

POwer Management system

Using ESP\_32, CT, PT, Oled display and Blynk Integration



January 11, 2025

Abdullah Nazmus-sakib

CSE, Jahangirnagar University, Saver, Dhaka

**Power Management System**

**with ESP32, ACS712, and ZMPT101B**

**Overview**

This project demonstrates a **Power Management System** using the ESP32 microcontroller, current sensor (**ACS712**), voltage sensor (**ZMPT101B**), OLED display (**SSD1306**), and **Blynk IoT platform**. It monitors voltage, current, power, and energy usage, and allows relay control via Blynk. Additionally, it calculates energy cost based on a specified per-unit cost.

**Features**

1. **Real-time Monitoring**:

• Measures **voltage (V)**, **current (mA)**, and **power (W)**.

• Calculates **energy (kWh)** and **cost (TK)**.

2. **OLED Display**:

• Displays measured parameters (Voltage, Current, Power, Energy, and Cost) in real-time.

3. **IoT Integration**:

• Sends data to the **Blynk IoT app** for remote monitoring and control.

4. **Relay Control**:

• Controls a relay module using the Blynk app.

5. **Data Logging**:

• Logs measurements for easy debugging via the Serial Monitor.

**Components Used**

**Hardware:**

1. **ESP32**: Main microcontroller for processing data and WiFi connectivity.

2. **ACS712 (Current Sensor)**: Measures current in the system.

3. **ZMPT101B (Voltage Sensor)**: Measures AC voltage.

4. **SSD1306 OLED Display**: Displays real-time measurements.

5. **Relay Module**: Toggles connected electrical devices ON/OFF.

6. **Resistors and Capacitors**: For noise filtering and stability.

**Software:**

1. **Arduino IDE**: For programming the ESP32.

2. **Blynk IoT Platform**: For remote data monitoring and relay control.

3. Libraries:

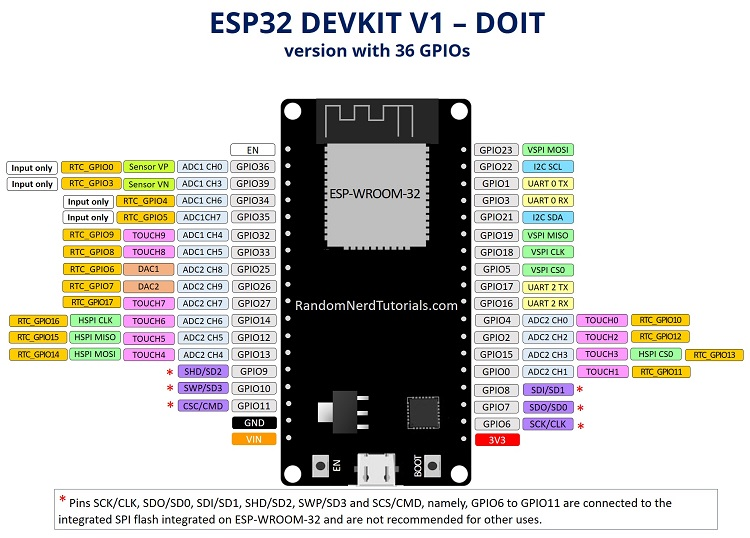
• WiFi.h and BlynkSimpleEsp32.h for Blynk and WiFi integration.

• ACS712.h and ZMPT101B.h for sensor readings.

• Adafruit\_GFX.h and Adafruit\_SSD1306.h for OLED display.

**Overview of Components**

**ESP32 Overview**



The **ESP32** is a versatile, low-cost microcontroller developed by **Espressif Systems**, widely used in IoT, robotics, and embedded system projects. Known for its powerful processing capabilities, integrated Wi-Fi, and Bluetooth, the ESP32 is a go-to choice for building smart devices and connected systems.

**Features of ESP32**

1. **Integrated Wi-Fi and Bluetooth**:

• Dual-mode Bluetooth (Classic and BLE) for connectivity.

• IEEE 802.11 b/g/n Wi-Fi support with WPA/WPA2 security.

2. **Processing Power**:

• Dual-core 32-bit Xtensa LX6 microprocessor.

• Clock speed: Up to 240 MHz.

• Ultra-low power co-processor for sleep modes.

3. **Memory**:

• 520 KB SRAM.

• 4MB of Flash memory (in most development boards).

4. **GPIO Pins**:

• 34 GPIO pins for interfacing sensors, actuators, and peripherals.

• Dedicated pins for ADC, DAC, PWM, I2C, SPI, UART, and I2S.

5. **Analog and Digital I/O**:

• 18 ADC pins (12-bit resolution) for reading analog signals.

• 2 DAC channels for analog output.

6. **Power Management**:

• Ultra-low power consumption modes for battery-operated projects.

• Supports deep sleep, light sleep, and modem sleep.

7. **Advanced Features**:

• Integrated Hall-effect sensor.

• Integrated temperature sensor.

• Hardware encryption (AES, RSA, SHA).

• Real-Time Clock (RTC).

**Technical Specifications**

|  |  |
| --- | --- |
| **Feature** | **Specification** |
| Microprocessor | Xtensa Dual-Core LX6 |
| Clock Speed | Up to 240 MHz |
| Wi-Fi Standard | IEEE 802.11 b/g/n |
| Bluetooth Version | Dual-mode (Classic and BLE) |
| GPIO Pins | 34 |
| ADC Channels | 18 (12-bit resolution) |
| DAC Channels | 2 |
| PWM Channels | 16 |
| Flash Memory | 4 MB (varies by board) |
| SRAM | 520 KB |
| Operating Voltage | 3.3V |
| Communication Interfaces | UART, SPI, I2C, CAN, I2S, Ethernet, SDIO |
| Temperature Range | -40°C to +125°C |

**Pin Diagram**

The ESP32 offers flexible pin configurations for a variety of applications. Below is a high-level summary of the key pin functions:

1. **Power Pins**:

• Vin: External power input (5V).

• 3V3: Regulated 3.3V output.

• GND: Ground pins.

2. **Analog Pins**:

• ADC: GPIO32 to GPIO39.

• DAC: GPIO25 and GPIO26.

3. **Digital Pins**:

• GPIO0 to GPIO34 (some pins are input-only).

4. **Communication Pins**:

• UART: GPIO1 (TX), GPIO3 (RX), and others.

• SPI: GPIO12 (MISO), GPIO13 (MOSI), GPIO14 (SCK), GPIO15 (CS).

• I2C: GPIO21 (SDA), GPIO22 (SCL).

5. **Special Pins**:

• Boot button: GPIO0.

• Enable button: EN pin for resetting the board.

**Applications**

The ESP32 is ideal for a wide range of applications, including:

1. **IoT Devices**:

• Smart home systems.

• Environmental monitoring.

2. **Wearables**:

• Health trackers.

• BLE-based devices.

3. **Robotics**:

• Sensor-based automation.

• Wireless remote controls.

4. **Industrial Systems**:

• Data logging.

• Process monitoring.

5. **DIY Projects**:

• Weather stations.

• Home automation systems.

