



4/21/2020

# Home Energy Monitoring System

Michael Locke and Scott Palmer  
EETG SENIOR RESEARCH PROJECT REPORT

## Project Overview

Our project was developed in partnership with the NSCC Applied Energy Research Lab (AERLab) with the intention of designing a home energy monitoring system prototype. Our project focuses on the measurement of voltage, current, temperature and flow within a residential environment. All measurement data collected from our device is stored temporarily within the ESP32s random access memory before being sent wirelessly to a server in the lab for storage.

The main idea behind our project was to develop a data acquisition system that can run continuously within a home, which would give the homeowner insight into their energy habits so that the decisions that they make about their energy usage are based on data. We hope that the decisions that the user makes based on the data provided by our system will allow their homes to become more efficient with their energy usage, which will allow them to save money and do their small part in combating climate change.

Our prototype is capable of measuring two voltages, five currents, three temperatures and one flow. The voltage measurements are intended to give the user an idea of when their line voltage deviates from the ideal  $120V_{RMS}$ . The current measurements are intended to monitor how much current is being used in a particular area of the home so that the device can calculate the amount of power being used. The temperature and flow measurements are both intended for use with the homes hot water tank. Data suggests that hot water tanks account for a large percentage of residential energy usage so our device puts emphasis on both quantities.

As stated above, our project is a prototype energy monitor. If the lab decides to move forward with the project, there are a number of changes that can be made to improve the quality of our device. These changes will be discussed in the relevant sections of this report.

## Energy Monitor Operation

Our project is designed to operate continuously throughout its lifetime so that the homeowner can acquire the most comprehensive data pertaining to the energy usage of their home. From a programming perspective, this means that the device is programmed to run in a loop, where each iteration of the loop calls the functions required to measure a particular physical quantity. In this section of the report, we will give an overview of the main loop programming and how the programming interacts with the hardware to give the reader an idea of how the device operates.

The loop begins by calling a function named `WIFI_CHECK()`, whose purpose is to check to see whether the device is connected to the internet or not. During normal operation, the device will be connected to the internet, which is communicated to the user by the illumination of a blue LED. In the event that the device loses connectivity, the blue LED is turned off and a red LED is turned on. Given that our device is designed to operate continuously, thorough data logging requires that the device be connected to the internet. In the future, we would like to add an SD card port which will allow the data to be diverted to the card until the device regains connectivity.

Following the `WIFI_CHECK()` function, our device measures voltage by calling the `Voltage()` function. Two float variables, `V1` and `V2`, have been declared for use with the `Voltage()` function. By setting `V1 = Voltage()` and `V2 = Voltage()`, the results of our voltage measurements are stored inside of `V1` and `V2`.

Next, our device begins measuring current. The current sensors are connected to an analog-to-analog multiplexer, so in order to measure current, we first must place a call to the `CH15()` function. The `CH15()` function is used to output a bit pattern from the ESP32 that is equivalent to  $15_{10}$ . This bit pattern is used to select channel 15 on the multiplexer, and once selected, the output from our current sensor passes through the multiplexers unique input to the multiplexers common output. The output of the current sensor is a voltage that is proportional to the current being measured. Eventually, that voltage reaches the analog-to-digital pin on the ESP32 that is dedicated to measuring current. By calling our `Current()` function, the ESP32 takes the voltage on the pin and applies a transfer function, which is defined as  $y = (0.6887 * \text{average}) + 0.2108$ , to produce a value that is proportional to the current being measured by the sensor.

Our device measures five currents total, and the output of each current sensor is connected to a unique channel on the multiplexer. In order to measure the output of all these current sensors, our device is programmed to sequentially call the functions CH15(), CH14(), CH13(), CH12(), CH11() and CH0(), which address the corresponding channel on the multiplexer. Each of these addressing functions have a float variable and a call to the Current() function associated with it to store the current being measured by that sensor. Once all five currents have been measured and stored, our device moves on to measuring temperature.

The measurement of temperature is the simplest of all our measurements. To measure temperature, we make use of both the OneWire.h library and the DallasTemperature.h library. Outside of the main loop, we declare a const int variable named ONE\_WIRE\_A and give it a value of 23. Next, we create an object named oneWireA, which is of class OneWire, and instantiate it by giving it ONE\_WIRE\_A as an argument. These actions allow us to declare digital pin 23 on the ESP32 as a OneWire bus line. Next, we create an object named sensorsA, which is of class DallasTemperature, and instantiate it with the oneWireA object. We call the member function requestTemperatures() of the sensorsA object, which requests the current temperature of any sensors on the bus line. The values of the temperature sensors are stored inside of three float variables.

After measuring temperature, our device measures flow. To measure flow, we place a call to our flow function by saying that flow1 = Flow(). By setting flow1 equal to the return value of the Flow() function, we store the result of the flow measurement inside of the float variable flow.

Once flow has been measured and stored, our device is ready to format all the measured values as a string to send the data to the server in the lab. To format our measurements as a string, we place a call to the sprintf() function. The sprintf() function transforms our float data into a string of characters and stores the string in a buffer. To send the data to the server, we place a call to post() function which takes the address of the buffer where the data string is stored as its argument. Once the data has been sent to the server, the code returns to the beginning of the loop and starts the whole process again.

## Component List

The following is a list of the components used in this project. More information on these components and how they are used can be found later in this report.

