EMBEDDED SYSTEM AUTOMATION

23SDEC02A

A.Y: 2024 –25

Semester: II/IV – Even Sem

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

Vaddeswaram, Guntur District

Name of the Student :

Roll number :

INDEX OF EXPERIMENTS

15.	Bluetooth Communication between Two ESP 32.			
16.	Connecting to Wi-Fi Network using ESP 32.			
17.	Wi-Fi reconnect using Esp 32.			
18.	Wi-Fi station using ESP 32.			
19.	Create Wi-Fi (Access Point) Web Server using ESP 32.			
20.	Client-Server Wi-Fi Communication Between Two ESP 32.			
21.	Observation of Hall Effect value and Temperature from ESP32.			

In L										
Experiment Title										
Date										
S. No.										

S. No.	Date	Experiment Title		In l							

Project V										
Project Title										
Date										
S. No.										

S. No.	Date	Project Title									
		Project									

Session 01: Changing the clock rate of ESP 32 Microcontroller

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of clock frequency, ESP32 board
- General idea of basic circuit

PRE-LAB:

1) What is the role of the CPU clock speed in a microcontroller like the ESP32?

2) What is the default CPU clock speed of the ESP32, and how can it be configured?

OBJECTIVE:

To design and implement a serial communication port, check the CPU clock rate & XTAL Freq & APB bus clock.

COMPONENTS REQUIRED:

- ESP32
- Breadboard
- Connecting wires
- Micro USB cable

THEORY:

ESP32 CPU Speed (Frequency):

The ESP32 is a dual-core system with two Harvard Architecture Xtensa LX6 CPUs. All embedded memory, external memory, and peripherals are located on the data bus and/or the instruction bus of these CPUs.

CPU Clock:

Both CPUs can be clocked from various sources internally or externally. The most important circuitry is the PLL (Phase-Locked Loop) which multiplies the input clock frequency (whether it's an external crystal or internal oscillator). This results in a much higher frequency signal that's derived from the main clock source (orders of magnitude in terms of frequency).

The PLL_CLK is an internal PLL clock signal with a frequency of 320 MHz or 480 MHz. Note that this PLL clock is divided /2 or /4 before it's fed to any CPU. The table down below from the datasheet shows the possible configurations for the clock.

PLL_CLK (320 MHz)	1	0	CPU_CLK = PLL_CLK / 4 CPU_CLK frequency is 80 MHz
PLL_CLK (320 MHz)	1	1	CPU_CLK = PLL_CLK / 2 CPU_CLK frequency is 160 MHz
PLL_CLK (480 MHz)	1	2	CPU_CLK = PLL_CLK / 2 CPU_CLK frequency is 240 MHz

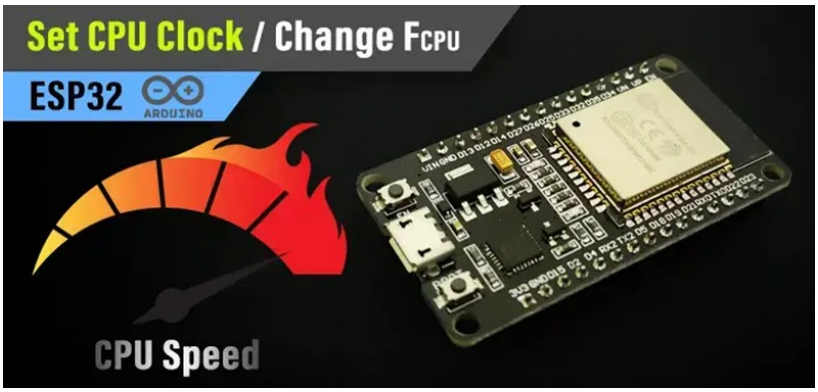
Peripherals Clock:

The APB bus clock (APB_CLK) is derived from CPU_CLK, the division factor depends on the CPU_CLK source as shown in the table down below. The peripherals clock signals are APB_CLK, REF_TICK, LEDC_SCLK, APLL_CLK, and PLL_D2_CLK.

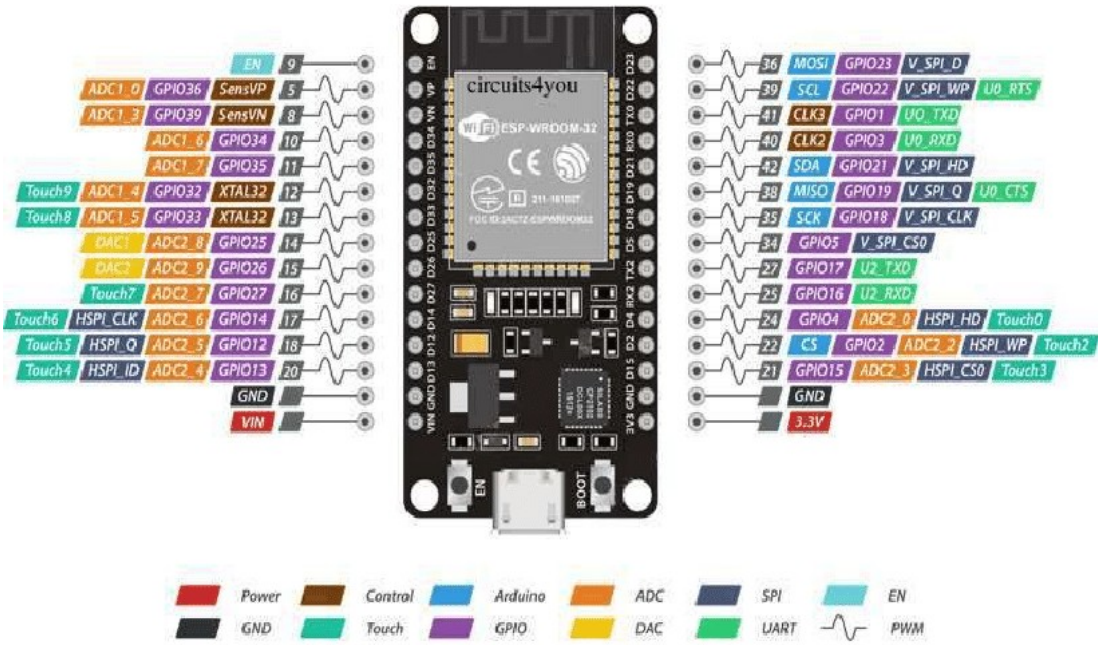
CPU_CLK Source	APB_CLK
PLL_CLK	80 MHz
APLL_CLK	CPU_CLK / 2
XTAL_CLK	CPU_CLK
RTC8M_CLK	CPU_CLK

With that being said, we can conclude that setting the PLL_CLK as a clock source will help us choose the 480MHz PLL clock output as a clock source for the CPU (the CPU_CLK will be 480/2 MHz). While the APB bus will be clocked @ 80MHz due to this option being selected.

CIRCUIT:



ESP32:



CODE:

PROCEDURE:

1. Choose the board, COM port, hold down the BOOT button, click upload and keep your finger on the BOOT button pressed.
2. When the Arduino IDE starts sending the code, you can release the button and wait for the flashing process to be completed.
3. Now, the ESP32 is flashed with the new firmware.

OUTPUT

POST LAB:

Take the snapshot of Tinker CAD simulation and paste here with your REG NO on it.

INTERFERENCE & ANALYSIS

RESULT

Session 02: Interrupt Latency measurement of ESP 32 Microcontroller

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of Interrupts, ESP32 board
- General idea of basic circuit

PRE-LAB:

1. What is the primary purpose of external interrupts in microcontroller development, such as the ESP32?

2. Explain the concept of an external interrupt trigger type. How does it influence the interrupt behaviour?

OBJECTIVE:

- To Define an output pin (for the LED)
- Define an input pin & Attach interrupt to it
- Write the ISR function to toggle the LED pin on each RISING edge

COMPONENTS REQUIRED:

- ESP32
- Breadboard
- Jumper Wires Pack
- Micro USB Cable

THEORY:

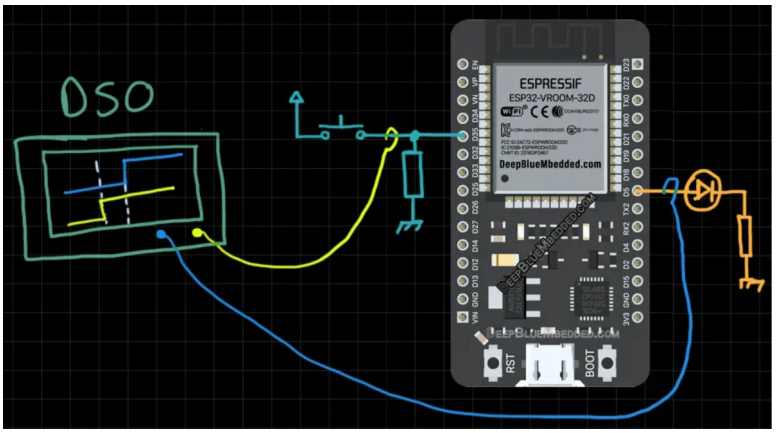
ESP32 Interrupt Pins (External Interrupts in Arduino) GPIO Interrupt:

In this tutorial, you'll learn how to use ESP32 interrupt pins in Arduino Core. We'll also discuss how to use interrupts and write your interrupt service routine (ISR) for ESP32 external interrupt GPIO pins. Then, we'll move to the Arduino Core libraries that implement drivers for the ESP32 interrupt pins and how to use its API functions, like attachInterrupt(). Without further ado, let's get right into it!

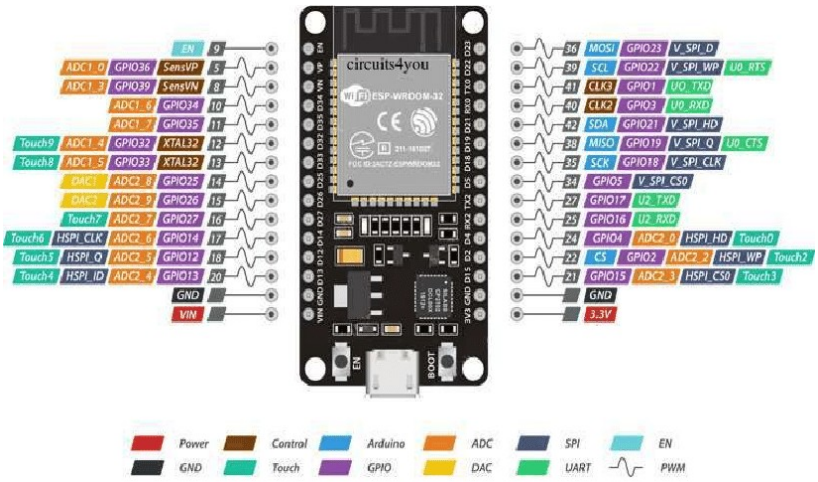
ESP32 External Interrupt Pins (IRQ):

In most microcontrollers, there are some dedicated GPIO pins that have an interrupt event generation capability. Usually referred to as IRQ pins or external interrupt pins. When the logic state of an external interrupt pin changes, it fires an interrupt signal to the CPU. So, the CPU suspends the main program execution and goes to handle a specific routine (or function) usually referred to as interrupt service routine (ISR). An ISR should always be short with minimal logic operations and calculations because it's going to happen a lot, so the main program execution will be suspended for longer periods of time, which can potentially harm the timing behaviour of your system.

CIRCUIT:



ESP32:


circuits4you
ESP-WROOM-32
EN
BOOT
Power
GND
Control
Touch
Arduino
GPIO
ADC
DAC
SPI
UART
EN
PWM

CODE:

PROCEDURE:

1. Choose the board, COM port, hold down the BOOT button, click upload and keep your finger on the BOOT button pressed.
2. When the Arduino IDE starts sending the code, you can release the button and wait for the flashing process to be completed.
3. Now, the ESP32 is flashed with the new firmware.

OUTPUT:

POST LAB:

Take the snapshot of Tinker CAD simulation and paste here with your REG NO on it.

INTERFERENCE & ANALYSIS

RESULT

Session 03: Frequency Counter using Timer and Interrupt of ESP 32 Microcontroller

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of Timer, ESP32 board
- General idea of basic circuit

PRE-LAB:

1. What is the primary purpose of timers in microcontroller development, such as the ESP32?

2. Explain the difference between hardware timers and software timers on the ESP32.

OBJECTIVE:

- Generate Timer Interrupt events in Arduino IDE.