**Getting Started with ESP32**

Setting up the Arduino IDE for ESP32 on a macOS system involves installing the necessary tools and configuring the IDE to recognize ESP32 boards. Follow these steps:

**1. Install Arduino IDE**

• Download the latest version of the Arduino IDE for macOS from the [official Arduino website](https://www.arduino.cc/en/software).

• Install the IDE by dragging it into the Applications folder.

**2. Install ESP32 Board Manager URL**

1. Open the Arduino IDE.

2. Go to **Arduino > Preferences** from the menu bar.

3. In the “Additional Boards Manager URLs” field, add the ESP32 boards URL:

https:*//raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package\_esp32\_index.json*

4. If there are multiple URLs, separate them with commas.

**3. Install the ESP32 Board Package**

1. Go to **Tools > Board > Boards Manager**.

2. Search for “ESP32” in the search bar.

3. Select the **esp32 by Espressif Systems** and click **Install**.

4. Wait for the installation to complete.

**4. Select the ESP32 Board**

1. Go to **Tools > Board** and scroll down to the **ESP32 Arduino** section.

2. Select the specific ESP32 board you’re using, e.g., “DOIT ESP32 DEVKIT V1 or ESP32 Dev Module.”

**5. Configure Port and Settings**

1. Connect your ESP32 to the Mac using a USB cable.

2. Go to **Tools > Port** and select the correct serial port (e.g., /dev/cu.SLAB\_USBtoUART or /dev/cu.usbserial-58B90…).

3. Verify the settings under **Tools**:

• Flash Frequency: 80MHz

• Upload Speed: 115200

• Partition Scheme: Default

**6. Install USB Drivers (if required)**

If your ESP32 is not recognized, you may need to install drivers:

• For CP210x USB-to-UART Bridge: Download from the [Silicon Labs website](https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers).

• For CH340 USB-to-Serial chips: Download the driver from the [WCH website](http://www.wch.cn/downloads/CH341SER_MAC_ZIP.html).

After installing the driver, reconnect your ESP32 and check if the port is detected.

**7. Test Your Setup**

1. Open a basic example sketch:

• Go to **File > Examples > WiFi > WiFiScan**.

2. Click **Verify** (checkmark icon) to compile the code.

3. Click **Upload** (arrow icon) to upload the code to your ESP32.

4. Open the Serial Monitor (**Tools > Serial Monitor**) to see the output.

**Troubleshooting**

• **Port Not Found**: Ensure the USB cable supports data transfer and that the drivers are installed.

• **Upload Error**: Press and hold the “Boot” button on the ESP32 while uploading.

Once the setup is complete, you can start developing ESP32 projects using the Arduino IDE!

**Checking ESP\_32**

Check esp 32 if it works fine or not

Sample code for checking esp 32:

#include "WiFi.h"

void setup() {

Serial.begin(115200); // Start Serial communication

delay(1000);

Serial.println("Starting Wi-Fi scan...");

WiFi.mode(WIFI\_STA); // Set the ESP32 as a Wi-Fi station

WiFi.disconnect();

delay(100);

}

void loop() {

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

int numNetworks = WiFi.scanNetworks();

if (numNetworks == 0) {

Serial.println("No networks found.");

} else {

Serial.println("Networks found:");

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

Serial.print(i + 1);

Serial.print(": ");

Serial.print(WiFi.SSID(i));

Serial.print(" (");

Serial.print(WiFi.RSSI(i));

Serial.println(" dBm)");

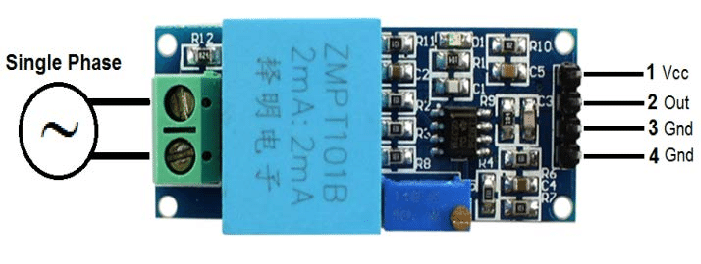
}

}

delay(10000); // Wait 10 seconds before scanning again

}

**ZMPT101B Overview**



The **ZMPT101B** is a precision voltage sensor module that is widely used for AC voltage monitoring in applications such as power measurement, energy monitoring systems, and protection devices. It is based on the **ZMPT101B voltage transformer**, which is designed for measuring AC voltage with high accuracy.

**Key Features of ZMPT101B Module**

1. **Voltage Transformer-Based**:

• Uses a ZMPT101B voltage transformer to step down AC voltage for measurement.

2. **Precision Signal Conditioning Circuit**:

• Includes operational amplifiers for scaling and conditioning the output signal.

• Ensures stable and linear output, even with small input voltages.

3. **Adjustable Gain**:

• The onboard potentiometer allows you to adjust the module’s sensitivity (gain), enabling accurate calibration for different voltage ranges.

4. **Compact and Easy to Use**:

• Small form factor and pre-soldered components make it easy to integrate with microcontrollers like Arduino, ESP32, or Raspberry Pi.

5. **Output**:

• Produces an analog signal proportional to the input AC voltage, which can be read by an ADC pin on a microcontroller.

**Technical Specifications**

|  |  |
| --- | --- |
| **­ Parameter** | **Value** |
| Input Voltage Range | Up to 250V AC |
| Output Voltage Rang | 0-5V (adjustable) |
| Transformer Model | ZMPT101B |
| Power Supply | 5V DC |
| Accuracy | High |
| Dimensions | Compact module size |

**Working Principle**

1. **Voltage Transformation**:

• The ZMPT101B transformer steps down the input AC voltage to a smaller, proportional voltage.

2. **Signal Conditioning**:

• The stepped-down voltage passes through an operational amplifier circuit for scaling and offset adjustment.

• The module outputs an analog voltage signal that oscillates around a DC offset (e.g., 2.5V for a 5V system).

3. **Reading the Output**:

• The microcontroller’s ADC reads the analog output, and software is used to calculate the original AC voltage using calibration factors.

**Applications**

1. **Energy Monitoring Systems**:

• Used for monitoring and analyzing AC voltage in residential or industrial setups.

2. **Power Measurement**:

• Paired with a current sensor (like CT or Hall Effect sensors), it can measure power and energy consumption.

3. **Overvoltage/Undervoltage Protection**:

• Helps detect abnormal voltage conditions in electrical systems.

4. **IoT Energy Projects**:

• Often integrated with microcontrollers for real-time voltage monitoring over IoT platforms.

**Advantages**

• High accuracy and stability.

• Adjustable sensitivity for various applications.

• Compact design makes it easy to use.

• Compatible with a wide range of microcontrollers.