- a) ESP32 Microcontroller (1)
- b) CD74HC4067 Multiplexer (1)
- c) ZMPT101B Voltage Sensor (2)
- d) YHDC SCT013-005 Current Sensor (5)
- e) DS18B20 Temperature Sensor (3)
- f) Unknown flow sensor from the lab (1)
- g) LD1117V33 3.3V Voltage Regulator (1)
- h) SJ1-3523N 3.5mm Aux Connector (5)
- i) 54-00167 2.5mm Power Barrel Connector (1)
- j) 282837-3 Terminal Block (1)
- k) 68uF Capacitor (1)
- l) 330k $\Omega$  Resistors 5% (12)
- m) 100k $\Omega$  Resistors 5% (3)
- n) 4.7k $\Omega$  Resistor 5% (1)
- o) 120 $\Omega$  Resistor 5% (1)
- p) LEDs (2)

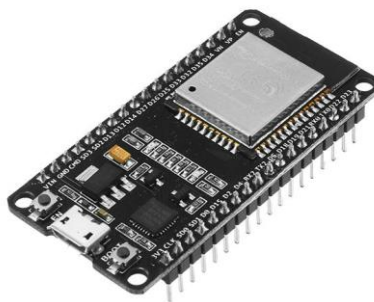
## Summary of the Microcontroller and Sensors

In this section of the report we discuss how our project was designed and the reasoning behind each design decision. We will discuss why we have chosen the components that we have, and how we make use of them.

### ESP32 Microcontroller:

The microcontroller that we have chosen for this project is the ESP32, which is manufactured by Espressif Systems. The ESP32 is a powerful 3.3V microcontroller that has built-in Wi-Fi capability, which is essential to our project. In addition to Wi-Fi capability, the ESP32 gives us 15 analog-to-digital converters (12-bit resolution), 3 SPI interfaces, 3 UART interfaces, 2 I<sup>2</sup>C interfaces, 16 PWM output channels, 2 digital-to-analog converters and 10 capacitive sensing GPIOs. The most important factor in our decision to use the ESP32, however, is the cost. The ESP32 chip itself costs ~\$2, and an ESP32 devboard is ~\$8.

Overall, the ESP32 has been a net-positive for us, but it is not without its flaws. The microcontroller becomes quite power-hungry when the designer makes use of its wireless communications capabilities, which, depending on the application, may render the ESP32 useless. Another major flaw is that when connected to Wi-Fi, every ADC on channel 2 becomes unusable, leaving only the six ADCs on channel 1 remaining. The ESP32 is also quite susceptible to noise, which means that in order to accurately measure analog values, the designer must employ heavy statistical sampling and averaging when taking their measurements.



Despite these flaws, we were able to make use of the ESP32 to control all our sensors.

## ZMPT101B Voltage Sensor:

For our voltage measurements, we decided to use a ZMPT101B AC voltage sensor. This device takes a high AC voltage, up to 250V, and transforms it down to 3.3V<sub>pp</sub>. It then adds a DC offset of 1.65V to the output so that it never goes into negative voltages, which can potentially damage the microcontroller. This device can be connected directly to an analog pin of the ESP32.



In order to take a voltage reading from this wave input, we place a call to our Voltage() function. The Voltage() function is defined as:

```
float Voltage()
{
    float Period = 16667;
    float average, voltage, start, sum, minv, maxv;
    int value;

    average = 0;
    voltage = 0;

    for (int i = 0; i < NO_OF_SAMPLES; i++)
    {
        sum = 0;
        maxv = 0;
        minv = 1024;
        start = micros();
        while ((micros() - start) < Period)
        {
            value = adc1_get_raw((adc1_channel_t)channel4);
            if (value > maxv)
                maxv = value;
            if (value < minv)
                minv = value;
        }
        sum = sum + (maxv - minv);
    }
}
```

```

    }
    average = sum / NO_OF_SAMPLES;
    voltage = (215.96 * average) - 132.99;
    return voltage;
}

```

The voltage function works by finding the maximum and minimum values over a single period. We do this by defining our period as 16667 in microseconds for a 60 Hz signal. Using the function `micros()`, which returns a value in microseconds for the current time, we can create a while loop with the condition of `((micros() - start) < Period)`, which will loop over a single period of the wave. In this loop we take analog readings of the voltage pin and obtain the maximum and minimum readings during the period, then take the sum of the difference between them. The method of using both maximum and minimum values instead of just maximum, eliminates any error that might come from variations in the DC offset. We take these readings over ten periods to produce a more stable reading, then using our transfer function, defined as:  $\text{voltage} = (215.96 * \text{average}) - 132.99$ , we convert this value into a reading for our RMS voltage.

### **YHDC SCT013-005 Current Sensor:**

For the current measurements, we have chosen to use five YHDC SCT013-005 split core current sensors. These sensors clip around a wire and are able to measure currents of up to 5A, with a sinusoidal output of up to 1V<sub>RMS</sub>. If the lab ever decides to move forward with this device beyond the prototyping stage, the final product will need to measure currents in the range of 50A. The decision to use the 5A sensors was reached while taking into consideration several factors, most notably: availability and ease of use.

The lab already had a couple of these sensors laying around, so we decided to use what was already available and ordered a few more. The biggest factor in our decision to use these sensors though was how difficult it was to generate large currents to test the sensors with and calibrate them. The lab had several sensors similar to this that could measure currents in the range of 30A, 50A and 100A, but without a reliable method of generating currents in these ranges, it becomes incredibly difficult to make use of the full range of the sensors input. For example, if you have a sensor that can measure 50A with an output of 1V<sub>RMS</sub> and you are only measuring a current in the range of 5A, you are only using 10% of the sensors range, meaning



that the output voltage will be  $\sim 0.1V_{\text{RMS}}$ . Using so little of the sensors range makes it very difficult to accurately measure changes in current. Without a reliable way to generate large currents, the logical choice of sensors was to use the 5A sensors.