COMPONENTS REQUIRED:

- ESP32
- breadboard
- Jumper Wires Pack
- Micro USB Cable

THEORY:

ESP32 Timers & Timer Interrupts (Arduino IDE):

In this tutorial, you'll learn how to use ESP32 internal Timers & generate Timer Interrupt events in Arduino IDE. We'll discuss how ESP32 Timers work, how to configure ESP32's Timers, and how to generate periodic interrupts to synchronize the execution of logic within your project. And also measure the timer between two events whether they're external or internal events.

ESP32 Timers:

The ESP32 SoCs come with 4 hardware timers, each of which is a general-purpose 64-bit up/down counter with a 16-bit prescaler. Except for ESP32-C3 which has only 2 timers each of which is 54 bits instead. The ESP32 timers have the capability of auto-reloading at the end of the counting period as well.

ESP32 Timers Functional Description:

Each ESP32 timer uses the APB clock (APB_CLK which is normally 80 MHz in frequency) as a base clock. This clock is then scaled downby a 16-bit prescaler which generates the time-base tick time. Therefore, we'll be changing the value of the prescaler in order to control the timer tick time.

The 16-Bit prescaler can divide the APB_CLK by a factor from 2 to 65536. When you set the prescaler value to be either 1 or 2, the clock divisor is 2; when you set the prescaler to 0, the clock divisor is 65536. Any other value will cause the clock to be divided by exactly that value that you've written to the prescaler register.

ESP32 Timers Alarm Generation:

ESP32 timers can trigger an alarm (Event) which will cause a timer to reload and/or interrupt to occur, depending on your configuration. The alarm event is triggered when the value you've stored in the alarm register matches the current timer value. This is very useful to set periodic interrupts to execute some pieces of logic periodically in your project as we'll be doing hereafter.

ESP32 Timers Equation:

In order to generate periodic events with ESP32, we'll be using the alarm event generation as well as the timer's prescaler in order to achieve the desired interrupt periodicity. There are 3 special cases for the Timer equation down below, which are when the prescaler value is = 0,1, and 2. When **Prescaler=1or2,** it's going to be as follows **[T_{OUT} = TimerTicks x (2/APB_CLK)].** When **Prescaler=0**, it's going to be as follows **[T_{OUT} = TimerTicks x (65536/APB_CLK)].** Otherwise, we can generally use the equation down below.

$$T_{OUT} = TimerTicks \times \frac{Prescaler}{APB\_CLK}$$

ESP32 Timer Example (Arduino):

Let's say we'd like to toggle an LED every 1 ms without using a **delay** that blocks the CPU and does much harm to the overall timing performance of your system. For this, we'll use the timer's equation above, Given that the default **APB_CLK** is **80MHz** or 80,000,000Hz. The desired $\mathbf{T_{OUT}}$ for the interrupt period in which we'll toggle the LED is 1ms, So $\mathbf{T_{OUT}}$ = **1ms or 0.001s**. The only two unknowns now are the Prescaler value and the TimerTicks count which we'll set the alarm event to.

We can set the Prescaler to whichever value we want but for the sake of simplifying the calculations, let's set the **Prescaler=80**. This will leave us with only one unknown which is the **TimerTicks** count. So, we'll solve the equation for TimerTicks:

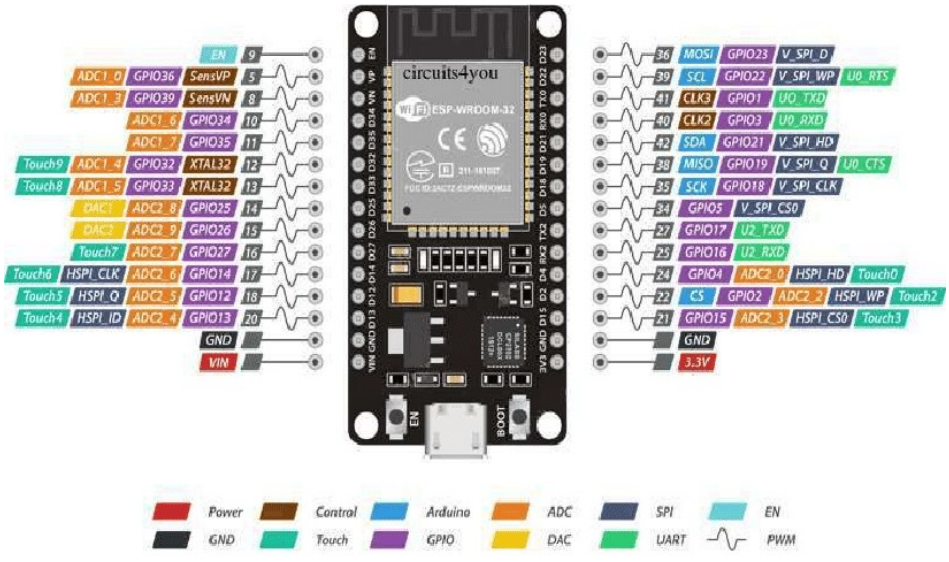
$$1ms = TimerTicks \times \frac{80}{80{,}000{,}000}$$

CIRCUIT:

PCF8574 I2C LCD Module	ESP32 DevKit v1 Board
SCL	GPIO22
SDA	GPIO21
Vcc	Vin
GND	GND

Note: the target may halt when you start the application with an external signal already connected to the input measurement pin. You can disconnect it, reset the ESP32 board, Reconnect the input signal again, and it's going to pick up the incoming signal and print its frequency on the LCD.

ESP32:



CODE:

PROCEDURE:

1. Include the libraries
2. Define an object of LiquidCrystal_I2C, set its parameters, and initialize the LCD
3. Initialize Timer0 with minimum prescaler value (2)
4. Initialize an input pin with external interrupt enabled
5. In the external interrupt handler: get the time between every two rising edges and use it to get the frequency in Hz. [Frequency = 1/Period].
6. Print the measured frequency to the I2C LCD display
7. Keep repeating…
8. The connection between ESP32 & I2C LCD module should be as follows.

PCF8574 I2C LCD Module	ESP32 DevKit v1 Board
SCL	GPIO22
SDA	GPIO21
Vcc	Vin
GND	GND

The input signal pin is defined as **D35** as you can see in the connections down below. The "**White**" wire going to D35 is the input signal to be measured.

OUTPUTS:

POST LAB:

Take the snapshot of Tinker CAD simulation and paste here with your REG NO on it.

INTERFERENCE & ANALYSIS

RESULT

Session 04: GPS module interfacing with ESP 32 Microcontroller

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of GPS module

PRE-LAB:

1. Identify the key components of the Neo-6M GPS module.

2. What communication interface does Neo-6M typically use for data transfer?

3. Which GPIO pins on the ESP32 are commonly used for connecting to the Neo-6M module?

4. Explain the purpose of each pin used in the connection.

5. What are NMEA sentences, and why are they important in GPS communication?

OBJECTIVE:

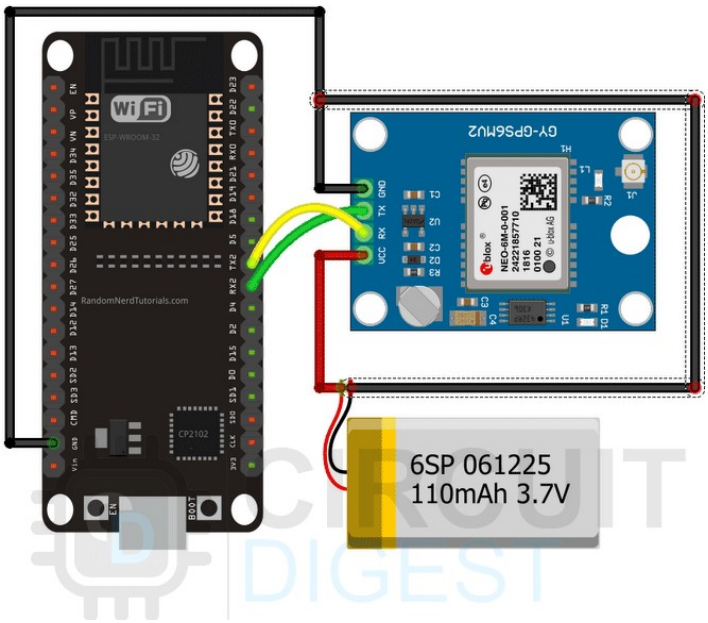
- To perform interface between ESP32 board with Neo 6M GPS module

REQUIRED COMPONENTS:

- ESP32 development board
- Neo 6M GPS module
- Connecting Wires

- Breadboard

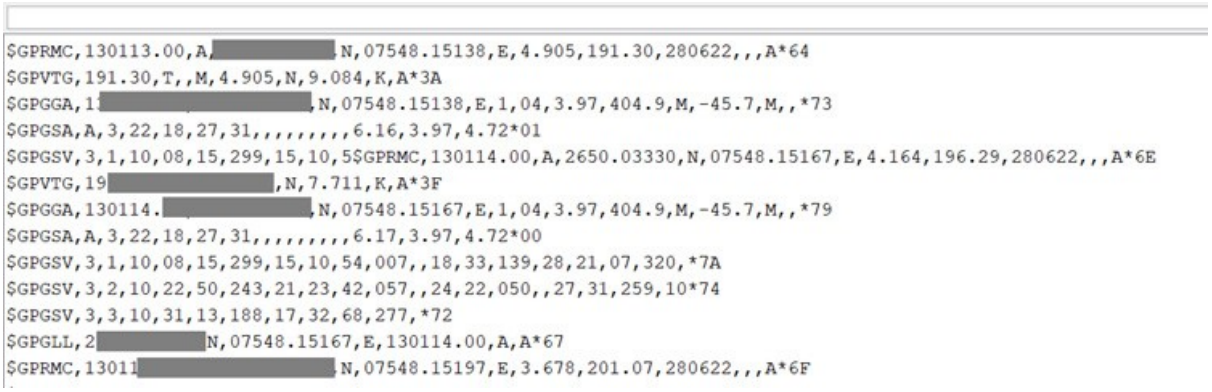
CIRCUIT:



CODE:

UNDERSTANDING NMEA SENTENCES:

If we connect the module to a USB to UART converter and open up a serial monitor window, we can see some data coming out of the NEO-6M GPS module. These are called NMEA sentences, NMEA is an acronym for National Marine Electronics Association. This is a standard message format for almost all GPS receivers. A screenshot of the serial monitor window is shown below.



NMEA sentences start with the **$ character**, and each data field is separated by a comma.

There are many different types of NMEA sentences. These different types are differentiated by the first character before the first comma. The **GP** after the **$ symbol** indicates that it's a **GPS position data**. The **$GPGGA** is a basic NEMA message that provides 3D location data.

$GPGGA, 130113.00, 37XX.XXXX,N, 07XXX.XXXX, E,1,04,3.97,404.9,M,45.7,M,,*79

130113 - represents the time when data is taken, 13:01:13 UTC

37XX.XXXX,N - Latitude 37 deg XX.XXXX' N

07XXX.XXXX,E - Longitude 007 deg 07XXX.XXXX,E

1 - fix quality (0 = invalid; 1= GPS fix. 2 = DGPS fix; 3 = PPS fix; 4 = Real Time Kinematic; 5 = Float RTK; 6 = estimated (dead reckoning); 7 = Manual input mode; 8 = Simulation mode)

04 – number of satellites being tracked

3.97 – Horizontal dilution of position

404.9, M – Altitude, in meters above the sea level

45.7, M – Height of geoid (mean sea level) above WGS84 ellipsoid

Empty - DGPS update time

Empty - DGPS station ID

***79 –** the checksum data, always begins with *

There are many different NMEA sentences. If you want to know more details about them, you can check out the GIDS website which lists all the basic information.

CODE FOR INTERFACING NEO-6M GPS WITH ESP32:

If you hook up the module with the serial monitor you will get the output on the serial monitor window that we have mentioned above. You can work with this type of data if you want to, but the easiest way is you can use a library to parse all the GPS data and store it in variables for later use. And for the code, we will do exactly that. We are going to use the TinyGPSPlus-ESP32 Library by Mike Hart. You can download the library from GitHub or you can use the Library manager of the Arduino to install the library.