Here’s the updated and merged ESP32 code and explanation for using the **ZMPT101B Voltage Sensor** with the ESP32:

**Code for ESP32 with ZMPT101B**

#include <ZMPT101B.h>

#define ZMPT101B\_PIN 34 // ADC pin connected to ZMPT101B output

#define SENSITIVITY 500.0f // Calibration factor for your ZMPT101B module

#define FREQUENCY 60.0 // Frequency of AC source (60 Hz or 50 Hz)

ZMPT101B voltageSensor(ZMPT101B\_PIN, FREQUENCY);

void setup() {

Serial.begin(115200);

// Initialize the ZMPT101B sensor

voltageSensor.setSensitivity(SENSITIVITY);

Serial.println("ZMPT101B sensor initialized.");

}

void loop() {

// Read RMS voltage from the sensor

float rmsVoltage = voltageSensor.getRmsVoltage();

// Print voltage reading with context

Serial.print("Measured AC Voltage (RMS): ");

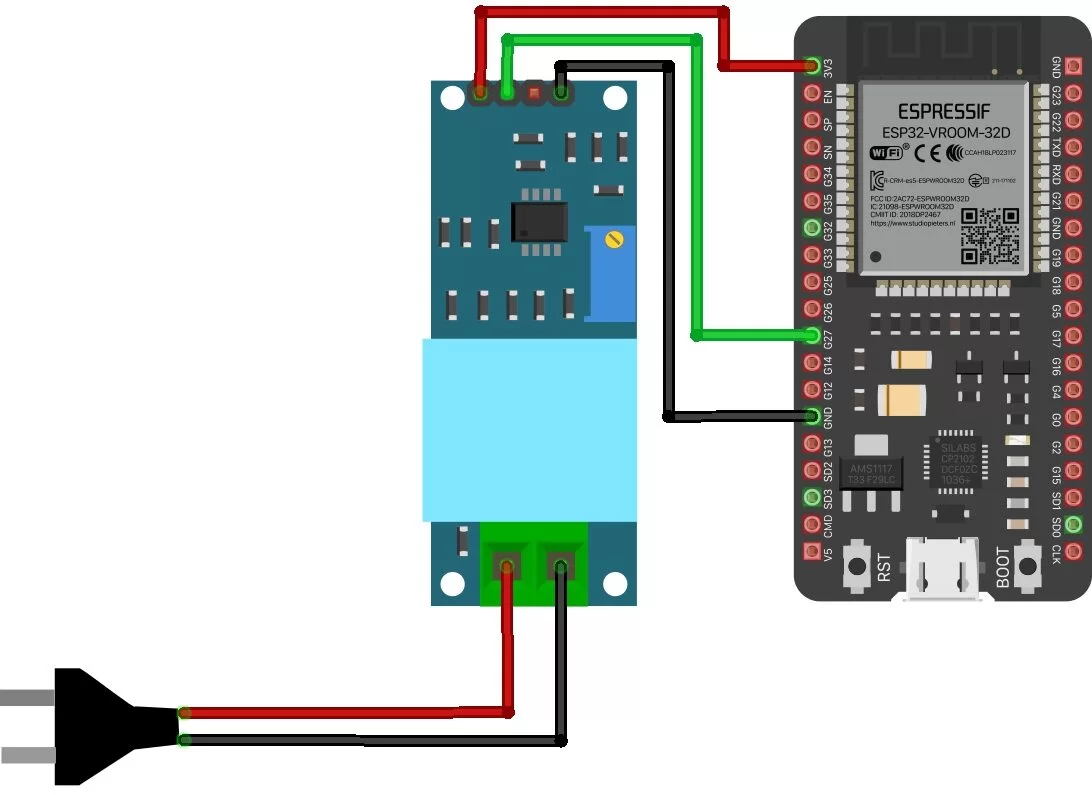
Serial.print(rmsVoltage);

Serial.println(" V");

// Delay for stability

delay(500);

}

****

**Explanation**

1. **Pin Selection**:

• The ZMPT101B’s analog output is connected to **GPIO34**, which is one of the ESP32’s ADC1 pins. You can change this to another ADC pin as needed.

2. **ADC Resolution**:

• The ESP32 ADC uses a **12-bit resolution** by default, giving raw values from 0 to 4095. You can explicitly set this using analogReadResolution(12).

3. **Reference Voltage**:

• The ESP32’s ADC reference voltage is **3.3V**, so the voltage conversion formula is adjusted to:

4. **Attenuation**:

• ESP32 ADC pins have adjustable attenuation levels to handle input voltages higher than the default **1.1V** range. The code uses ADC\_11db, which extends the range to **0–3.9V**, suitable for a **3.3V system**.

• Attenuation options:

• ADC\_0db: 0–1.1V

• ADC\_2\_5db: 0–1.5V

• ADC\_6db: 0–2.2V

• ADC\_11db: 0–3.9V

5. **Delay**:

• A **500ms delay** between readings ensures that data isn’t overwhelmed on the Serial Monitor.

**Optional Enhancements for Better Accuracy**

• **Calibration**:

To improve accuracy, compare the raw ADC values to known input voltages using a multimeter, and apply a scaling factor in the conversion formula.

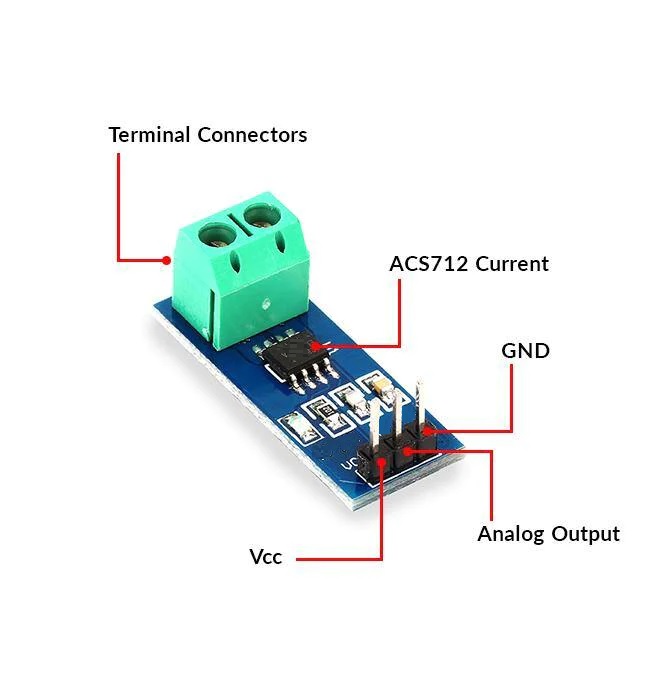
• **Voltage Filtering**:

For AC voltage measurements, use averaging or root-mean-square (RMS) calculations to stabilize readings.

• **Signal Conditioning**:

Add a capacitor across the sensor output to smooth any noise in the ADC readings.

This approach ensures accurate voltage measurements with the ZMPT101B sensor using the ESP32, tailored to its ADC capabilities.

**ACS712 30A Overview with ESP32 Code**

The **ACS712 30A** is a **Hall Effect-Based Current Sensor** capable of measuring both AC and DC currents up to ±30A. It is widely used in various applications requiring current measurement due to its accuracy, compact size, and easy interfacing with microcontrollers.

**Key Features**

1. **Current Measurement Range**:

• ±30A (DC or AC).

2. **Output**:

• Analog voltage proportional to the current being measured.

3. **Accuracy**:

• Typical sensitivity: **66mV per Amp** for the 30A model.