To generate current, we used a variable AC voltage source and a small heatsink that had ten  $1\Omega$  resistors attached to it and connected in series. Each resistor was able to dissipate 20W of power, giving us a total of 200W of power dissipation to work with. With the voltage source set to 25V, we were reliably able to generate current in the range of 2.5A, which required that the heatsink dissipate 62.5W. We found that we could safely operate at around 2.5A but pushing it much higher than that resulted in smoke. The highest current that we could measure without causing a fire was 3.5A.

The sensors connect to our board using a 3.5mm aux jack for the output. Before we can connect them to the microcontroller, we must first give them a DC offset, so that the negative voltages do not damage the ESP32. The DC offset is produced using a simple voltage divider that consists of two  $330k\Omega$  resistors, which divides the voltage from the 3.3V rail in half. Dividing the 3.3V rail in half allows us to measure the sinusoidal output signal of the current sensor with a DC offset of 1.65V, by placing the output of the sensor in the middle of the voltage divider.



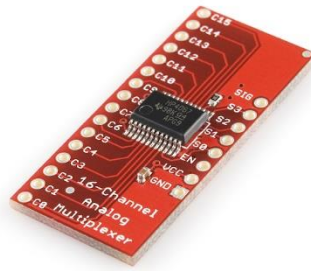
Our code for the Current() function uses the same logic as the Voltage() function, with the only real difference being the transfer function used to map the input signal to the output signal. The code for the Current function is as follows:

```
float Current()
{
    float Period = 16667;
    float average = 0, current = 0, start = 0, sum = 0, mini = 0, maxi = 0;
    int value;
    average = 0;
    current = 0;

    for (int i = 0; i < NO_OF_SAMPLES; i++)
    {
        sum = 0;
        maxi = 0;
        mini = 1024;
        start = micros();
        while ((micros() - start) < Period)
        {
            value = adc1_get_raw((adc1_channel_t)channel);
            if (value > maxi)
                maxi = value;
            if (value < mini)
                mini = value;
        }
        sum = sum + (maxi - mini);
    }
    average = sum / NO_OF_SAMPLES;
    current = (0.6887 * average) + 0.2108;
    return current;
}
```

### CD74HC4067 Multiplexer:

We have chosen to use the CD74HC4067 analog-to-analog multiplexer to handle all our current measurements. The decision to use the CD74HC4067 as our multiplexer was based around cost (\$7.76 CAD) and how simple the multiplexer is to operate. The CD74HC4067 requires that four digital output pins from our microcontroller be connected to it for addressing purposes, while also requiring that one ADC channel from the microcontroller be connected to the common output of the CD74HC4067.



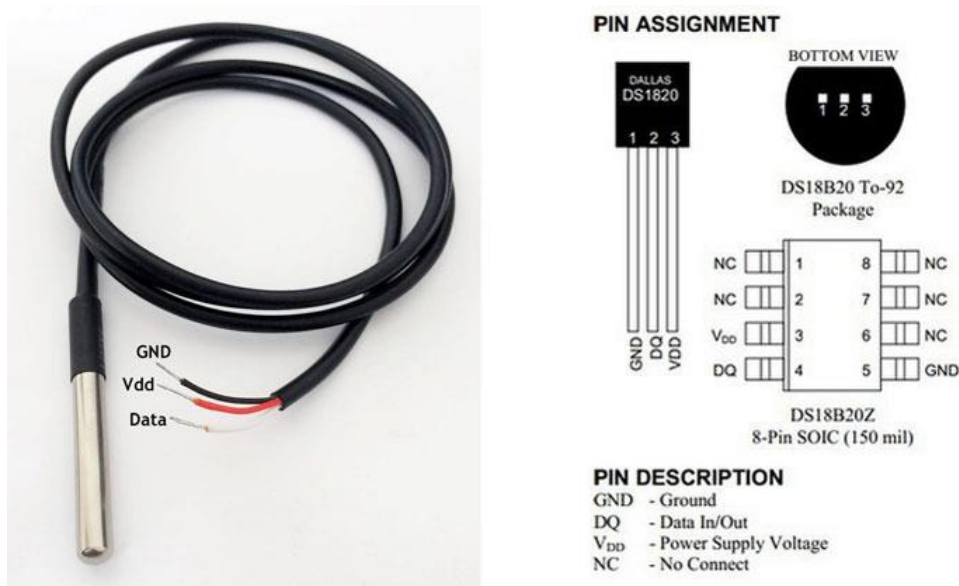
We have written several functions, that when called, output a bit pattern on the digital outputs of the ESP32. These functions are used to select one of sixteen channels (0 to 15) on the CD74HC4067. When a given channel is selected, the output voltage signal from the current sensor passes through the multiplexer and reaches the ADC on the ESP32. From a programming perspective, we have five float variables named C1, C2, C3, C4, C5 for current which correspond to a unique channel on the CD74HC4067. Addressing the channel that corresponds to the appropriate current variable allows the output of that current sensor to be stored inside the correct current variable. Once all the currents have been stored, the program moves onto temperature and flow before formatting all the measurement data as a string to be sent to the server for storage.

The CD74HC4067 is a critical component in the functionality of our project. Without it, we would simply not have enough ADC pins on the ESP32 necessary to measure the number of currents requested by the lab. The CD74HC4067 acts as an extension of the ESP32 by increasing the number of ADC pins we have to work with.