Now as all the preparation is done we can move on to the code portion for ESP32, at first we start by including all the required libraries. And as the example is very basic so we are using only one library.

```
#include <TinyGPSPlus.h>
```

Next, we create a **TinyGPSPlus** object so that we can work with the library.

```
TinyGPSPlus gps;
```

Next, we have our **setup** function. In the setup function, we initialize the **Serial** and **Serial2** of the ESP32 module so we can communicate with both the PC and the GPS module.

```
void setup() {
  Serial.begin(9600);
  Serial2.begin(9600);
  delay(3000);
```

```
}
```

Next we have our **updateSerial()** function. This function is a **loopback** between the **UART1** and **UART2** so that we can monitor the incoming data out of the serial monitor window.

```
void updateSerial(){
 delay(500);
 while (Serial.available())  {
  Serial2.write(Serial.read());//Forward what Serial received to Software Serial Port
 }
 while (Serial2.available())  {
  Serial.write(Serial2.read());//Forward what Software Serial received to Serial Port
 }
}
```

Next, we have the **displayInfo()** function, in this function, we parse the GPS Latitude and Longitude with the help of **gps.location.lat()** and **gps.location.lng()** methods of the **TinyGPSPlus** library and print those in the serial monitor window.

```
void displayInfo()
{
 Serial.print(F("Location: "));
 if (gps.location.isValid()){
  Serial.print(gps.location.lat(), 6);
  Serial.print(F(","));
  Serial.print(gps.location.lng(), 6);
 }
 else
 {
  Serial.print(F("INVALID"));
 }
}
```

Finally, we have our **loop** function, in the loop function we have first called the **updateSerial()** function, and comment it out, you can uncomment this if you are running the device for the first time or you are debugging the GPS module. Next, we check if **serial2** is available or not. If serial2 is available, we call the **displayInfo()** function, this function in returns gives us the lat and long data.

```
void loop() {
 //updateSerial();
```

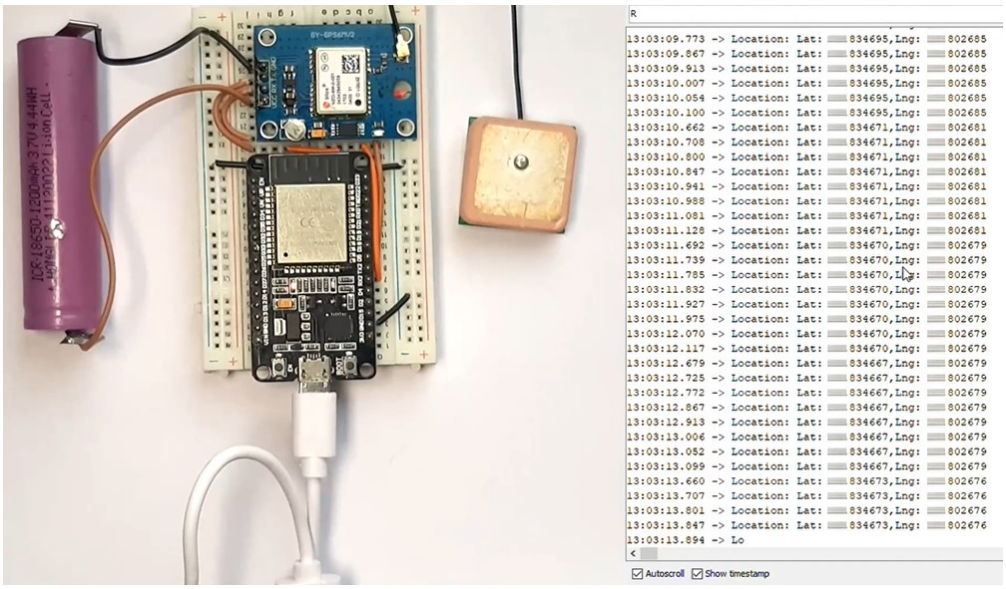
```
  while (Serial2.available() > 0)
    if (gps.encode(Serial2.read()))
      displayInfo();
  if (millis() > 5000 && gps.charsProcessed() < 10)
  {
    Serial.println(F("No GPS detected: check wiring."));
    while (true);
  }
}
```

This marked the end of our coding portion and we can move on to the next portion of the article.

WORKING OF THE NEO-6M GPS MODULE:

The gif below shows how the NEO-6M GPS module works. We have written the code so that after the ESP32 is initialized, it checks if the module is working and gives the output in the serial monitor windows. Once every checking procedure is completed, the module checks if data is available and prints it in the serial monitor window.

OUTPUT:



DEBUGGING THE NEO-6M GPS MODULE:

While working with the NEO-6M GPS module, we faced a number of issues,

- As we are using a ESP32 to communicate with the GPS module, initial thought was to power it with the 3.3V rail of the ESP but the module was not wired and I had to connect an external battery to work with the module.
- The second big issue was with the antenna. If you are thinking about using the module in indoor conditions, let me tell you it will not work because the antenna is very poor and cannot receive the satellite in indoor conditions.

OUTPUTS:

INTERFERENCE & ANALYSIS

RESULT

Session 05: I2C Device interface with ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of I2C

PRE-LAB:

1. Identify the pins on the ESP32 commonly used for I2C communication.

2. What is the purpose of the SDA and SCL pins in the I2C interface?

3. What is the purpose of I2C device scanning?

4. How does the ESP32 discover and identify I2C devices on the bus?

5. Describe the role of the Wire library in Arduino IDE for I2C communication.

OBJECTIVE:

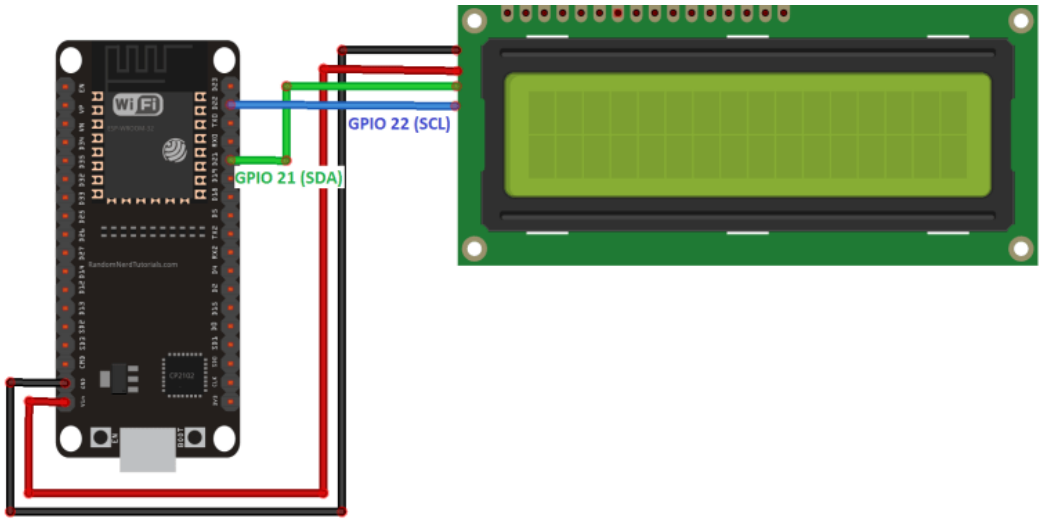
To perform I2C Device Scanning using ESP 32

REQUIRED COMPONENTS:

- ESP32 development board
- I2C LCD interface module
- Connecting Wires
- Breadboard

CONNECTIONS:

I2C LCD	ESP32
GND	GND
VCC	VIN
SDA	GPIO 21
SCL	GPIO 22



CODE:

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

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT

Session 06: Internet Radio Using I2S Amplifier and ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of I2S protocol

PRE-LAB:

1. How does an internet radio work?

2. What components are required for an ESP32-based internet radio?

3. How do you handle user interactions or control the radio functions through software?

4. What challenges might arise when working with audio streaming on an ESP32?

5. How do you manage network stability and audio quality?

OBJECTIVE:

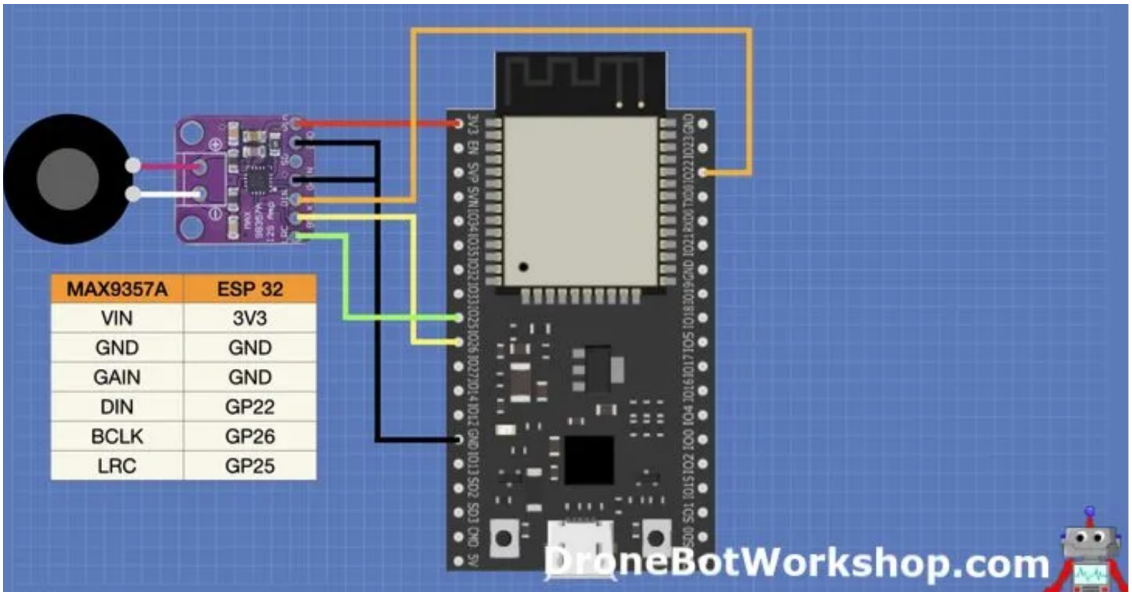
To build an internet radio using an ESP32 and an I2S amplifier module.

REQUIRED COMPONENTS:

- ESP32 development board

- I2S amplifier module
- Connecting Wires
- Breadboard

CONNECTIONS:



CODE:

Note: We will be using the same library we used for the MP3 player, as it makes it very easy to build an Internet Radio since you can just specify a URL as a sound source. Our code is quite similar to the MP3 player, except we don't have the MicroSD breakout board to deal with. After defining the I2S amplifier connections, we create an audio object, just as we did before. Then we grab the WiFi credentials, as our Internet Radio will (obviously) need to connect to the Internet! Setup starts the Serial Monitor, which we will use to display program information, and then connects to the WiFi. We then define the connections to the amplifier module and set the volume, again as we did in the last sketch. Finally, we use the libraries connecttohost function to connect to a URL, I have listed a few here and there are many more you can use. Once again, the Loop just

uses the audio object Loop method to play music. We also have a number of other functions below the Loop. These functions will display status information on the serial monitor.

OUTPUT:

Load the code onto the ESP32 and press reset. Observe the serial monitor.

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 07: Interfacing PN532 RFID board with ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of PN532 controller chip

PRE-LAB:

1. What are the suitable communication interfaces (I2C, SPI, or UART) between the PN532 and ESP32?

2. What are the specific pins on the ESP32 that will be used to communicate with the PN532 module?

3. How will the ESP32 handle data received from RFID tags via the PN532 board?

OBJECTIVE:

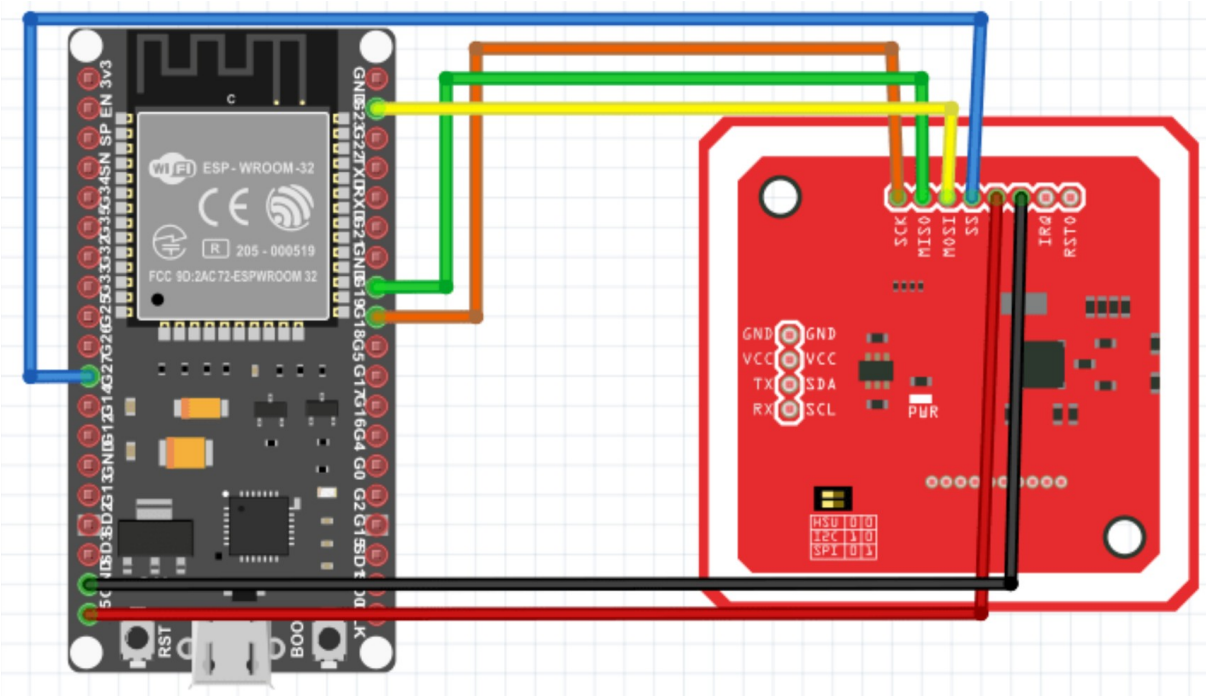
To establish seamless communication between the PN532 RFID board and the ESP32 microcontroller for efficient RFID tag detection and data transmission.