4. **Input Voltage**:

• Operates at **5V DC**.

5. **Isolation**:

• Provides electrical isolation between the current-carrying conductor and the measurement circuit (via Hall Effect sensing).

6. **Compact and Easy to Interface**:

• 3-pin interface: **Vcc, GND, and OUT** (analog output).

7. **High Bandwidth**:

• Bandwidth: **80kHz** (can be reduced by adding an external capacitor to the filter pin for noise reduction).

**Pin Description**

1. **Vcc**: Connects to the 5V supply.

2. **GND**: Connects to the ground of the power supply.

3. **OUT**: Outputs the analog voltage corresponding to the current.

**How It Works**

1. **Hall Effect Sensing**:

• The ACS712 measures the magnetic field generated by the current passing through the conductor.

• The magnetic field is converted into a voltage proportional to the current using the Hall Effect principle.

2. **Output Voltage**:

• At **0A**, the output voltage is at the middle of the supply voltage (approximately **2.5V** for a 5V supply).

• For a 30A current:

• Positive current increases the output above 2.5V.

• Negative current decreases the output below 2.5V.

3. **Sensitivity**:

• The sensitivity is **66mV/A** for the ACS712 30A module.

• Example:

• At 10A, the output voltage would be:

**Applications**

1. **Overcurrent Protection**.

2. **Battery Management Systems (BMS)**.

3. **Motor Current Sensing**.

4. **Inverters and Power Supplies**.

5. **AC/DC Power Monitoring**.

**Advantages**

• Compact and cost-effective.

• Measures both AC and DC currents.

• Electrically isolated measurement.

• Fast response time and high bandwidth.

**Limitations**

1. **Noise Sensitivity**:

• High-frequency noise can affect readings; use filtering for stability.

2. **Accuracy**:

• Non-linearity and temperature drift may require calibration for precise applications.

3. **Power Supply**:

• Requires a stable 5V supply for consistent output.

**Interfacing with Microcontrollers**

The ACS712 is simple to connect to microcontrollers like Arduino or ESP32. Its output voltage can be read using an ADC (Analog-to-Digital Converter) pin. The current can then be calculated using the sensitivity factor and the formula:

**ESP32 Code**

The **ACS712 30A** current sensor uses the Hall Effect to measure AC/DC currents up to ±30A. It outputs an analog voltage proportional to the measured current, which can be read by the ESP32’s ADC pin. Below is a code snippet and a short explanation of its working.

**ESP32 Code for ACS712 30A**

#include "ACS712.h"

ACS712 ACS(34, 3.3, 4095, 125); // Parameters: analog pin, Vref, ADC resolution, sensor sensitivity

int calibration\_factor = 120;

void setup() {

Serial.begin(115200);

Serial.println("ACS712 Current Measurement Initialized");

}

void loop() {

float current = readCurrent();

// Print the measured current

Serial.print("Current: ");

Serial.print(current);

Serial.println(" mA");

delay(1000); // Delay for readability

}

float readCurrent() {

float average = 0;

// Take multiple readings to average out noise

for (int i = 0; i < 100; i++) {

average += ACS.mA\_AC();

}

// Calculate the average current and adjust for calibration

float mA = (abs(average / 100.0) - calibration\_factor);

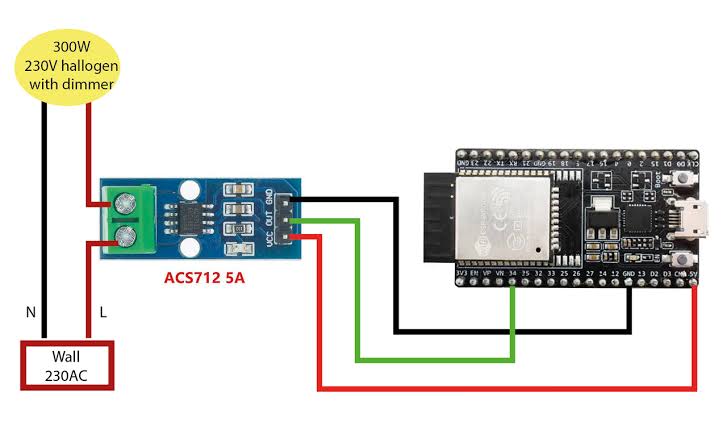
// Handle noise below a threshold

if (mA <= 5) mA = 0;

return mA; // Return the measured current

}

}



**Explanation of Code**

1. **Pin Definition**:

• ACS712\_PIN is set to GPIO34, an ADC-capable pin on the ESP32.

2. **Voltage and ADC Setup**:

• VREF: ESP32’s analog input range is 0 to 3.3V.

• ADC\_RES: The ESP32 ADC has a 12-bit resolution (0 to 4095).

3. **Calculations**:

• The analog input is read using analogRead().

• The raw ADC value is converted to voltage using:

• The current is calculated from the voltage:

• The offset (VREF/2) accounts for the zero-current output at 2.5V.

4. **Current Output**:

• The calculated current is displayed on the Serial Monitor in Amperes with two decimal places.

**Applications**

• Measure AC/DC currents for loads such as motors, inverters, or appliances.

• Monitor battery charging/discharging currents.

• Implement overcurrent protection in systems.

