0-255) on A0  
  
  DroneBot Workshop 2025  
  https://dronebotworkshop.com  
*/  
  
// Include Wire library for I2C  
#include <Wire.h>  
  
constexpr uint8_t I2C_ADDR = 0x2A; // Peripheral address  
constexpr int POT_PIN = 1;        // A0 on XIAO  
  
volatile uint8_t potByte = 0;  
  
// Read potentiometer and map to 0-255  
uint8_t readPotByte() {  
  analogReadResolution(12); // 0..4095  
  uint32_t acc = 0;  
  for (int i = 0; i < 8; i++) {  
    acc += analogRead(POT_PIN);  
    delayMicroseconds(200);  
  }  
  uint16_t avg = acc / 8;
```

```
    return map(avg, 0, 4095, 0, 255);
}

// Called when Controller requests data
void onI2CRequest() {
    uint8_t value = potByte;
    Wire.write(&value, 1);
}

// Called when Controller sends data (not used here)
void onI2CReceive(int numBytes) {
    while (Wire.available()) (void)Wire.read();
}

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

    // Start in Peripheral (target) mode at I2C_ADDR
    // This uses the XIAO ESP32-S3 DEFAULT I2C pins (SDA=GPIO5, SCL=GPIO6)
    Wire.begin(I2C_ADDR);
    Wire.onRequest(onI2CRequest);
    Wire.onReceive(onI2CReceive);

    pinMode(POT_PIN, INPUT);

    Serial.printf("XIAO S3 Peripheral ready @ 0x%02X (default SDA/SCL)\n", I2C_ADDR);
}
```

```
void loop() {  
    potByte = readPotByte();  
    delay(5); // light pacing  
}
```

The **setup()** function is where the ESP32 is configured to act as a Peripheral. The *Wire.begin(I2C_ADDR)* command initializes the I²C bus and assigns the device the specific address of 0x2A. The two most important lines are *Wire.onRequest(onI2CRequest)* and *Wire.onReceive(onI2CReceive)*. These register “callback” functions, telling the Wire library which functions to automatically run whenever the controller interacts with it.

The **loop()** function is very simple. It continuously calls the *readPotByte()* function to get the current position of the potentiometer and stores the result in the *potByte* variable. The *readPotByte* function is designed for accuracy, taking eight quick readings from the analog pin and averaging them to reduce electrical noise before mapping the result to a single byte.

The core of the peripheral functionality is the *onI2CRequest()* function. This function is automatically executed by the Wire library only when an I²C Controller sends a request for data to address 0x2A. When that happens, this function takes the last value read from the potentiometer (stored in *potByte*) and sends it out onto the I²C bus for the controller to read.

ESP32 I²C Peripheral Demo Code – Controller

The Controller is an ESP32-C6 DevKit module. Here is the code that we will be loading onto it:

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
/*  
  ESP32 I2C Controller Demo  
  esp32-controller.ino  
  Demonstrates use of ESP32 as I2C Controller  
  Reads potentiometer values from peripheral  
  
  DroneBot Workshop 2025  
  https://dronebotworkshop.com  
*/  
  
#include <Wire.h>  
  
constexpr uint8_t  I2C_ADDR  = 0x2A;  
constexpr int      SDA_PIN   = 4;  
constexpr int      SCL_PIN   = 5;  
  
void setup() {  
  Serial.begin(115200);  
  delay(200);  
  
  // Start I2C in Controller mode on pins 4/5  
  Wire.begin(SDA_PIN, SCL_PIN);  
  
  // Set a common clock; 100 kHz is fine, 400 kHz also works on short wires  
  Wire.setClock(100000);  
  
  Serial.println("I2C Controller ready. Querying 0x2A...");  
}
```