REQUIRED COMPONENTS:

- ESP32 development board
- RFID PN532 module
- Connecting Wires

- Breadboard

CONNECTIONS:



ESP32	PN532
VCC	VCC
GND	GND
18	SCK
19	MISO
23	MOSI
27	SS

CODE:

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 08: Interfacing TJA1051 High-Speed CAN Transceiver with ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of CAN bus interface

PRE-LAB:

1. What is CAN bus interface?

2. How to make use of CAN interface communicating using ESP 32?

3. What are the specific pins on the ESP32 that will be used to communicate with the PN532 module?

4. How will the ESP32 handle data received from RFID tags via the PN532 board?

OBJECTIVE:

To establish seamless communication between the PN532 RFID board and the ESP32 microcontroller for efficient RFID tag detection and data transmission.

REQUIRED COMPONENTS:

- ESP32 development board
- TJA1051 module
- Connecting Wires

- Breadboard

THEORY:

TJA1051 High-Speed CAN Transceiver

TJA1051 is a High-Speed CAN transceiver from NXP. Available in different 8-pin SMD packages, it supports speeds up to 5 Mbps. Yes, it also supports CAN-FD. TJA1051 supports dual supply voltages, one for the transceiver section (VCC) and one for the logic section (VIO). VCC should be from 4.5-5.5V and VIO can be from 2.8-5.5V. We will use 5V as VCC and 3.3V as VIO.

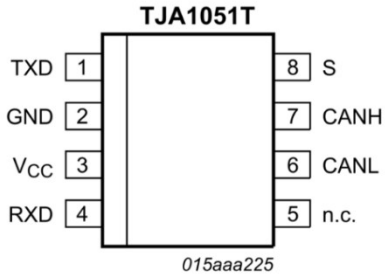
Features

- Fully ISO 11898-2:2003 compliant
- Timing guaranteed for data rates up to 5 Mbit/s in the CAN FD fast phase
- Suitable for 12 V and 24 V systems
- Low Electro Magnetic Emission (EME) and high Electro Magnetic Immunity
- VIO input on TJA1051T/3 and TJA1051TK/3 allows for direct interfacing with 3 V to 5 V microcontrollers (available in SO8 and very small HVSON8 packages respectively)
- EN input on TJA1051T/E allows the microcontroller to switch the transceiver to a very low-current Off mode
- Available in SO8 package or leadless HVSON8 package (3.0 mm × 3.0 mm) with improved Automated Optical Inspection (AOI) capability
- Dark green product (halogen free and Restriction of Hazardous Substances (RoHS) compliant)
- AEC-Q100 qualified
- Functional behaviour predictable under all supply conditions
- Transceiver disengages from the bus when not powered up (zero load)
- High Electro-Static Discharge (ESD) handling capability on the bus pins
- Bus pins protected against transients in automotive environments
- Transmit Data (TXD) dominant time-out function
- Undervoltage detection on pins VCC and VIO
- Thermally protected

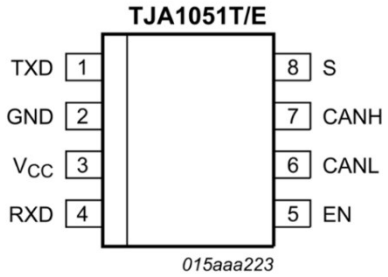
Symbol	Pin	Description
TXD	1	Transmit data input
GND	2	Ground
Vcc	3	Supply voltage
RXD	4	Receive data output; reads out data from the bus lines
NC	5	Not connected in TJA1051T version
EN	5	Enable control input in TJA1051T/E only
VIO	5	Supply voltage for I/O level adapter. TJA1051T/3 and TJA1051TK/3 only
CANL	6	LOW-level CAN bus line
CANH	7	HIGH-level CAN bus line
S	8	Silent mode control input. HIGH = TRX off, LOW = TRX on

TJA1051 pinout

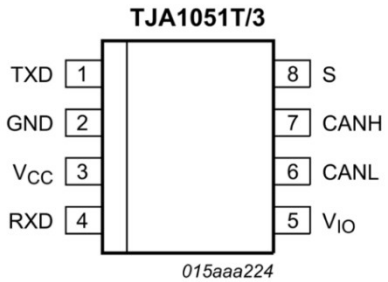
PIN CONFIGURATION DIAGRAM:



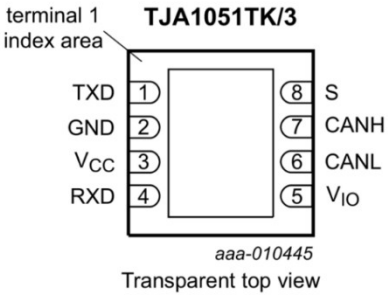
a. TJA1051T: SO8



b. TJA1051T/E: SO8

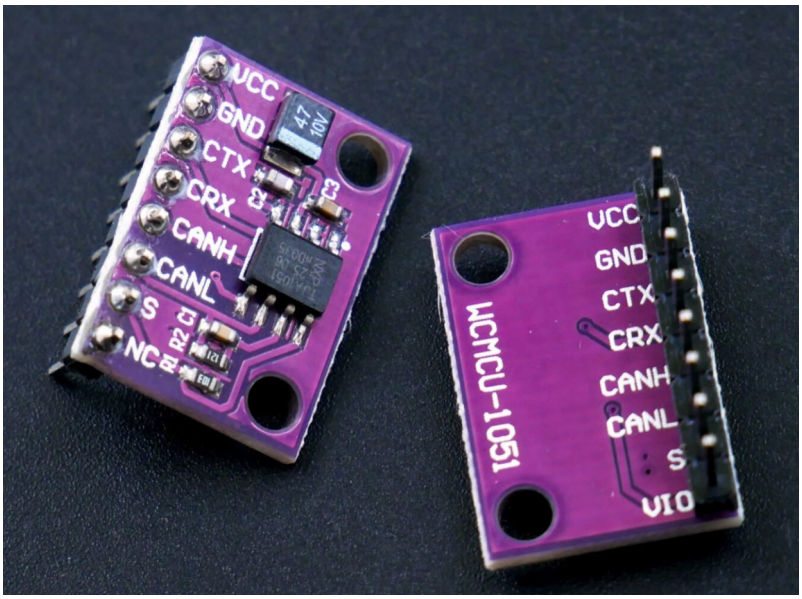


c. TJA1051T/3: SO8

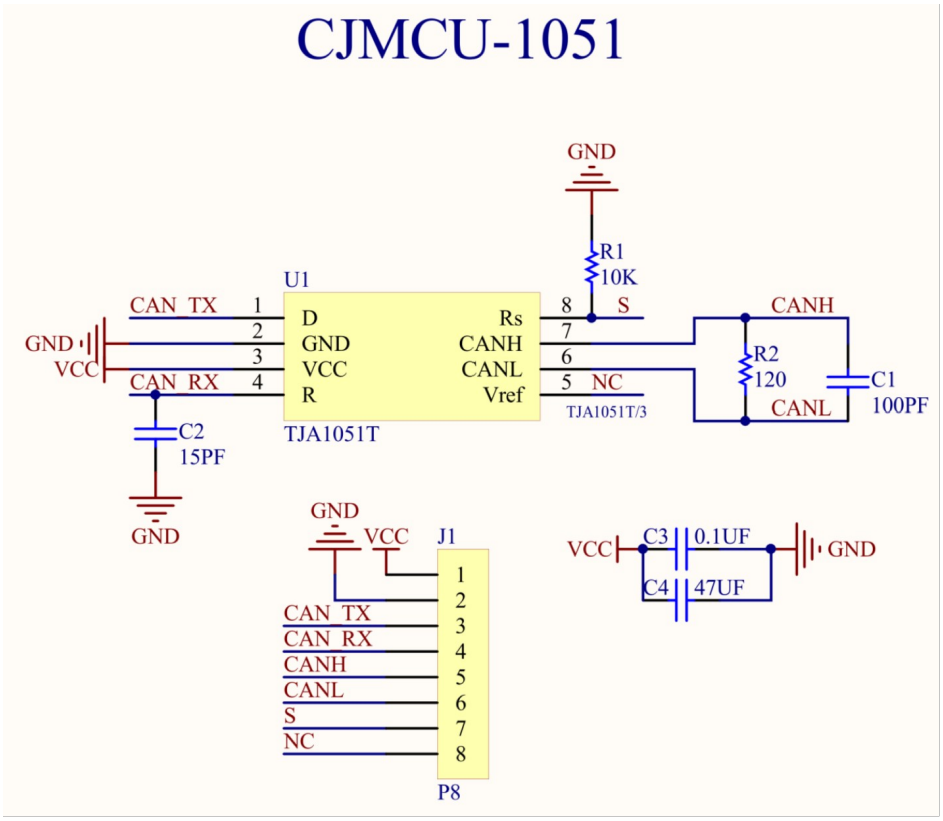


d. TJA1051TK/3: HVSON8

CJMCU-1051

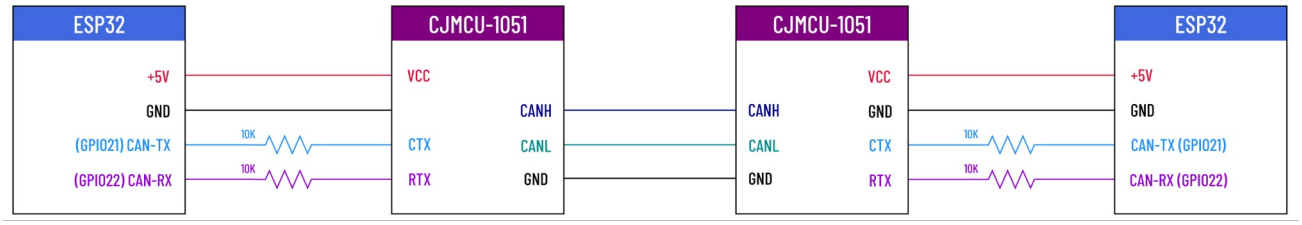


CONNECTIONS:



ESP32 and CJMCU-1051 wiring

We need two ESP32 boards and two CAN transceivers for this experiment. Connect all the pins as shown below. Note that the CAN-RX and CAN-TX pins should be connected to CRX and CTX respectively. It is different from the convention used for UARTs. We are using 6.8K resistors in series with RX and TX lines to be safe with the different voltages. You can also use a dedicated four-channel level-shifter module. If you need a complete pinout reference for ESP32, we also have a dedicated page for that.



CODE:

How the code works?

For the Arduino framework, we will use Sandeep Mistry‘s Arduino-CAN library that supports MCP2515 standalone controller from Microchip as well as the ESP32’s integrated CAN controller. You can install this library straight from the Arduino IDE by searching for “can”.

The library comes with many examples, and we have combined the Sender and Receiver sketches together in the code below. In the loop section, just need the functionality you need.
We are using GPIO21 as TX pin and GPIO22 as RX pin for this example. But you can change it to any GPIO pins. To set a custom pin for the CAN controller, we can use the API function setPins(). After that, we can initialize the CAN controller with begin() function by also passing the nominal bit rate. We are using 500E3 here which is 500 kbps.

In the canSender() function, we will send two types of CAN messages. The first one is a standard message with 8 bytes of data payload and a standard ID (11-bit) of 0x12. We can use beingPacket() to start a new message by passing the ID and then use write() function to write any messages. Make sure to limit the data to 8 bytes. The data length (DLC) will be automatically calculated from the data you are sending. Finally use endPacket() to send the message. Just after sending a normal data frame, we will send an RTR packet requesting for 3 bytes of data. beginPacket() function has a few overloads and we can send additional parameters as we need. For an RTR message, the third parameter should be true.