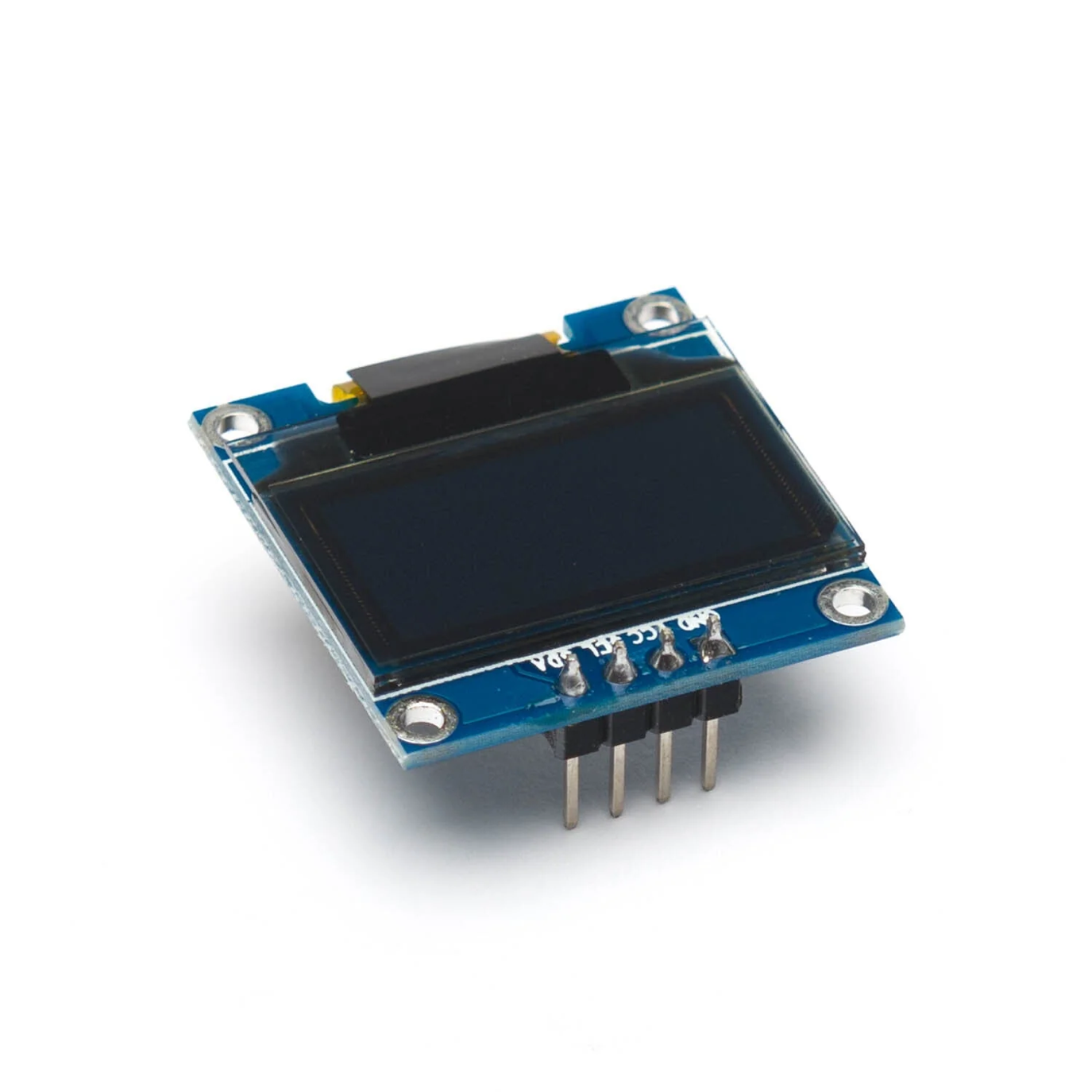
**Advantages of Using ESP32**

• Multiple ADC pins for simultaneous sensor readings.

• Wi-Fi or Bluetooth capability for remote monitoring.

• Higher ADC resolution compared to Arduino Uno.

This setup efficiently monitors current with the ACS712 using the ESP32.

**0.96” OLED Display Overview**

A **0.96-inch OLED (Organic Light Emitting Diode)** display is a compact, low-power display module commonly used in embedded systems. It uses **organic compounds** to emit light, eliminating the need for a backlight, which makes it power-efficient and provides high contrast.

**Key Features**

• **Display Resolution**: Typically **128x64 pixels**.

• **Interface**: Supports I²C or SPI communication for easy interfacing.

• **Power Consumption**: Very low due to self-illuminating pixels.

• **Color**: Commonly available in monochrome (white, blue, or yellow) or multicolor variants.

• **Viewing Angle**: Excellent, nearly 180°.

• **Driver IC**: Most 0.96” OLEDs are based on the **SSD1306** driver chip.

**Applications**

• Status displays for IoT devices.

• Portable gadgets, such as wearables.

• Real-time system monitoring in embedded systems.

• User interfaces for compact devices.

This display’s compact size and high efficiency make it ideal for small devices requiring a visual interface.

**Code for ESP32 with 0.96” OLED:**

#include <Wire.h>

#include <Adafruit\_GFX.h>

#include <Adafruit\_SSD1306.h>

#define SCREEN\_WIDTH 128 // OLED display width, in pixels

#define SCREEN\_HEIGHT 64 // OLED display height, in pixels

#define OLED\_RESET -1 // Reset pin # (or -1 if sharing Arduino reset pin)

Adafruit\_SSD1306 display(SCREEN\_WIDTH, SCREEN\_HEIGHT, &Wire, OLED\_RESET);

void setup() {

// Start serial communication for debugging purposes

Serial.begin(115200);

// Initialize the OLED display

if (!display.begin(SSD1306\_SWITCHCAPVCC, 0x3C)) { // 0x3C is the I2C address of the OLED

Serial.println(F("SSD1306 allocation failed"));

for (;;); // Don't proceed, loop forever

}

// Clear the buffer

display.clearDisplay();

// Print the initialization message

display.setTextSize(1); // Normal 1:1 pixel scale

display.setTextColor(SSD1306\_WHITE); // Draw white text

display.setCursor(0, 0); // Start at top-left corner

display.println(F("OLED Display Test"));

display.display(); // Show the message on the display

delay(2000); // Wait for 2 seconds

}

void loop() {

static int counter = 0;

// Clear the display

display.clearDisplay();

// Display a counter value

display.setTextSize(2);

display.setTextColor(SSD1306\_WHITE);

display.setCursor(0, 0);

display.print("Counter: ");

display.println(counter);

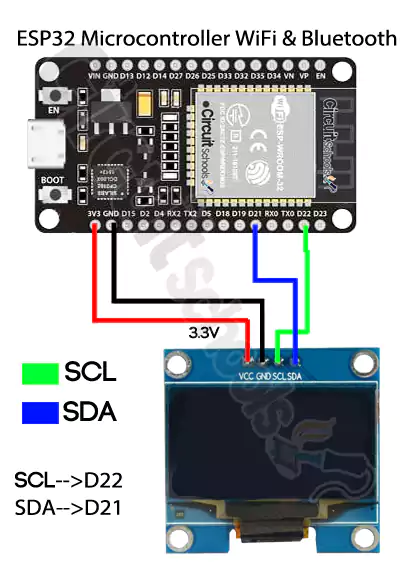
display.display(); // Show updated content on OLED

// Debug output to the serial monitor

Serial.print("Counter: ");

Serial.println(counter);

counter++;

 delay(1000); // Update every second

}

**Explanation of the Code:**

1. **Include Libraries**:

• Wire.h: This library is used for I2C communication between the ESP32 and the OLED.

• Adafruit\_GFX.h: This library provides graphics functions, such as drawing text, shapes, and images.

• Adafruit\_SSD1306.h: This library is specifically for controlling SSD1306-based OLED displays.

2. **Display Initialization**:

• SCREEN\_WIDTH and SCREEN\_HEIGHT define the dimensions of the OLED display (128x64 pixels).

• Adafruit\_SSD1306 display(SCREEN\_WIDTH, SCREEN\_HEIGHT, &Wire, OLED\_RESET) initializes the display object with the screen dimensions and I2C communication.

• display.begin(SSD1306\_I2C\_ADDRESS, OLED\_RESET) initializes the OLED. The I2C address is typically 0x3C or 0x3D for most OLEDs.

3. **Display Text**:

• display.clearDisplay() clears the screen.

• display.setTextSize(1) sets the text size (1 is the default).

• display.setTextColor(SSD1306\_WHITE) sets the text color (in white for monochrome displays).

• display.setCursor(0,0) sets the starting point of the text (top-left corner).

• display.print(F("Hello, ESP32!")) displays the message on the screen.

4. **Loop**: In this simple example, the loop function is empty because the text doesn’t need to change or update continuously.