### **DS18B20 Temperature Sensor:**

To measure temperature, we have chosen to use the DS18B20 digital temperature sensor. The DS18B20 communicates with the microcontroller using the 1-Wire bus protocol. The 1-Wire protocol is a serial communication protocol that allows for half-duplex communication using two wires: data and ground. The DS18B20 can operate in parasitic power mode, which allows the device to draw all its operating power from the data line. The DS18B20 that we chose

comes in the standard TO-92 package, which includes a  $+V_{cc}$  input. To avoid unforeseen complications, we felt it best to provide operating power from the power rail.



For prototyping purposes, we have only three DS18B20 sensors connected to our device. Our program treats each sensor as its own device, and we measure three different temperatures which are sent along with the rest of the data to the server as a string.

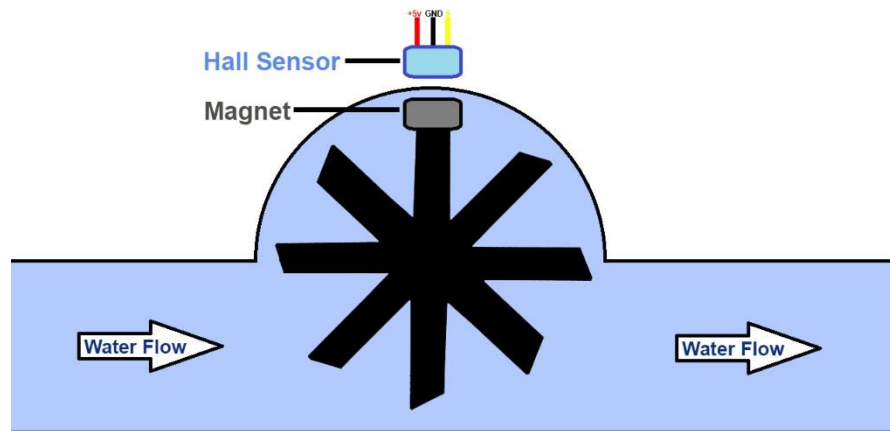
Beyond the prototyping stage, we would employ a large number of DS18B20 sensors by attaching them to various locations on a hot water tank. The reason for this is so that an accurate average temperature for the water in the tank can be measured. To accomplish this, we would program our device to create an array of sensors. This array would be accessed from a for() loop and the temperature of each device would be added to the previous temperature to give us the sum of all the temperatures. The average temperature of the water in the tank would then be found by simply dividing the sum by the number of sensors on the bus.

The lab requested temperature for the purpose of calculating how much energy is being used by the hot water tank in a home. By knowing the temperature, we can compute how many kWh are being used by the tank. Knowing the kWh, as well as the amount of water leaving the

tank, which is given by our flow sensor, we aim to provide the homeowner with the information necessary to save energy and water, both of which are precious resources.

### Flow:

For the flow measurement, we use a standard Hall Effect water flow sensor. Water passes through the sensor, which causes a rotor to spin. The rate at which the rotor spins is proportional to the amount of water flowing through the sensor. The sensor then outputs a square wave with a frequency that is proportional to the rate of flow. The sensor connects to our 3.3V rail and ground, and its output can be measured on a digital input pin of the ESP32.



The code for the flow function makes use of the `pulseIn()` function on the ESP32. This function outputs a length of time in microseconds for either a high or low pulse on a digital pin. Using the sum of both of high and low times give us the period for the wave. We repeat the measurement 10 times to obtain a stable reading, then convert the reading into a frequency by taking the reciprocal of the period and multiplying it by 1000000 to convert it into seconds. We then use a transfer function to turn the frequency into a reading for flow rate in L/min. The function to measure flow is defined as:

```
float Flow()
{
    float frequency = 0, avgperiod = 0, flow = 0;
    long high = 0, low = 0, period = 0, sum = 0;
```

```

const int flow_pin = 21;

for (int i = 0; i < 100; i++)
{
    high = pulseIn(flow_pin, HIGH, 250000);
    low = pulseIn(flow_pin, LOW, 250000);
    period = (high + low);
    sum = sum + period;
}

avgperiod = sum / 100;
frequency = 1000000 / avgperiod;
flow = (frequency * 0.1151) + 0.9628;
return flow;
}

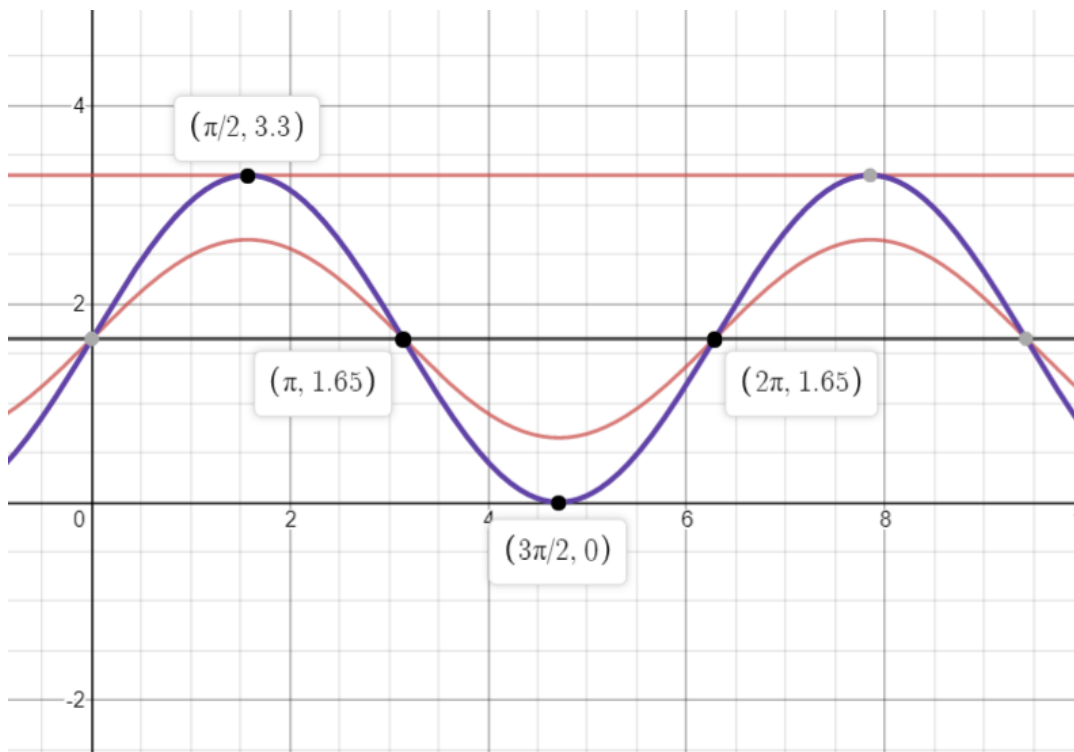
```