In the canReceiver() function, we will continuously check for new incoming CAN messages. If we receive a message, the packetSize will return a non-zero value. We will then check the packet type and use the read() function to read the CAN message data.

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 09: ADC Calibration with ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of Analog to Digital Converter (ADC)

PRE-LAB:

1. What is ADC and where it can be used?

2. How many ADC pins are there in ESP 32?

3. List out the features of ADC in ESP 32.

OBJECTIVE:

To ensure precise and consistent conversion of analog signals into accurate digital values.

REQUIRED COMPONENTS:

- ESP32 development board
- Connecting Wires
- Breadboard

THEORY:

ESP32 ADC Calibration:

The ESP32 Arduino Core ADC driver's API provides functions to correct for differences in measured voltages caused by variation of ADC reference voltages (VREF) between ESP32 chips.

Correcting ADC readings using this API involves characterizing one of the ADCs at a given attenuation to obtain a characteristics curve (ADC-Voltage curve) that takes into account the difference in ADC reference voltage.

The characteristics curve is in the form of y = coeff_a * x + coeff_b and is used to convert ADC readings to voltages in mV. Calculation of the characteristics curve is based on calibration values which can be stored in eFuse or provided by the user.

CONNECTION PROCEDURE:

Make sure to connect the potentiometer to that analog channel pin and tweak the pot until you get 2v (or any other value). Just make sure the DMM is reading an exact value (2v or any other value).

Now, read the ADC with the analogRead() function without calibration or whatsoever and note down that value. For me, it was 1.85v, while the DMM is reading an exact 2v. Then, incorporated the ADC calibration functions in the code.

CODE:

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 10: ADC Noise Reduction by Multi-Sampling & Moving Average Digital Filtering with ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of ESP32 ADC basics resolution, sampling rates, and channel configurations.

PRE-LAB:

1. What are the common sources of noise affecting ADC readings in the ESP32?

2. How does multi-sampling help in reducing noise, and what are its limitations?

3. What is a moving average filter, and how does it operate in the context of ADC data?

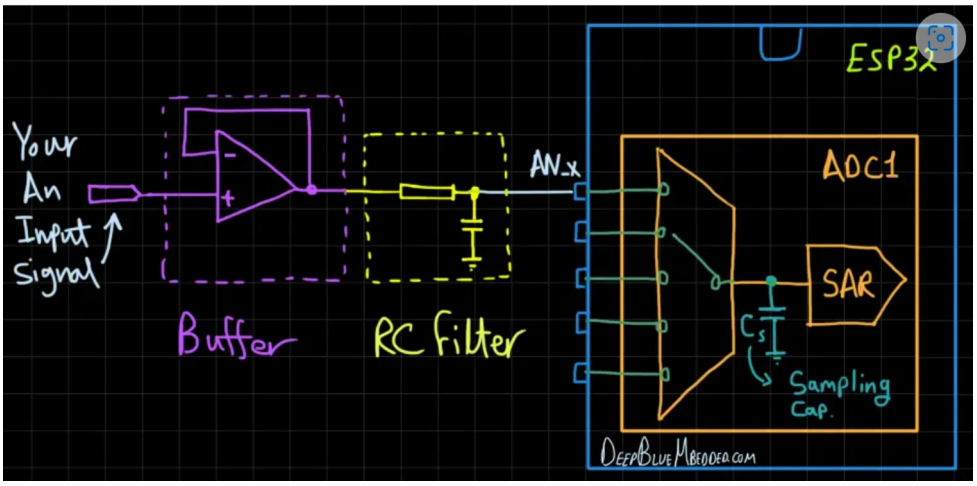
4. How does the moving average technique differ from other digital filtering methods?

OBJECTIVE:

To implement multi-sampling and moving average digital filtering techniques on the ESP32's ADC to effectively reduce noise, enhancing the accuracy and reliability of analog-to-digital conversions.

THEORY:

The ESP32 ADC can be sensitive to noise leading to large discrepancies in ADC readings. To minimize noise, users may connect a 0.1uF capacitor to the ADC input pad in use. Multi-sampling may also be used to further mitigate the effects of noise. Usually, everyone incorporate an active buffering for ADC inputs in most of the designs. A great advantage of having an active buffering (like a voltage follower op-amp config.) is that it reduces the ADC channels cross-coupling while the ADC is switching from channel to channel, the internal ADC's sampling capacitor will end up picking some measurement noise due to this. Having an active buffer will eliminate this sort of error.



REQUIRED COMPONENTS:

- ESP32 development board
- Potentiometer
- Connecting Wires
- Connecting cable
- Breadboard

CODE:

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 11: Generating audio output using DAC of ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of DAC operation
- Understand the connections of amplifiers, speakers, or headphones

PRE-LAB:

1. What is the role of a DAC in converting digital signals to analog output for audio generation?

2. How many DAC channels does the ESP32 have, and what are their specifications?

3. What are the key components of an audio signal, including frequency, sampling rate, and amplitude?

4. What are the necessary steps to configure and initialize the DAC on the ESP32 for audio output?

OBJECTIVE:

To utilize the ESP32 DAC to generate audio output using LM386 audio power amplifier

REQUIRED COMPONENTS:

- ESP32 development board
- Connecting cable
- Connecting Wires
- Breadboard
- LM386 audio power amplifier

THEORY:

ESP32 Audio Music with DAC + TouchPADs:

Here, we read 4 different touch pads and will generate a different tone for each touchpad as long as it's "touched". When the user releases the pad, the audio tone shall stop immediately.

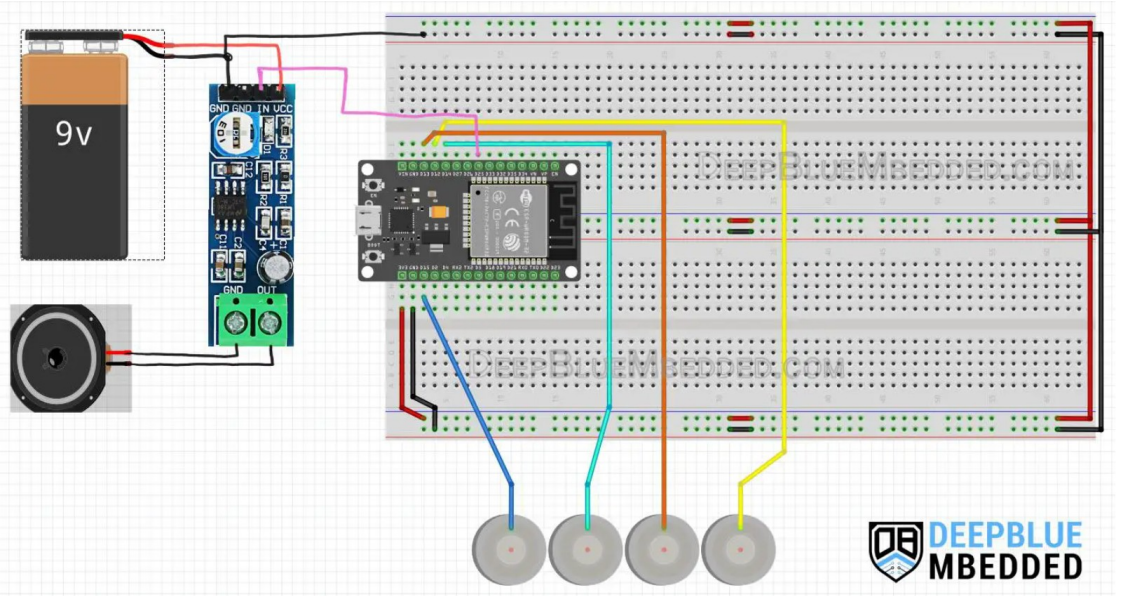
The Arduino code example for this application is very similar to the previous example but it only has 2 main additional operations going on. The first one, is we declare and read 4 touch pads for the input detection, and the sine waveform frequency control is done by changing the DAC_Sampling_Rate or in other words the Timer0_Interrupt time interval. They are essentially the same thing DAC_Sampling_Freq = (1/TimerInterrupt_TimeInterval).

The 4 tones we'll generate have the following frequencies: (200Hz, 500Hz, 1000Hz, 1428Hz). Use the equations from the previous example to validate the required configurations to achieve each frequency or to implement your own set of tones if you need to.

$$F_{out} = \frac{1}{Ns * Ts}$$

Keep the sample points count (Ns) that represent the lookuptable length as fixed. Decide on your desired output frequency (F_out) and solve for (Ts) the required Time interval for Timer0 interrupt (which is the DAC sampling time).

CIRCUIT:


9v
GND GND IN VCC
GND OUT
DEEPBLUEMBEDDED.COM
DEEPBLUE
MBEDDED

CODE:

Code Explanation:

The main loop() function starts by reading the 4 touch PADs and checking if any is “touched”. The threshold value will change depending on your touchpad material and surface. If a pad is touched, we change the Timer0 auto-reload register’s value, which in turn changes the timer interrupt interval. This will increase/decrease the DAC sampling rate and ultimately change the output sine waveform’s frequency. We finally assign the updated value of the Timer0_ARR depending on which pad is currently touched. If none is touched, we disable the DAC output.

```
void loop()
{
 // Read The Touch PADs
 TOUCHPAD_Values[0] = touchRead(TOUCHPAD_1);
 TOUCHPAD_Values[1] = touchRead(TOUCHPAD_2);
 TOUCHPAD_Values[2] = touchRead(TOUCHPAD_3);
 TOUCHPAD_Values[3] = touchRead(TOUCHPAD_4);

 if(TOUCHPAD_Values[0] < TOUCH_THR)  // Play 200Hz Tone
 {
  TMR0_ARR_Val = 50;
  dac_output_enable(DAC_CHANNEL_1);
 }
 else if(TOUCHPAD_Values[1] < TOUCH_THR) // Play 500Hz Tone
 {
  TMR0_ARR_Val = 20;
  dac_output_enable(DAC_CHANNEL_1);
 }
 else if(TOUCHPAD_Values[2] < TOUCH_THR) // Play 1000Hz Tone
 {
  TMR0_ARR_Val = 10;
  dac_output_enable(DAC_CHANNEL_1);
 }
 else if(TOUCHPAD_Values[3] < TOUCH_THR)  // Play 1428Hz Tone
 {
  TMR0_ARR_Val = 7;
  dac_output_enable(DAC_CHANNEL_1);
 }
 else
 {
  dac_output_disable(DAC_CHANNEL_1);
 }
 // Update The Timer Interrupt Interval According To The Desired Tone
 timerAlarmWrite(Timer0_Cfg, TMR0_ARR_Val, true);
}
```

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 12: Save and read data in Flash Memory of ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 board
- General idea of flash memory on the ESP32
- Familiarity with the ESP32-specific functions and APIs used for interacting with the flash memory

PRE-LAB:

1. What are the distinctive features of flash memory, and how does it differ from other types of memory?

2. How much memory does the ESP32's flash memory have?

3. How the data is organized and stored in flash memory?

4. Can you name a few functions used for reading and writing data to flash memory on the ESP32?

OBJECTIVE:

To test the Flash memory by saving and reading a string to the Flash memory

REQUIRED COMPONENTS:

- ESP32 development board
- Connecting cable
- Connecting wires
- Breadboard

THEORY:

The ESP32 has an external SPI Flash memory that's by default 4MB in size. But generally speaking, we'll say it's 4MB. And those 4MB of Flash memory space are divided into some partitions for various uses. You can however add-remove-modify those partitions as per your application's need.

The ESP32 default Flash partition table has:

- NVS: for user data storage
- 2x App Partitions: for user program storage + ability to perform OTA firmware updates
- OTAdata: used for OTA (Over-The-Air) firmware updates
- SPIFFS: SPI Flash File System for file storage and management
- CoreDump: for debugging and diagnostics purposes

We are specifically interested in the NVS partition, which is meant to be used for user data storage. It has a space of 0x5000 bytes (20kB), and it has an address offset of 0x9000. This means any write operation to the Flash memory at the absolute address 0x9000 will go into the NVS partition and you can continue from that address up to the end of the NVS partition.

ESP32 Flash Memory Read Write Functions:

There is a couple of functions in the built-in ESP.h that provide us direct access to the full SPI Flash Memory to perform read and write operations to literally any address in the entire memory space. This means it's our responsibility to check the validity of the address which we're writing to.