**Wiring the OLED to ESP32:**

• **VCC** to 3.3V

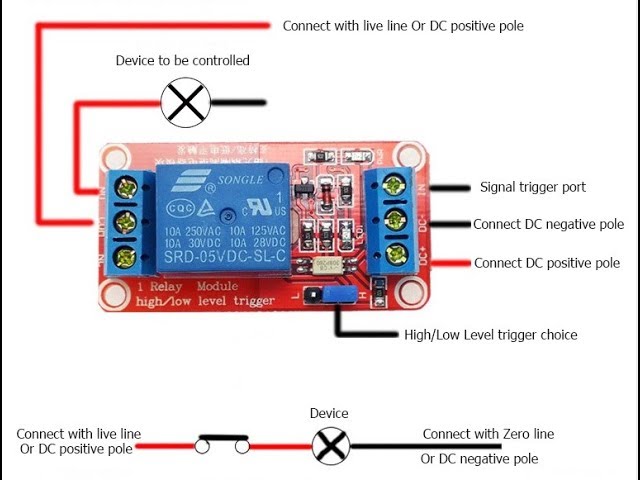
• **GND** to GND

• **SDA** to GPIO21 (default for I2C SDA on ESP32)

• **SCL** to GPIO22 (default for I2C SCL on ESP32)

This code will display “Hello, ESP32!” on the OLED screen once the ESP32 is powered on.

**Overview of a 5V 1-Channel Relay Module:**



A **5V 1-channel relay module** is a component that allows you to control high-voltage devices (e.g., 110V/220V AC appliances) using a low-voltage signal (such as from a microcontroller like the **ESP32**, **Arduino**, etc.). The relay acts as an electrically operated switch that can control the current flow to a device, making it ideal for controlling high-power devices like motors, lights, fans, etc.

**Components of a 5V 1-Channel Relay Module:**

1. **Relay**:

• A mechanical switch that is used to control the flow of electricity to the connected device.

• It has a **NO (Normally Open)**, **NC (Normally Closed)**, and **COM (Common)** contact.

• When the relay coil is energized, the NO (Normally Open) and COM (Common) contacts close, allowing current to pass through to the device.

2. **Optocoupler (for isolation)**:

• This component provides electrical isolation between the low-voltage microcontroller and the high-voltage side of the relay. It helps protect the microcontroller from voltage spikes or surges.

• It transfers signals from the low-voltage side (ESP32/Arduino) to the high-voltage relay circuit.

3. **Transistor**:

• A transistor is often used as a switch to activate the relay. It allows the low-power microcontroller to control the higher current required to activate the relay coil.

4. **Diode (Flyback Diode)**:

• A diode is connected in parallel with the relay coil to protect against voltage spikes (inductive kickback) that could damage the components when the relay is switched off.

5. **Jumper pins or screw terminals**:

• These are used to connect the relay module to the microcontroller and the external device being controlled.

**Pinout of a 5V 1-Channel Relay Module:**

1. **VCC**: Connects to 5V power supply.

2. **GND**: Ground (GND) connection.

3. **IN**: Control pin (input signal from a microcontroller like ESP32 or Arduino).

4. **NO (Normally Open)**: A terminal for connecting the load when the relay is activated.

5. **COM (Common)**: The common terminal connected to either NO or NC.

6. **NC (Normally Closed)**: This is connected to the device when the relay is deactivated.

**How It Works:**

• When a low-voltage signal (typically 3.3V or 5V) is applied to the **IN** pin of the relay module, the **transistor** is turned on, activating the **relay**.

• This closes the **NO** and **COM** terminals, allowing current to flow to the high-voltage device connected to those terminals.

• The device is turned on when the relay is activated.

• When the signal to the **IN** pin is turned off, the relay deactivates, and the **NO** terminal opens, disconnecting the device from the power supply.

**Applications of 5V 1-Channel Relay Module:**

1. **Home Automation**: Use to control appliances like lights, fans, or other household electronics via microcontrollers.

2. **Automation Projects**: Automatically switch on/off devices based on sensor readings or timed events.

3. **IoT Projects**: Control devices remotely using Wi-Fi-enabled microcontrollers (e.g., ESP32, ESP8266) with mobile apps or web interfaces.

4. **Testing and Prototyping**: Control high-voltage circuits safely in lab experiments or during prototyping.

**Wiring the Relay Module to ESP32/Arduino:**

• **VCC** -> 5V (Power Supply)

• **GND** -> GND (Power Supply)

• **IN** -> GPIO pin of ESP32 (e.g., GPIO23) or Arduino (e.g., pin 8)

• **NO (Normally Open)** -> Device (e.g., a light)

• **COM (Common)** -> Power supply of the device (e.g., AC mains or DC power supply)

**Important Considerations:**

• **Relay Rating**: Ensure the relay you use matches the voltage and current requirements of the device you intend to control.

• **Safety**: When dealing with high-voltage devices, always take necessary precautions to avoid electric shock or damage. If you are new to working with high-voltage circuits, it is advisable to work under the supervision of someone experienced or use proper isolation.

• **External Power Source**: Some relay modules may require an external power supply (5V or 12V), especially when driving larger devices.

This relay module is a simple yet powerful tool for interfacing low-voltage controllers with high-voltage loads, making it essential for many automation projects.

**ESP32 Code:**

#define RELAY\_PIN 23  *// Connect the IN pin of the relay to ESP32 GPIO23*

void setup() {

  Serial.begin(115200);  *// Start serial communication for debugging*

  pinMode(RELAY\_PIN, OUTPUT);  *// Set GPIO23 as output to control the relay*

  digitalWrite(RELAY\_PIN, LOW);  *// Ensure the relay is initially off*

}

void loop() {

  Serial.println("Turning ON Relay...");