We created the transfer function for flow by using a steady flow from a hose, and a bucket marked with volume measurements. By timing how long it takes for the hose to fill the bucket up to a certain point, we can calculate the flow rate and use that as our standard measurement. In the future we will use a more accurate method to obtain a transfer function by comparing our values to a trusted, calibrated flow meter.

As mentioned in the DS18B20 section, our device measures the average temperature of the water in the tank, as well as the amount of water leaving the tank. The reason for both of these measurements is to obtain an accurate picture of the amount of hot water being used, and how much energy is being used to heat the water.

## Sinusoidal Signal Conditioning

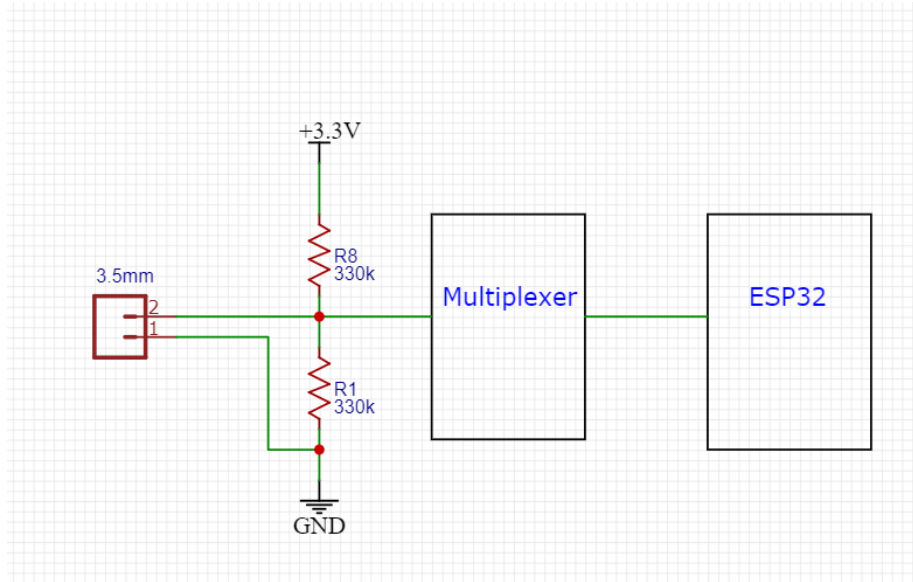
A significant portion of our project involves the measurement of both sinusoidal voltage signals and sinusoidal current signals. In order to measure these signals, we must condition the signals to be compatible with the ESP32s maximum voltage input, while also conditioning the signals so that they never drop below zero, which can damage the microcontroller. This section of the report discusses the techniques used in conditioning the sinusoidal output signals from the voltage sensors and current sensors in more detail, while also discussing how transfer functions are used to measure the signals.



To begin, we must create a DC offset of 1.65V, which acts as the reference for the signal. So instead of crossing the 0V threshold, which causes the signal to reverse polarity, the reference is now 1.65V, which ensures that the signal stays positive so that it does not damage the ESP32.

The ZMPT101B voltage sensors are shipped on a small PCB that have the voltage dividers in place to create the DC offset. The same is not true for the YHDC SCT013-005 split core current sensors. In order to properly use these current sensors, we must create the DC offset ourselves. To accomplish this, we have created voltages dividers using two 330k $\Omega$  resistors with the tops of the voltage dividers connected to the 3.3V rail and the bottoms of the voltage dividers

connected to ground. The output of the current sensors are connected to a 3.5mm aux jack, with the signal wire connected to the middle of the voltage divider, which will allow the sinusoidal signal to potentially reach as high as 3.3V, and as low as 0V, while never actually going below 0V.