To use those 2 APIs (Functions), you have to first include the <Arduino.h> built-in header file. No libraries are required for this method as stated earlier.

```
1 // Include Arduino.h (To Access ESP.h)
```

ESP32 Flash Write:

This is the ESP32 Flash Write API (Function)

```
bool flashWrite(uint32_t offset, uint32_t *data, size_t size);
```

It takes the offset from the beginning of the Flash memory space which is basically the address you want to write to, a pointer to the data to be written, and the size of the data (number of bytes). It's worth noting that we need to handle the addressing and offset checking on our own, and also the data type conversion from and to (uint32_t) which is what this function expects as an input.

Moreover, we need also to keep track of how many bytes have been written to the memory to determine what should be the address value for the next piece of data to be written.

For example, let's say we want to write a uint32_t myVar to the Flash memory at address zero, here is how to do it in code.

```
const uint32_t NVM_Offset = 0x290000;  // Offset Value For NVS Partition
uint32_t address = 0; // Address Zero
uint32_t myVar = 100; // My Variable To Save

// Write To The Flash
ESP.flashWrite(NVM_Offset+address, &myVar , sizeof(myVar));
```

Note that we have to add the offset value to the beginning of the NVS partition in the Flash memory. And that's how we write a variable to the ESP32 Flash memory without any additional libraries.

ESP32 Flash Read:

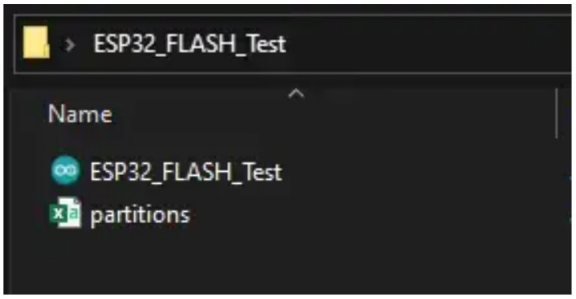
Similarly, we'll use the following API (Function) to read a memory location from the Flash memory. And note that we have to know the exact size of the data element that we want to retrieve from the Flash.

```
bool flashRead(uint32_t offset, uint32_t *data, size_t size);
```

CODE:

Partitions.csv File:

Don’t forget that we need to have our custom partitions.csv file in the same folder with the sketch to access the NVM partition correctly.



Code Explanation:

The code above does simple a write operation to the Flash memory which sends a string called StrIn and then read from the same address into a new string variable StrOut and then we print the latter one to see if it’s read successfully from the memory.

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 13: Testing of Flash Memory of ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of Interrupts, ESP32 board
- General idea of flash memory on the ESP32
- Familiarity with the ESP32-specific functions and APIs used for interacting with the flash memory

PRE-LAB:

1. How much memory does the ESP32's flash memory have?

2. Give the specific details of the ESP32's flash memory, including its size, organization, and access characteristics.

3. Explain the methods are commonly used to test flash memory functionality, such as read/write operations

OBJECTIVE:

To test the Flash memory by saving and reading a Float variable to the Flash memory in the setup() function

COMPONENTS REQUIRED:

- ESP32
- Breadboard

- Connecting cable
- Connecting wires

CODE:

Partitions.csv File:

Don’t forget that you need to have custom partitions.csv file in the same folder with the sketch to access the NVM partition correctly.

Code Explanation:

The code above does simple a write operation to the Flash memory which sends a Float variable called Pi and then read from the same address into a new Float variable Read_Pi and then we print the latter one to see if it’s read successfully from the memory

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 14: Control ESP 32 using Bluetooth of Android phone

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 development board
- General idea of Bluetooth technology protocols and communication process between devices
- Understanding how to interface Android app for sending commands or controlling functions on the ESP32

PRE-LAB:

1. What are the different Bluetooth versions, and which ones are commonly used for IoT applications like ESP32 interfacing?

2. What are the Bluetooth capabilities of the ESP32, and which protocols (BLE, Classic Bluetooth) does it support?

3. Which communication protocols are commonly used for Android-to-ESP32 communication over Bluetooth?

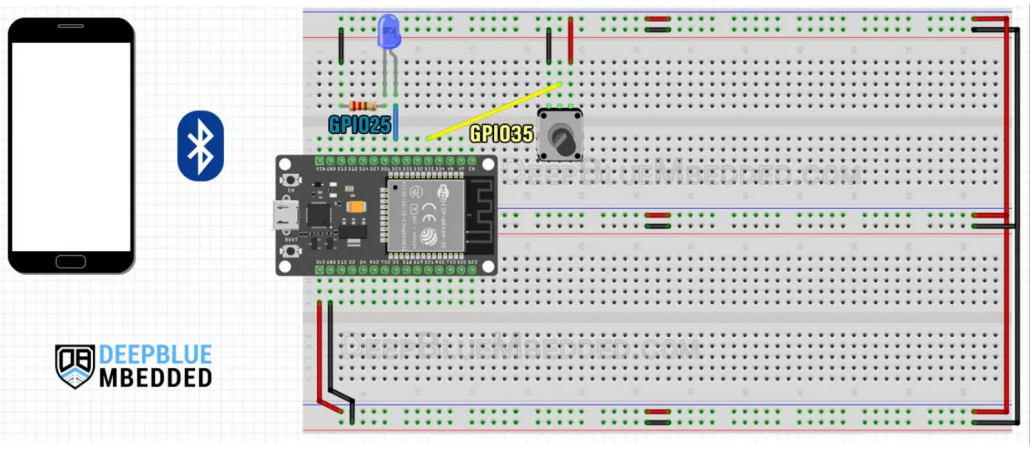
OBJECTIVE:

To establish Bluetooth communication using Bluetooth of Android phone and an ESP32 to exchange the data between the devices

COMPONENTS REQUIRED:

- ESP32
- Android mobile
- Potentiometer
- Breadboard
- Connecting cable
- Connecting wires

CONNECTION:



CODE

Code Explanation:

Configuring **Timer0** to trigger an interrupt periodically every 100ms in the setup() function.

```
// Configure Timer0 Interrupt (Every 100mS)
Timer0_Cfg = timerBegin(0, 80, true);
timerAttachInterrupt(Timer0_Cfg, &Timer0_ISR, true);
timerAlarmWrite(Timer0_Cfg, 100000, true);
timerAlarmEnable(Timer0_Cfg);
```

Define the **Timer0 ISR** (interrupt service routine) handler function in which we'll read the ADC channel for the potentiometer and send its value over Bluetooth.

```
// Timer0 ISR Handler Function (Configured To Execute Every 100ms)
void IRAM_ATTR Timer0_ISR()
{
  // Read The Potentiometer ADC Channel & Send Raw Data Over Bluetooth
  SerialBT.println(analogRead(AN_Pot));
}
```

And that's all, the **loop()** function will remain the same. It's handling the Bluetooth data reception and processing, while the Bluetooth data transmission is handled in the Timer0 ISR which executes every 100ms.

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 15: Bluetooth Communication between Two ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 development board
- General idea of Two-Way Serial Bluetooth communication

PRE-LAB:

1. How do you configure the ESP32 devices to enable Bluetooth communication between them?

2. What is the process for pairing two ESP32 devices via Bluetooth?

3. What methods or protocols can be utilized for transmitting data between two ESP32 devices over Bluetooth?

4. List out the factors affect the Bluetooth signal strength and range between two ESP32 devices.

OBJECTIVE:

To establish a Two-Way Serial Bluetooth connection link between two ESP32 boards for data transmission.

COMPONENTS REQUIRED:

- ESP32 2 Nos.
- Breadboard

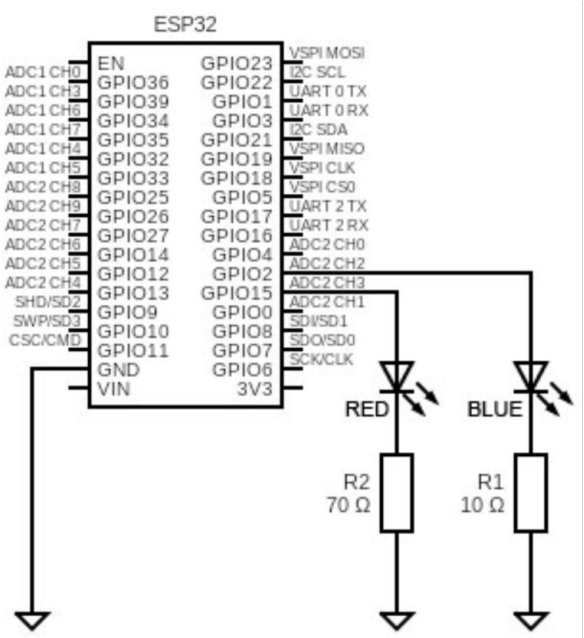
- Connecting cable
- Connecting wires

CONNECTION PROCEDURE:

The **SERVER** (Master) side uses a two-colors LED (Blue/Red) to indicate the connection state.
The **red** color indicates three states:
The connection is not established yet

- The connection was lost
- There are reconnect attempts running including the Bluetooth device reset
- The **blue** color indicates that the connection is established and working now

The schematic below shows an example of the the two-colors indication of the connection state. The R1 and R2 values on the schematic are approximate and calculated for my specific solution with the LL-309VBC2E-C-2B red/blue LED.



Server (Master) indication LEDs connection

CODE:

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 16: Connecting to Wi-Fi Network using ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 Wi-Fi features and available protocols
- General idea of Wi-Fi network access SSID and Password

PRE-LAB:

What are the available protocols of Wi-Fi technology?

What are the supported Wi-Fi protocols and modes available on the ESP32?

List the details required for ESP32 Wi-Fi Connection.

OBJECTIVE:

To configure and establish a stable Wi-Fi connection on the ESP32.

COMPONENTS REQUIRED:

- ESP32
- Connecting cable

THEORY:

ESP32 WiFi Modes:

The ESP32 WiFi can run in one of the following modes: WiFi Station, Access Point, or Both at the same time. There is an additional mode called Promiscuous mode in which the ESP32 will act as a WiFi sniffer. The ESP32 Library does support the first 3 modes by default. To set the ESP32 WiFi mode, you can use the WiFi.mode() function which takes one argument as an input (the desired mode).

The network here is established by the router (AP) to which the devices are connected. The ESP32 connects to the WiFi network of your router and gets assigned a unique IP address that can be used to communicate with other devices on the network.

Station Mode (STA): the ESP32 connects to other existing networks (like your router at home or any other access point)	`WiFi.mode(WIFI_STA);`
Access Point Mode (AP): the ESP32 creates its own network by acting as an access point that other WiFi stations can connect to it	`WiFi.mode(WIFI_AP);`
Access Point + Station Mode (AP_STA): in this mode, the ESP32 WiFi will act as both Access Point and WiFi Station Simultaneously	`WiFi.mode(WIFI_AP_STA);`

ESP32 Scan Wi-Fi Networks:

The ESP32 can scan for WiFi networks within its range and return the found networks' SSIDs and signal strength for each network. You'll find a sample code example in Arduino IDE for ESP32 WiFi Scanner. Open File > Examples > WiFi > WiFiScan sketch.

CODE

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 17: Wi-Fi reconnect using Esp 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 Wi-Fi features and available protocols
- General idea of Wi-Fi network access SSID and Password

PRE-LAB:

1. What are the common reasons for Wi-Fi disconnection on the ESP32

2. How can the ESP32 monitor the Wi-Fi connection status in real-time to detect disconnections promptly?

3. How many reconnection attempts should the ESP32 make before considering alternative actions or notifying the user?

OBJECTIVE:

To reconnect to a Wi-Fi network after disconnection using ESP32.

COMPONENTS REQUIRED:

- ESP32
- Connecting cable

Wi-Fi Reconnect:

The ESP32 may sometimes fail to connect to a Wi-Fi network temporarily and the best way to get the connection back is just to restart and re-attempt the connection once again. This can be

implemented by adding a timeout counter and allowing some time for ESP32 Wi-Fi connection establishment. If it's exceeded, commit a restart operation for the ESP32.

CODE:

Note:

You can also try it by disabling your router's WLAN (WiFi) while the ESP32 is trying to connect or just by giving it a wrong SSID or a wrong password in the string within your code, it should also keep restarting every 10 seconds.

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 18: Wi-Fi station using ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 Wi-Fi features and available protocols
- General idea of Wi-Fi network access SSID and Password

PRE-LAB:

1. What are the essential components required to configure an ESP32 as a Wi-Fi station?

2. Can the ESP32 handle connection to multiple Wi-Fi networks in station mode

3. What Wi-Fi protocols and modes does the ESP32 support for station configuration?

4. How does the ESP32 handle connection errors or failures during the Wi-Fi station setup or while attempting to connect to a network?