  digitalWrite(RELAY\_PIN, HIGH);  *// Activate the relay (turn ON device)*

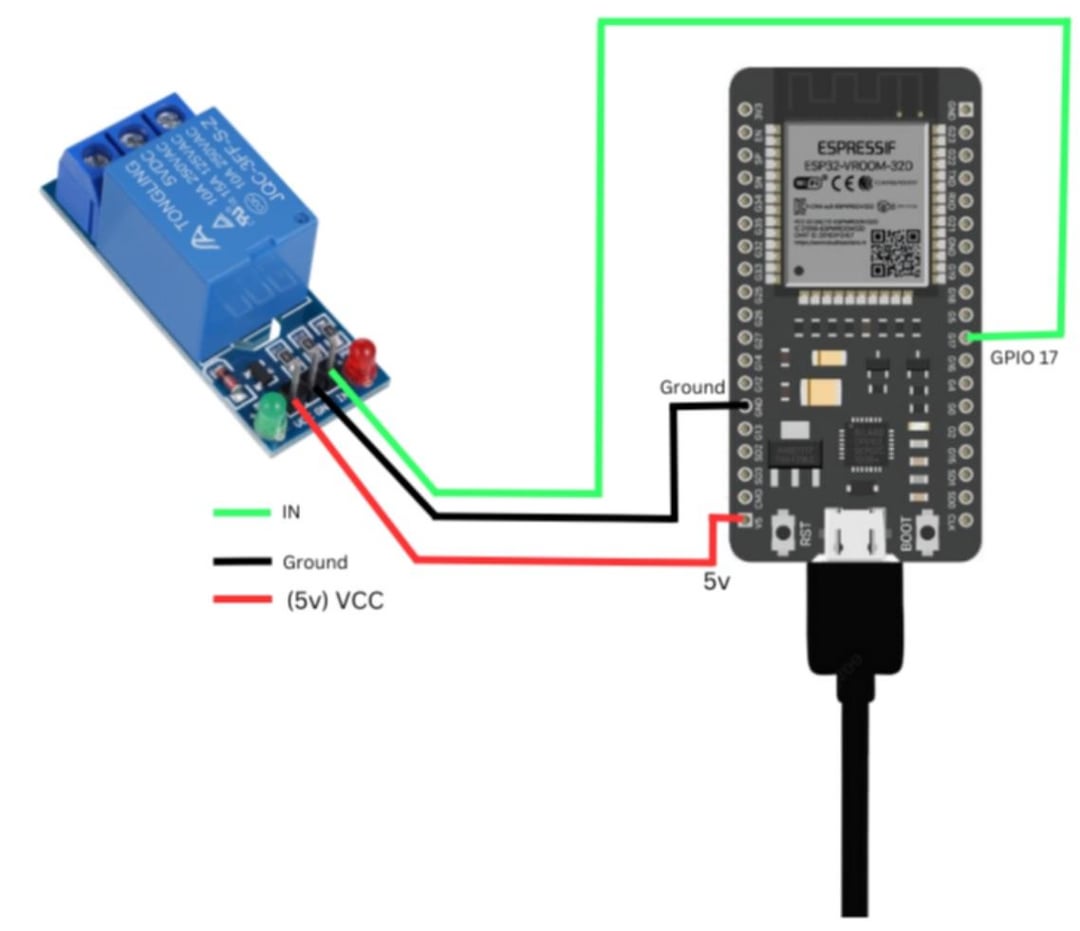
  delay(2000);  *// Wait for 2 seconds*

  Serial.println("Turning OFF Relay...");

  digitalWrite(RELAY\_PIN, LOW);  *// Deactivate the relay (turn OFF device)*

  delay(2000);  *// Wait for 2 seconds*

}



**Explanation:**

1. **Relay Pin Setup**:

• #define RELAY\_PIN 23: The relay control pin is connected to **GPIO23** of the ESP32.

2. **Setup Function**:

• pinMode(RELAY\_PIN, OUTPUT);: The relay pin is set as an **OUTPUT** pin to control the relay module.

• digitalWrite(RELAY\_PIN, LOW);: Initially, we ensure that the relay is **off** by setting the pin to LOW.

3. **Loop Function**:

• The relay is turned **on** by setting the pin to HIGH (this activates the relay and allows current to flow to the device).

• After 2 seconds (delay(2000)), the relay is turned **off** by setting the pin to LOW.

• This on/off cycle repeats every 2 seconds.

**Wiring:**

• **VCC** of relay module -> **5V** on the ESP32 (if using a separate 5V power supply, connect the GND to ESP32 GND).

• **GND** of relay module -> **GND** on the ESP32.

• **IN** of relay module -> **GPIO23** (or any other GPIO you choose).

• **COM (Common)** and **NO (Normally Open)** should be connected to the device you wish to control.

**Notes:**

• When the **IN** pin is HIGH, the relay will activate and allow current to flow through the device. When the **IN** pin is LOW, the relay will deactivate, cutting off the current to the device.

• Ensure that the relay module and the connected device are rated for the power you plan to switch (AC or DC).

This code can be easily modified to control the relay based on other conditions or inputs.

**How It Works**

**Core Functionalities:**

1. **Voltage Measurement**:

• The ZMPT101B sensor reads the RMS voltage from the AC source.

• Sensitivity is calibrated using the setSensitivity() method.

2. **Current Measurement**:

• The ACS712 sensor reads the AC current flowing through the load.

• Noise is filtered, and a calibration factor is applied for accuracy.

3. **Power & Energy Calculation**:

• Power (P) = Voltage (V) × Current (I).

• Energy (E) = Power × Time.

• Cost is calculated using a predefined **per-unit cost** (12.67 TK).

4. **Relay Control**:

• A relay is toggled using Virtual Pin V4 on the Blynk app.

• ON: Activates relay to power the load.

• OFF: Deactivates relay.

5. **OLED Display**:

• Displays Voltage, Current, Power, Energy, and Cost in real-time.

6. **Blynk IoT Integration**:

• Sends measurements to Blynk app on Virtual Pins:

• V0: Voltage

• V1: Current

• V5: Power

• V6: Energy

• V3: Cost

• Controls relay via V4.

**Pin Configuration**

|  |  |  |
| --- | --- | --- |
| Component | Pin | Description |
| ACS712 (Current Sensor) | GPIO 34 | Reads current values. |
| ZMPT101B (Voltage Sensor) | GPIO 35 | Reads voltage values. |
| OLED Display (SSD1306) | SDA: GPIO 21 | I2C Data line. |
|  | SCL: GPIO 22 | I2C Clock line. |
| Relay Module | GPIO 2 | Toggles the relay. |

**Setup Instructions**

1. **Hardware Setup**:

• Connect the ACS712, ZMPT101B, OLED Display, and Relay to the ESP32 as per the pin configuration.

2. **Install Libraries**:

• Install required libraries via the Arduino Library Manager:

• Blynk

• Adafruit GFX

• Adafruit SSD1306

• ACS712

• ZMPT101B

3. **Code Configuration**:

• Replace the **WiFi credentials**:

Below is an explanation of the code for the project, broken down line by line or in small sections for clarity:

**Header Information and Library Inclusion**

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

*This is a simple demo of sending and receiving some data.*

*Be sure to check out other examples!*

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

• This is a comment block explaining that the code is a demo for sending and receiving data, likely through the Blynk platform.

#define BLYNK\_TEMPLATE\_ID           "TMPL68ptrSiHP"

#define BLYNK\_TEMPLATE\_NAME         "Quickstart Template"

#define BLYNK\_AUTH\_TOKEN            "msQRzTZLjJ88\_\_UYisl\_ZtOga2YivDgD"

• Defines constants for the Blynk template ID, template name, and the authentication token used to connect the device to the Blynk server.

#define BLYNK\_PRINT Serial

• Enables debug prints for Blynk. Data and logs will be output to the Serial console.

#include <WiFi.h>

#include <WiFiClient.h>

#include <BlynkSimpleEsp32.h>

#include "ACS712.h"

#include <ZMPT101B.h>

#include <Wire.h>

#include <Adafruit\_GFX.h>

#include <Adafruit\_SSD1306.h>

• Includes the necessary libraries:

• WiFi.h: For Wi-Fi connectivity.

• WiFiClient.h: Provides a client for connecting to servers.

• BlynkSimpleEsp32.h: Connects ESP32 with Blynk.

• ACS712.h: Handles the ACS712 current sensor.

• ZMPT101B.h: Manages the ZMPT101B voltage sensor.

• Wire.h: I2C communication for the OLED display.