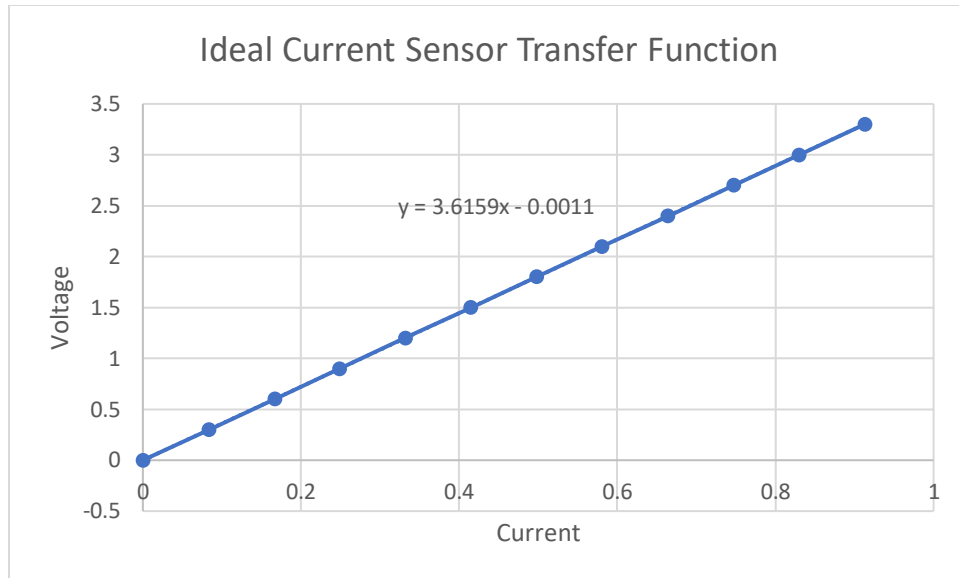


The above picture illustrates the basics of our voltage divider setup, which has the output of a current sensor connected to the 3.5mm aux jack. The ESP32 will output the bit pattern that corresponds to the multiplexer channel that a particular current sensor is connected to. When that bit pattern is output, the channel is selected and the signal from the current sensor passes through the multiplexer and arrives at the ESP32 ADC channel for measurement.

With the DC offset established and the sinusoidal signals reaching the ADC channels of the ESP32, we can begin measuring the signals. In order to properly measure these signals, we must make use of transfer functions.

Transfer functions are mathematical functions, in the form of  $y = mx + b$ , that take a physical quantity, such as current, and maps a range of values for this quantity over a small voltage range. For example, the current sensors used in our project can measure current from 0A to 5A. A current value of 0A would be described by 0V and a current value of 5A would be described by 3.3V. An ideal transfer function for this current sensor would be:

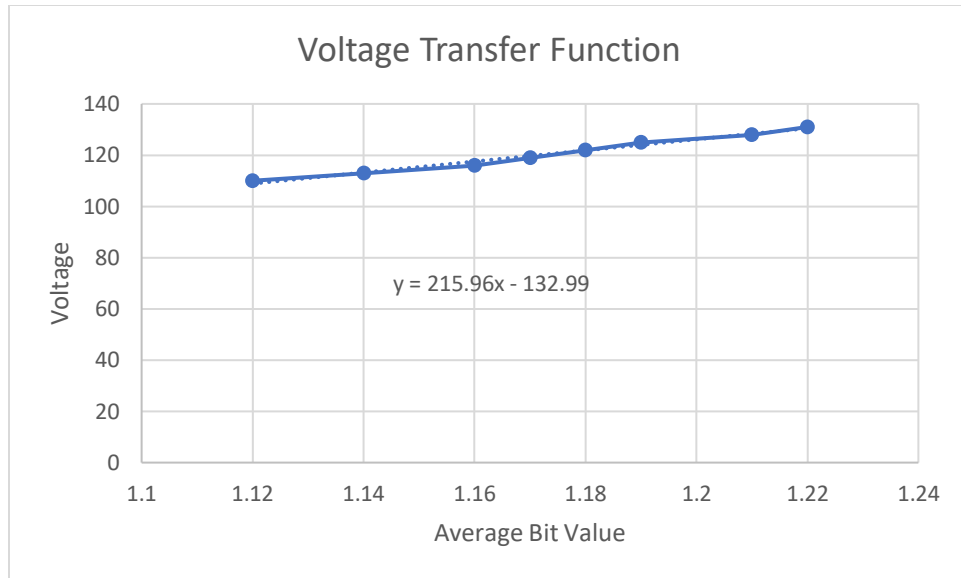




The above transfer function is ideal and ignores the conversion of voltage to a bit value that occurs inside the microcontroller, but demonstrates the basic concept of a transfer function. The transfer function used in our device for measuring current is derived by producing currents from 0A to 3.5A, and in 0.5A increments, recording the average bit value that the ESP32 assigns to the voltage that is present at the ADC pin for that current value. With the current on the y-axis and the average bit value on the x-axis, graphs are produced in Microsoft Excel that yield an equation in the form of  $y = mx + b$  that is used to return the current being measured by the sensor.

Many transfer functions were created in the process of refining our current measurements, and the final transfer function used to measure current is:  $\text{current} = (0.6887 * \text{average}) + 0.2108$  where current is the dependent variable y and the average bit value is the independent variable x multiplied by the slope. Unfortunately, the data used to produce this transfer function has been deleted so a graph of the data cannot be shown in this report. It is worth noting that the graph of the function is non-ideal and is not as linear as the ideal function shown above.

Our device is only concerned with measuring voltage in the range of ~110V to ~126V, so using the same process for deriving our transfer function for current, we started at 110V and recorded the average bit value given to that voltage by the ESP32, and in 3V increments, recorded the average bit value for each voltage up to 131V. The graph of that data is as follows:



This graph is not as linear as we would like, but it does a good job of mapping a large voltage value to a small voltage value that is compatible with the ESP32 voltage input.

## Wireless Transmission of Measurement Data

Our device is programmed to run continuously in a loop where each iteration of the loop measures voltage, current, temperature and flow. Each measurement is saved temporarily as a floating-point variable before being formatted as a string and sent to the server in the lab for storage. This section of the report will discuss the process of sending data in more detail.