OBJECTIVE:

To configure the ESP32 as a Wi-Fi station, allowing it to successfully connect to a designated Wi-Fi network using the provided credentials

COMPONENTS REQUIRED:

- ESP32
- Connecting cable

CODE:

Code Explanation:

We simply need to call the WiFi.onEvent() function and pass to it 2 parameters. The first parameter is the handler function that we've created and would like to get it called whenever a specific event happens. The second parameter is the ID or name of the event that we'd like to trigger the callback function on.

Here we are giving it a function to be called whenever the WiFi connection is successful.

```
WiFi.onEvent(ConnectedToAP_Handler, ARDUINO_EVENT_WIFI_STA_CONNECTED);
```

And here I'm giving it a function to be called whenever the ESP32 WiFi station gets an IP from the Access Point.

```
WiFi.onEvent(GotIP_Handler, ARDUINO_EVENT_WIFI_STA_GOT_IP);
```

This is the function that I've defined to be called on a successful connection event. It's going only to print a message on the serial terminal.

```
void ConnectedToAP_Handler(WiFiEvent_t wifi_event, WiFiEventInfo_t wifi_info) {
  Serial.println("Connected To The WiFi Network");
}
```

And this is the function that I've defined to be called on getting an IP from the Access Point. It's going to print the ESP32 WiFi IP.

```
void GotIP_Handler(WiFiEvent_t wifi_event, WiFiEventInfo_t wifi_info) {
  Serial.print("Local ESP32 IP: ");
  Serial.println(WiFi.localIP());
}
```

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 19: Create Wi-Fi (Access Point) Web Server using ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 Wi-Fi network features and available protocols
- General idea of Wi-Fi station and access point

PRE-LAB:

1. What are the essential components required to configure an ESP32 as a Wi-Fi station?

2. What defines a Wi-Fi access point in networking, and how does it differ from other Wi-Fi modes (e.g., station mode)?

3. How are these network parameters obtained or set up within the ESP32 code for the access point?

4. Define the parameters which is necessary to configure the ESP32 as a Wi-Fi access point

OBJECTIVE: To configure the ESP32 as a Wi-Fi access point and host a functional web server, allowing connected client to access and interact with webservices.

COMPONENTS REQUIRED:

- ESP32
- Connecting cable

CODE:

Code Explanation:

First of all, you know that the ESP32 WiFi can operate as a WiFi station (**STA**) that connects to existing networks or as a WiFi access point (**AP**) that creates its own network. Given that we're trying to connect to an existing WiFi network we should set the ESP32 WiFi mode to STA mode. And this is what we've done in the first step.

```
WiFi.mode(WIFI_STA);
```

Next, we call the WiFi.begin() function which attempts to connect the ESP32 to the desired network. You should have edited the ssidand passwordstrings to match the actual network name and password for your WiFi network.

```
WiFi.begin(ssid, password);
```

The connection process is not instantaneous, it's going to take some time. So, we need to add a delay and keep checking (polling) for the connection status. We'll be using the WiFi.status() function to get the WiFi connection status, and it should return WL_CONNECTED whenever the connection is successfully established.

```
while(WiFi.status() != WL_CONNECTED)
{
  Serial.print(".");
  delay(100);
}
```

Whenever a successful connection is established, the while loop will break and we'll print a message to the serial monitor indicating this event. And we'll also get the local IP for the ESP32 and print it as well.

```
Serial.println("\nConnected to the WiFi network");
Serial.print("Local ESP32 IP: ");
Serial.println(WiFi.localIP());
```

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 20: Client-Server Wi-Fi Communication Between Two ESP 32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 Wi-Fi network features and available protocols
- General idea of Wi-Fi communication client-server

PRE-LAB:

1. Define client-server communication in a Wi-Fi network. how does it differ from direct device-to-device communication?

2. What parameters are necessary to configure one ESP32 as a Wi-Fi server and another as a Wi-Fi client, such as SSID (network name), and password?

3. How are these network parameters configured in both ESP32 devices for client-server communication?

4. What steps are involved in establishing a connection between the client and server ESP32 devices over Wi-Fi?

OBJECTIVE: To enable client-server communication between two ESP32 devices over Wi-Fi, configuring one as a server and the other as a client, allowing them to exchange data.

COMPONENTS REQUIRED:

- ESP32 2 Nos.
- Connecting cable
- BME280 sensor
- I2C SSD1306 OLED display

- Connecting wires

Installing Libraries:

Asynchronous Web Server Libraries:

We'll use the following libraries to handle HTTP request:

- ESPAsyncWebServer library (download ESPAsyncWebServer library)
- Async TCP library (download AsyncTCP library)

These libraries are not available to install through the Library Manager. So, you need to unzip the libraries and move them to the Arduino IDE installation libraries folder.

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

BME280 Libraries:

The following libraries can be installed through the Arduino Library Manager. Go to **Sketch** > **Include Library**> **Manage Libraries** and search for the library name.

- Adafruit_BME280 library
- Adafruit unified sensor library

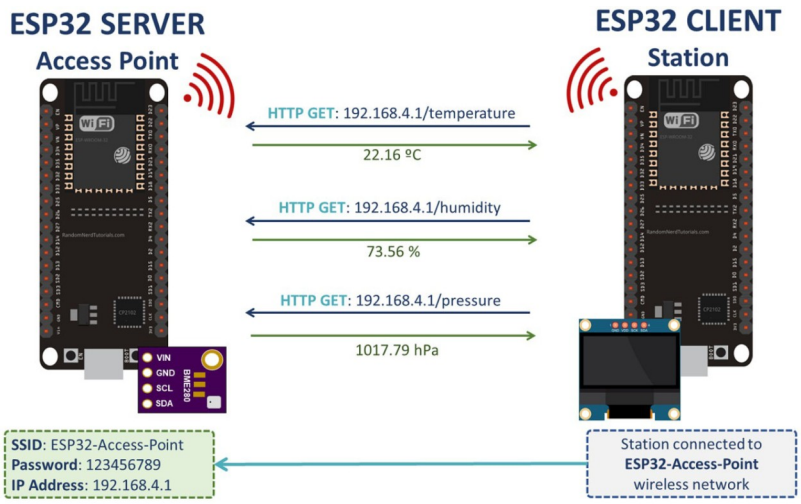
I2C SSD1306 OLED Libraries

To interface with the OLED display you need the following libraries. These can be installed through the Arduino Library Manager. Go to **Sketch** > **Include Library**> **Manage Libraries** and search for the library name.

- Adafruit SSD1306
- Adafruit GFX Library

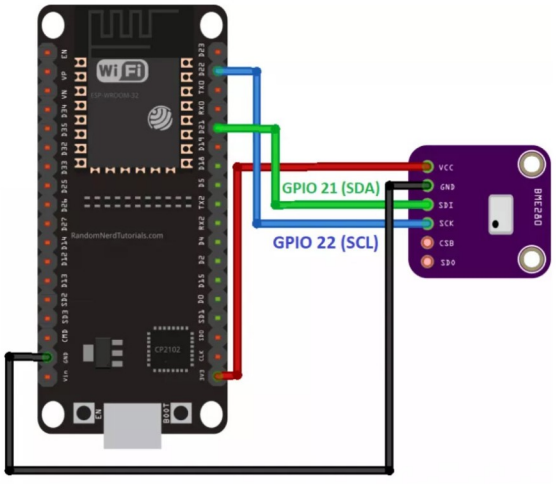
#1 ESP32 SERVER (ACCESS POINT)

One ESP32 board will act as a server and the other ESP32 board will act as a client. The following diagram shows an overview of how everything works.



- The ESP32 server creates its own wireless network (ESP32 Soft-Access Point). So, other Wi-Fi devices can connect to that network (SSID: ESP32-Access-Point, Password: 123456789).
- The ESP32 client is set as a station. So, it can connect to the ESP32 server wireless network.
- The client can make HTTP GET requests to the server to request sensor data or any other information. It just needs to use the IP address of the server to make a request on a certain route: /temperature, /humidity or /pressure.
- The server listens for incoming requests and sends an appropriate response with the readings.
- The client receives the readings and displays them on the OLED display.

CONNECTION



BME280	ESP32
VIN/VCC	3.3V
GND	GND
SCL	GPIO 22
SDA	GPIO 21

CODE:

Code explanation:

Start by including the necessary libraries. Include the WiFi.h library and the ESPAsyncWebServer.h library to handle incoming HTTP requests.

```
#include "WiFi.h"
#include "ESPAsyncWebServer.h"
```

Include the following libraries to interface with the BME280 sensor.

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
```

In the following variables, define your access point network credentials:

```
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";
```

We're setting the SSID to ESP32-Access-Point, but you can give it any other name. You can also change the password. By default, its set to 123456789.
Create an instance for the BME280 sensor called bme.

```
Adafruit_BME280 bme;
```

Create an asynchronous web server on port 80.

```
AsyncWebServer server(80);
```

Then, create three functions that return the temperature, humidity, and pressure as String variables.

```
String readTemp() {
  return String(bme.readTemperature());
  //return String(1.8 * bme.readTemperature() + 32);
}

String readHumi() {
  return String(bme.readHumidity());
}

String readPres() {
  return String(bme.readPressure() / 100.0F);
}
```

In the setup(), initialize the Serial Monitor for demonstration purposes.

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

Set your ESP32 as an access point with the SSID name and password defined earlier.

```
WiFi.softAP(ssid, password);
```

Then, handle the routes where the ESP32 will be listening for incoming requests.

For example, when the ESP32 server receives a request on the /temperature URL, it sends the temperature returned by the readTemp() function as a char (that's why we use the c_str() method.

```
server.on("/temperature", HTTP_GET, [](AsyncWebServerRequest *request){
  request->send_P(200, "text/plain", readTemp().c_str());
});
```

The same happens when the ESP receives a request on the /humidity and /pressure URLs.

```
server.on("/humidity", HTTP_GET, [](AsyncWebServerRequest *request){
  request->send_P(200, "text/plain", readHumi().c_str());
});
server.on("/pressure", HTTP_GET, [](AsyncWebServerRequest *request){
  request->send_P(200, "text/plain", readPres().c_str());
});
```

The following lines initialize the BME280 sensor.

```
bool status;

// default settings
// (you can also pass in a Wire library object like &Wire2)
status = bme.begin(0x76);
if (!status) {
  Serial.println("Could not find a valid BME280 sensor, check wiring!");
  while (1);
}
```

Finally, start the server.

```
server.begin();
```

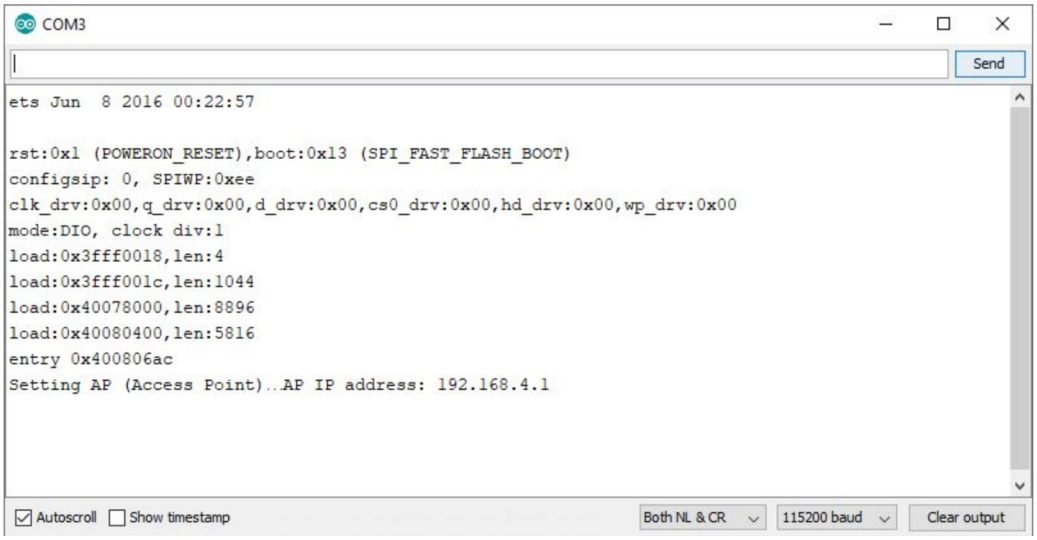
Because this is an asynchronous web server, there's nothing in the loop().