<https://dronebotworkshop.com>

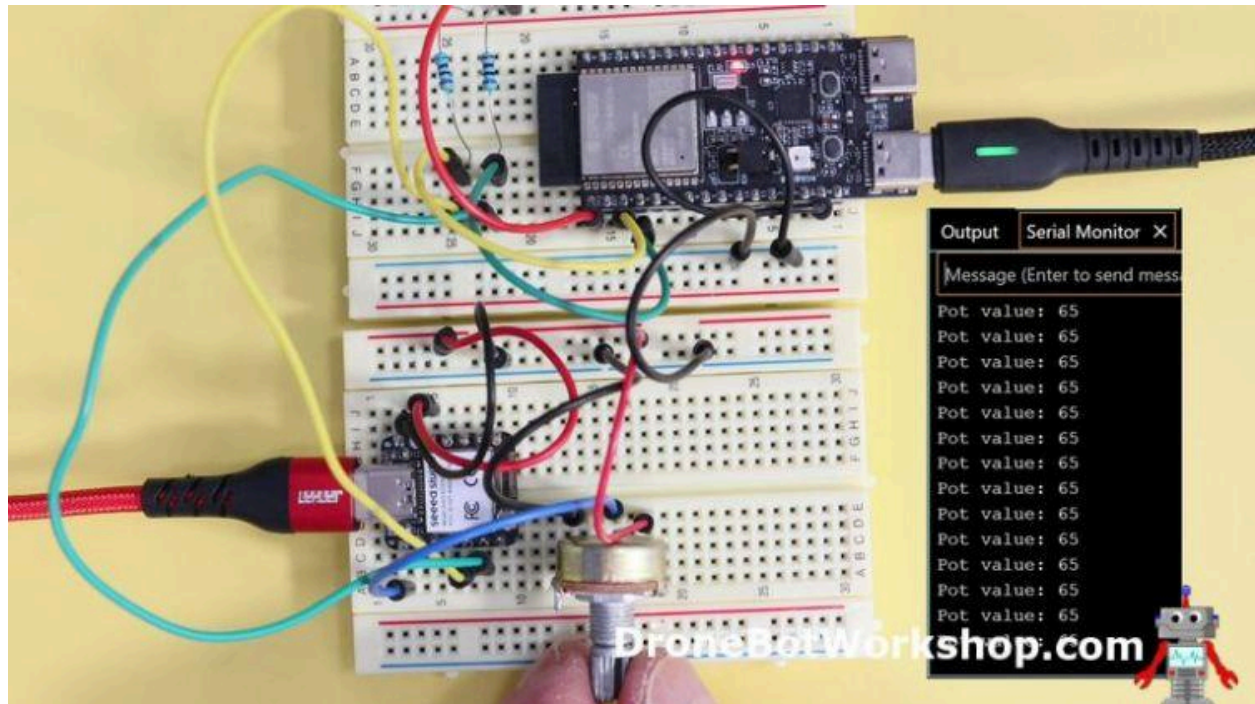
```
void loop() {  
    // Ask the peripheral for 1 byte (the pot position)  
    uint8_t toRead = 1;  
    uint8_t got = Wire.requestFrom((int)I2C_ADDR, (int)toRead);  
  
    if (got == toRead && Wire.available()) {  
        uint8_t pot = Wire.read();  
        Serial.print("Pot value: ");  
        Serial.println(pot);    // 0..255  
    } else {  
        Serial.println("Read failed (check wiring/address)");  
    }  
  
    delay(200); // Update ~5 times per second  
}
```

The **setup()** function initializes the ESP32 for its role as the bus Controller. It starts the Wire library, explicitly assigning GPIO4 and GPIO5 as the SDA and SCL pins. It also sets the I²C communication speed to a standard 100 kHz.

The **loop()** function contains the repeating action of the controller. It uses the *Wire.requestFrom()* command to ask the peripheral device at address 0x2A for exactly one byte of data. The code then checks to ensure a byte was successfully received. If it was, the controller reads the byte using *Wire.read()* and prints the value to the serial monitor. If the request fails for any reason (like incorrect wiring or the wrong peripheral address), it prints an error message. This entire process repeats approximately five times per second.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

Load up both sketches and observe the serial monitor on the Controller. You should see the value of the potentiometer, from 0 to 255, depending on its position.



While this is a simple sketch, the principle that it demonstrates can be used to construct your own advanced I²C peripherals using an ESP32.

Conclusion

I²C is an incredibly useful bus arrangement, allowing you to simplify the connection of sensors, displays, and other peripherals. Now that you are familiar with the advanced I²C functions within the ESP32, you can use I²C devices more efficiently in your projects.

It's time to get on the bus! Hope you enjoyed the article and video.

<https://dronebotworkshop.com>