• Adafruit\_GFX.h and Adafruit\_SSD1306.h: Libraries for handling the OLED display.

**OLED Display Setup**

#define SCREEN\_WIDTH 128

#define SCREEN\_HEIGHT 64

#define OLED\_RESET -1

Adafruit\_SSD1306 display(SCREEN\_WIDTH, SCREEN\_HEIGHT, &Wire, OLED\_RESET);

• Defines the OLED screen dimensions (128x64) and creates an Adafruit\_SSD1306 object to control the display.

**ACS712 (Current Sensor) Setup**

#define ACS712\_PIN 34

ACS712 ACS(ACS712\_PIN, 3.3, 4095, 125);

int calibration\_factor = 120;

• Defines the GPIO pin connected to the ACS712 sensor (34).

• Configures the ACS712 object with:

• The connected pin.

• Reference voltage (3.3V for ESP32).

• ADC resolution (12-bit = 4095).

• Sensor sensitivity (125 mV/A for a specific ACS712 model).

• Sets a calibration\_factor to adjust readings.

**ZMPT101B (Voltage Sensor) Setup**

#define ZMPT101B\_PIN 35

#define SENSITIVITY 500.0f

#define FREQUENCY 60.0

ZMPT101B voltageSensor(ZMPT101B\_PIN, FREQUENCY);

• Defines:

• GPIO pin connected to the ZMPT101B sensor (35).

• Sensor sensitivity for calibration.

• Frequency of the AC source (60 Hz).

• Creates a ZMPT101B object to manage the voltage sensor.

**Relay Setup**

#define RELAY\_PIN 2

bool relayState = false;

• Defines:

• GPIO pin connected to the relay module (2).

• A boolean (relayState) to store the current relay state.

**Wi-Fi and Blynk Configuration**

char ssid[] = "322";

char pass[] = "murtazafy";

BlynkTimer timer;

• Defines Wi-Fi credentials.

• Creates a BlynkTimer object for timed events.

**Power and Energy Variables**

float power = 0.0;

float energy = 0.0;

unsigned long lastUpdateTime = 0;

• Initializes variables for power (watts) and energy (kWh).

• lastUpdateTime tracks the last update for energy calculations.

**Power and Energy Calculation**

void calculatePowerAndEnergy(float current, float voltage) {

  power = voltage \* current / 1000.0;

  float K\_power = (power / 1000.0);

  float units = K\_power \* 1;

  float per\_unit\_cost = (units \* 12.67);

  Blynk.virtualWrite(V5, power);

  Blynk.virtualWrite(V6, units);

  Blynk.virtualWrite(V3, per\_unit\_cost);

  Serial.print("Power: ");

  Serial.println(power);

  display.setCursor(0, 35);

  display.print("Power: ");

  display.print(power);

  display.println(" W");

  display.display();

}

• Calculates power (P = V \* I) and energy (units = P \* time).

• Sends data to Blynk virtual pins and displays it on the OLED and Serial Monitor.

**Relay Control**

BLYNK\_WRITE(V4) {

  int buttonState = param.asInt();

  if (buttonState == 1) {

    digitalWrite(RELAY\_PIN, LOW);

    relayState = true;

  } else {

    digitalWrite(RELAY\_PIN, HIGH);

    relayState = false;

  }

}

• Controls the relay based on a button state received from Blynk.

**Timer and Sensor Reading Functions**

void myTimerEvent() {

  Blynk.virtualWrite(V2, millis() / 1000);

}

float readCurrent() {

  float average = 0.0;

  for (int i = 0; i < 100; i++) {

    average += ACS.mA\_AC();

  }

  return (abs(average / 100.0) - calibration\_factor);

}

float readVoltage() {

  float average\_voltage = 0.0;

  for (int i = 0; i < 100; i++) {

    average\_voltage += voltageSensor.getRmsVoltage();

  }

  return average\_voltage / 100.0;

}

• myTimerEvent: Sends uptime to Blynk every second.

• readCurrent: Averages 100 readings to reduce noise.

• readVoltage: Averages 100 voltage readings to improve accuracy.

**Setup Function**

void setup() {

  Serial.begin(115200);

  pinMode(RELAY\_PIN, OUTPUT);

  digitalWrite(RELAY\_PIN, LOW);

  if (!display.begin(SSD1306\_SWITCHCAPVCC, 0x3C)) {

    Serial.println(F("SSD1306 allocation failed"));

    for (;;);

  }

  WiFi.begin(ssid, pass);

  while (WiFi.status() != WL\_CONNECTED) {

    delay(500);

  }

  Blynk.begin(BLYNK\_AUTH\_TOKEN, ssid, pass);

  voltageSensor.setSensitivity(SENSITIVITY);

  timer.setInterval(1000L, myTimerEvent);

}

• Initializes peripherals, Wi-Fi, and Blynk. Sets up the OLED and configures sensors.

**Loop Function**

void loop() {

  Blynk.run();

  timer.run();

  float current = readCurrent();

  float rmsVoltage = readVoltage();

  Serial.print("Voltage : ");

  Serial.println(rmsVoltage);

  Serial.print("Current : ");

  Serial.println(current);

  Blynk.virtualWrite(V0, rmsVoltage);

  Blynk.virtualWrite(V1, current);

  display.clearDisplay();

  display.setTextSize(1);

  display.setTextColor(SSD1306\_WHITE);

  display.setCursor(0, 0);

  display.print("Power Management");

  calculatePowerAndEnergy(current, rmsVoltage);

  delay(1000);

}

• Runs the Blynk library, timer, and sensor readings.

• Updates Blynk and OLED with the latest data.

4. **Upload Code**:

• Compile and upload the code to your ESP32 using the Arduino IDE.

5. **Blynk App Setup**:

• Add widgets for displaying Voltage, Current, Power, Energy, and Cost.

• Add a Button widget for Relay control on Virtual Pin V4.

6. **Calibration**:

• Adjust the calibration factors for **ACS712** and **ZMPT101B** as per your hardware setup for accurate readings.

**Blynk Virtual Pins**

|  |  |  |
| --- | --- | --- |
| Virtual Pin | Parameter | Description |
| V0 | Voltage (V) | Real-time voltage readings. |
| V1 | Current (mA) | Real-time current readings. |
| V5 | Power (W) | Power consumption in watts |
| V6 | Energy (kWh) | Total energy consumption. |
| V3 | Cost (TK) | Energy cost in TK |
| V4 | Relay Control | Button to toggle relay state. |
| V2 | Uptime | Displays system uptime in seconds. |

**Example Output**

Voltage: 220.0 V

Current: 500.0 mA

Power: 110.0 W

Energy: 0.11 kWh

Cost\_Per\_Unit: 1.39 TK

**OLED Display:**

Power Management

Voltage: 220.0 V

Current: 500.0 mA

Power: 110.0 W

Energy: 0.11 kWh

Cost: 1.39 TK

**Future Enhancements**

• Integrate more sensors for advanced monitoring (e.g., temperature, humidity).

• Implement data logging using an SD card or cloud storage.

• Add energy-saving modes or automation based on thresholds.