As stated earlier in this report, after measuring flow, all the measurement data is converted to a string using the `sprintf()` function and stored temporarily in a character array. The address of this character array is then passed to the `post()` function as the functions argument. The `post()` function is used to send the data string to the server in the lab and is defined as:

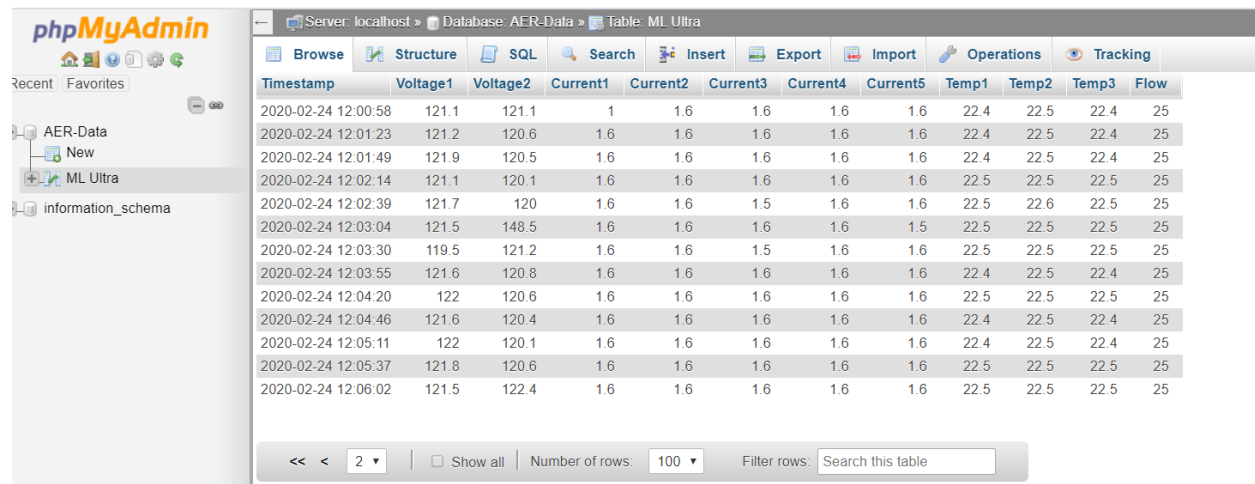
```
void post(String PostData)
{
    Serial.println("Connecting to Server");
    if (client.connect("192.168.1.151", 80))
    {
        Serial.println("connected");
        client.println("POST /data/post.php HTTP/1.1");
        client.println("Host: 192.168.1.151");
        client.println("User-Agent: ESP32");
        client.println("Connection: close");
        client.println("Content-Type: application/x-www-form-urlencoded");
        client.print("Content-Length: ");
        client.println(PostData.length());
        client.println();
        client.println(PostData);
    }
    else Serial.println("Connection Failed");
}
```

The `post()` function makes use of a protocol known as HTTP post requests, which have very specific requirements. In order to successfully send the data, we must first let the server know that we seek to post data, using the `client.println("POST /data/post.php HTTP/1.1")` line of code. Next, we must send the appropriate IP address of the host server so that the data is sent to the correct server by sending `client.println("Host: 192.168.1.151")`. We then let the server know which device is sending the data using the `client.println("User-Agent: ESP32")` code.

The HTTP post request then requires that we specify which kind of content we will be sending to the server. By sending `client.println("Content-Type: application/x-www-form-urlencoded")` to the server, we let the server know that we will be sending string data. After

specifying the content type, we need to give the server the exact length of the string that we will be sending using `client.print("Content-Length: ") client.println(PostData.length())`. If the length of the string is off by +/- 1 value, the entire data will not be sent correctly. After all these requirements have been met, the data can be sent to the server by giving the argument of the function (the character array where the string is stored) by saying `client.println(PostData)`.

Once the data has been successfully sent to the server, the data appears in a table as follows:



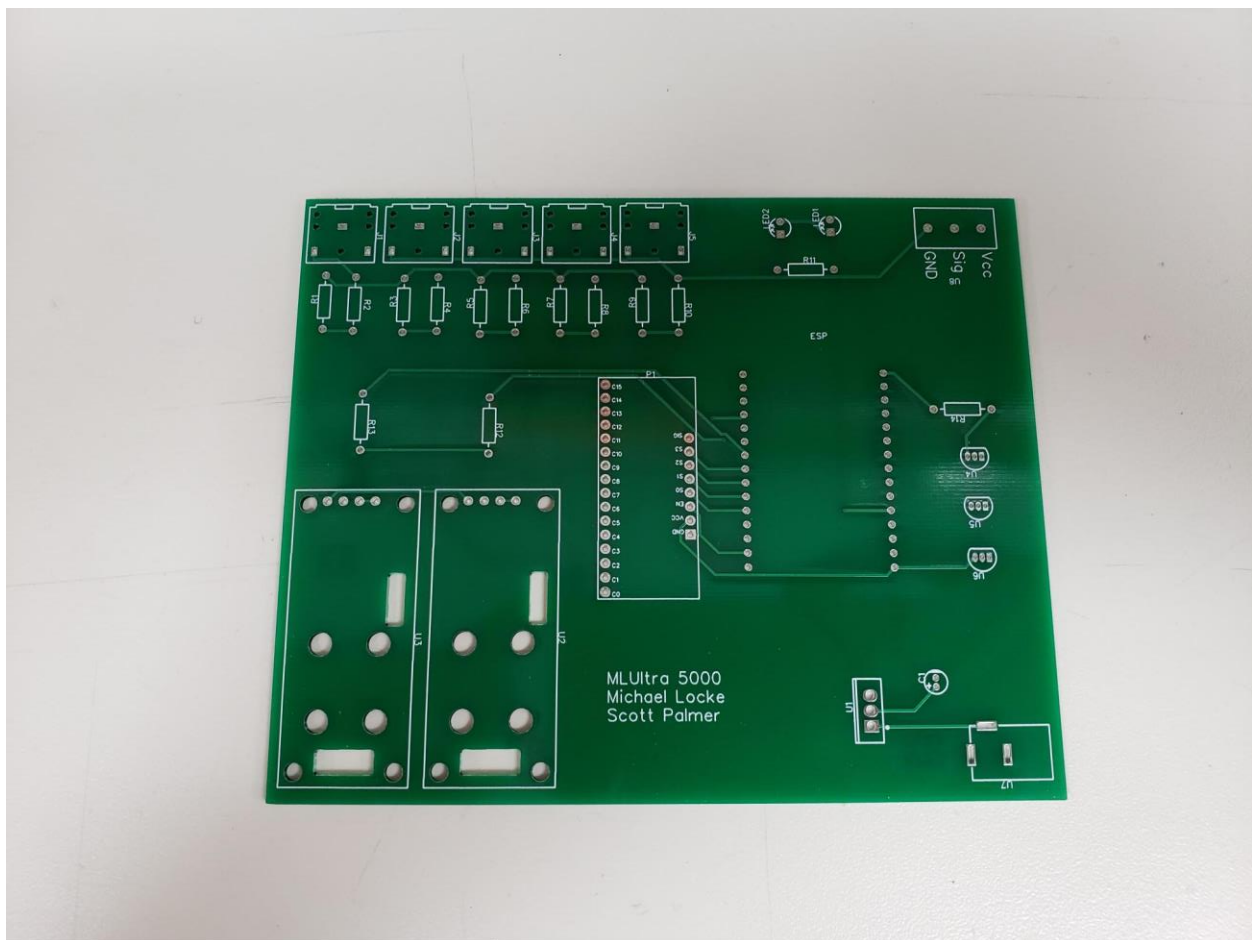
Timestamp	Voltage1	Voltage2	Current1	Current2	Current3	Current4	Current5	Temp1	Temp2	Temp3	Flow
2020-02-24 12:00:58	121.1	121.1	1	1.6	1.6	1.6	1.6	22.4	22.5	22.4	25
2020-02-24 12:01:23	121.2	120.6	1.6	1.6	1.6	1.6	1.6	22.4	22.5	22.4	25
2020-02-24 12:01:49	121.9	120.5	1.6	1.6	1.6	1.6	1.6	22.4	22.5	22.4	25
2020-02-24 12:02:14	121.1	120.1	1.6	1.6	1.6	1.6	1.6	22.5	22.5	22.5	25
2020-02-24 12:02:39	121.7	120	1.6	1.6	1.5	1.6	1.6	22.5	22.6	22.5	25
2020-02-24 12:03:04	121.5	148.5	1.6	1.6	1.6	1.6	1.5	22.5	22.5	22.5	25
2020-02-24 12:03:30	119.5	121.2	1.6	1.6	1.5	1.6	1.6	22.5	22.5	22.5	25
2020-02-24 12:03:55	121.6	120.8	1.6	1.6	1.6	1.6	1.6	22.4	22.5	22.4	25
2020-02-24 12:04:20	122	120.6	1.6	1.6	1.6	1.6	1.6	22.5	22.5	22.5	25
2020-02-24 12:04:46	121.6	120.4	1.6	1.6	1.6	1.6	1.6	22.4	22.5	22.4	25
2020-02-24 12:05:11	122	120.1	1.6	1.6	1.6	1.6	1.6	22.4	22.5	22.4	25
2020-02-24 12:05:37	121.8	120.6	1.6	1.6	1.6	1.6	1.6	22.5	22.5	22.5	25
2020-02-24 12:06:02	121.5	122.4	1.6	1.6	1.6	1.6	1.6	22.5	22.5	22.5	25

All the data is timestamped and each measurement is placed in the appropriate column. A homeowner would be able to log into the server with their unique login ID and password to view their data.

## COVID-19 Considerations

Overall, we are content with what we have accomplished throughout the course of this project. There are, however, certain aspects of our project which remain unfinished due to the ongoing pandemic and in this section of the report we will discuss the work that still needs to be done.

The most important aspect of our project which remains unfinished is the assembly of our PCB. An order was placed for a PCB sometime before March break and the PCB was delivered to the lab on April 14<sup>th</sup>, 2020. A picture of our PCB without any components is shown below:



Our voltage measurements have been plagued with noise since the beginning and we have been unable to properly diagnose the problem. The transfer function used to measure voltage is reliable, and we compensate for the intrinsic noise of the ESP32 ADC channels by

using the appropriate amount of statistical sampling. Accurate measurements within 1V of the measured line voltage can be measured by our device. Despite all of this, we find that there are frequent fluctuations in our measurements which we have been unable to explain. Our device can measure 120V for several iterations of the loop, only to drop to ~110V and jump to ~130V. We believe that these fluctuations are the result of outside interference in the form of stray charges in the air, as well as stray capacitance in the length of wires used to connect our voltage sensors to the microcontroller.

By completing our PCB, we would be moving our circuit off the breadboard and into a case which would alleviate some of the noise by stray charges in the air. We would also be switching our connections to much smaller copper traces as opposed to longer lengths of wire, which would help eliminate some of the stray capacitance that is present in the wires. We had hoped to assemble our PCB to test our hypothesis regarding the fluctuations in our voltage measurements, but due to the pandemic, we have been unable to do so.