```
void loop(){

}
```

Testing the ESP32 Server

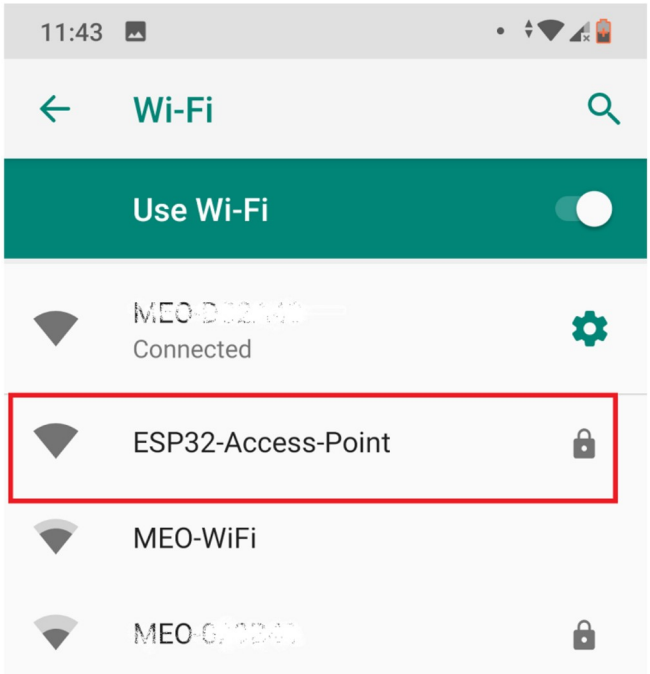
Upload the code to your board and open the Serial Monitor. You should get something as follows:



This means that the access point was set successfully.

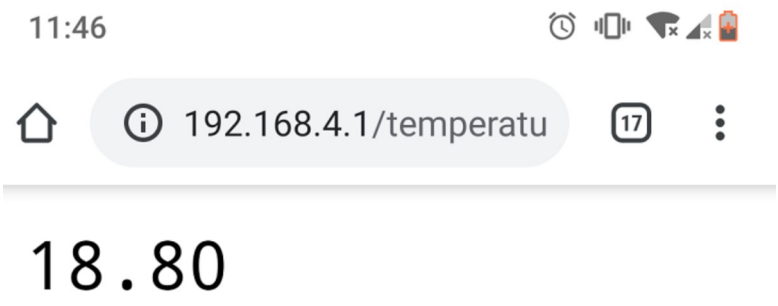
Now, to make sure it is listening for temperature, humidity and pressure requests, you need to connect to its network.

In your smartphone, go to the Wi-Fi settings and connect to the **ESP32-Access-Point**. The password is **123456789**.

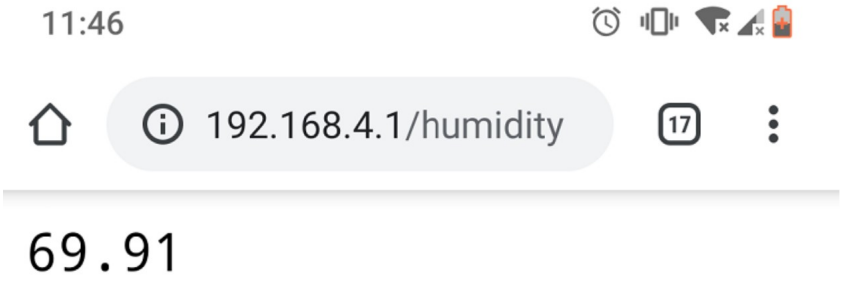


While connected to the access point, open your browser and type 192.168.4.1/temperature

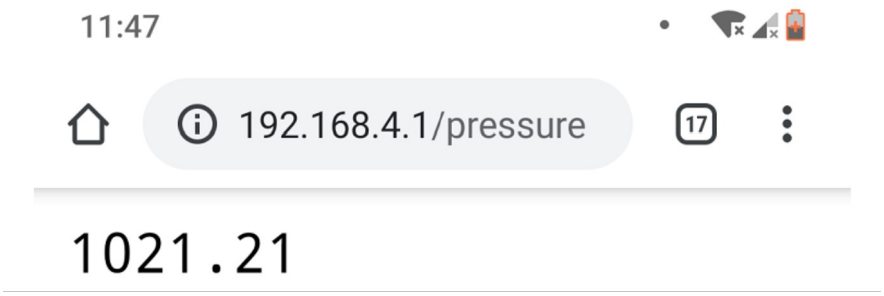
You should get the temperature value in your browser:



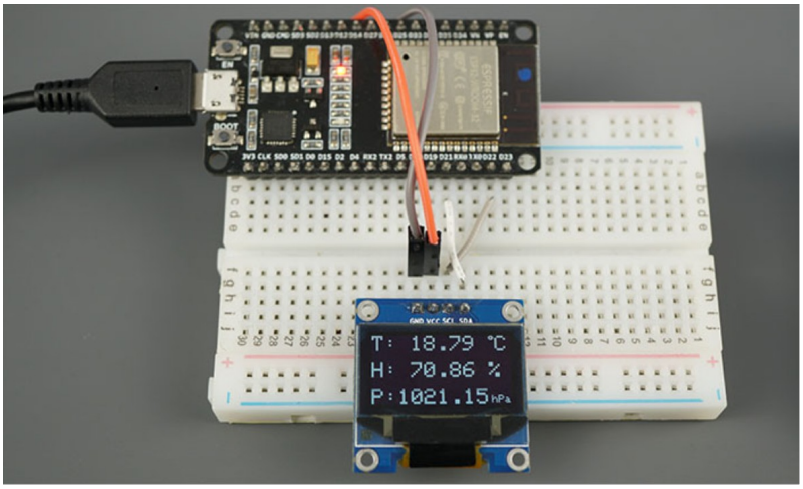
Try this URL path for the humidity 192.168.4.1/humidity:



Finally, go to 192.168.4.1/pressure URL:



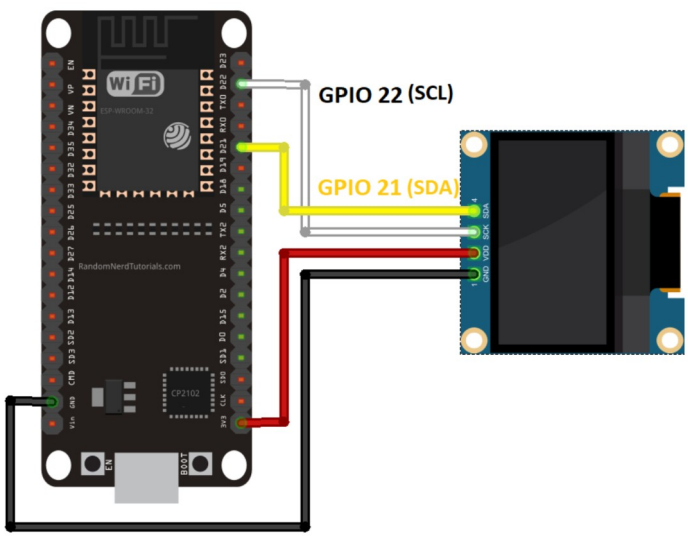
#2 ESP32 CLIENT (STATION)



The ESP32 Client is a Wi-Fi station that is connected to the ESP32 Server. The client requests the temperature, humidity and pressure from the server by making HTTP GET requests on the /temperature, /humidity, and /pressure URL routes. Then, it displays the readings on an OLED display.

CONNECTION:

Wire the ESP32 to the OLED display as shown in the following schematic diagram.



OLED	ESP32
VIN/VCC	VIN
GND	GND
SCL	GPIO 22
SDA	GPIO 21

CODE:

Code explanation:

Include the necessary libraries for the Wi-Fi connection and for making HTTP requests:

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

Insert the ESP32 server network credentials. If you've changed the default network credentials in the ESP32 server, you should change them here to match.

```
const char* ssid = "ESP32-Access-Point";
const char* password = "123456789";
```

Then, save the URLs where the client will be making HTTP requests. The ESP32 server has the 192.168.4.1 IP address, and we'll be making requests on the /temperature, /humidity and /pressure URLs.

```
const char* serverNameTemp = "http://192.168.4.1/temperature";
const char* serverNameHumi = "http://192.168.4.1/humidity";
const char* serverNamePres = "http://192.168.4.1/pressure";
```

Include the libraries to interface with the OLED display:

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

Set the OLED display size:

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

Create a display object with the size you've defined earlier and with I2C communication protocol.

```
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);
```

Initialize string variables that will hold the temperature, humidity and pressure readings retrieved by the server.

```
String temperature;
String humidity;
String pressure;
```

Set the time interval between each request. By default, it's set to 5 seconds, but you can change it to any other interval.

```
const long interval = 5000;
```

In the setup(), initialize the OLED display:

```
// Address 0x3C for 128x64, you might need to change this value (use an I2C scanner)
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
  Serial.println(F("SSD1306 allocation failed"));
  for(;;); // Don't proceed, loop forever
}
```

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

Note: if your OLED display is not working, check its I2C address using an I2C scanner sketch and change the code accordingly.

Connect the ESP32 client to the ESP32 server network.

```
WiFi.begin(ssid, password);
Serial.println("Connecting");
while(WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
}
Serial.println("");
Serial.print("Connected to WiFi network with IP Address: ");
```

In the `loop()` is where we make the HTTP GET requests. We've created a function called `httpGETRequest()` that accepts as argument the URL path where we want to make the request and returns the response as a `String`.

You can use the next function in your projects to simplify your code:

```
String httpGETRequest(const char* serverName) {
  HTTPClient http;

  // Your IP address with path or Domain name with URL path
  http.begin(serverName);

  // Send HTTP POST request
  int httpResponseCode = http.GET();

  String payload = "--";

  if (httpResponseCode>0) {
    Serial.print("HTTP Response code: ");
    Serial.println(httpResponseCode);
    payload = http.getString();
  }
  else {
    Serial.print("Error code: ");
    Serial.println(httpResponseCode);
  }
  // Free resources
  http.end();
```

```
  return payload;
}
```

We use that function to get the temperature, humidity and pressure readings from the server.

```
temperature = httpGETRequest(serverNameTemp);
humidity = httpGETRequest(serverNameHumi);
pressure = httpGETRequest(serverNamePres);
```

Print those readings in the Serial Monitor for debugging.

```
Serial.println("Temperature: " + temperature + " *C - Humidity: " + humidity + " % - Pressure: " +
pressure + " hPa");
```

Then, display the temperature in the OLED display:

```
display.setTextSize(2);
display.setTextColor(WHITE);
display.setCursor(0,0);
display.print("T: ");
display.print(temperature);
display.print(" ");
display.setTextSize(1);
display.cp437(true);
display.write(248);
display.setTextSize(2);
display.print("C");
```

The humidity:

```
display.setTextSize(2);
display.setCursor(0, 25);
display.print("H: ");
display.print(humidity);
display.print(" %");
```

Finally, the pressure reading:

```
display.setTextSize(2);
display.setCursor(0, 50);
```

```
display.print("P:");
display.print(pressure);
display.setTextSize(1);
display.setCursor(110, 56);
display.print("hPa");

display.display();
```

We use timers instead of delays to make a request every x number of seconds. That's why we have the previousMillis, currentMillis variables and use the millis() function. We have an article that shows the difference between timers and delays that you might find useful (or read ESP32 Timers). Upload the sketch to #2 ESP32 (client) to test if everything is working properly.

Testing the ESP32 Client

Having both boards fairly close and powered, you'll see that ESP #2 is receiving new temperature, humidity and pressure readings every 5 seconds from ESP #1.
This is what you should see on the ESP32 Client Serial Monitor.

The sensor readings are also displayed in the OLED.

That's it! Your two boards are talking with each other.

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT:

Session 21: Observation of Hall Effect value and Temperature from ESP32

Date of the Session:___/____/___ **Time of the Session:_____to____**

PREREQUISITE:

- General idea of ESP32 development board
- General idea of ESP32 inbuilt Hall effect and Temperature sensors

PRE-LAB:

1. What is the principle behind the Hall Effect

2. How many inbuilt sensors are there in ESP32?

3. What are the components and characteristics of the hall effect sensor?

OBJECTIVE:

To Observe of Hall Effect value and Temperature from ESP32.

COMPONENTS REQUIRED:

- ESP32
- Connecting cable
- Connecting wires
- Breadboard

THEORY:

A hall effect sensor can detect variations in the magnetic field in its surroundings. The greater the magnetic field, the greater the sensor's output voltage

The hall effect sensor can be combined with a threshold detection to act as a switch, for example. Additionally, hall effect sensors are mainly used to:

- Detect proximity;
- Calculate positioning;
- Count the number of revolutions of a wheel;
- Detect a door closing;

CONENCTION:

There is no external sensor connections for this experiment, because there is inbuilt Hall effect sensor and Temperature Sensor.

CODE:

OUTPUT:

POST LAB

INTERFERENCE & ANALYSIS

RESULT: