Sam Ross

BEng (Hons) Final Year Project Report

# Smart Wireless Sensor

April 2023

## Abstract

This project aims to research, design, and develop an embedded smart wireless sensor solution that can augment individual three-phase system Digital Fault Recorders (DFRs) to solve their main two problems – the DFR recordings and other useful near-real-time measurements/metrics cannot be accessed/retrieved remotely and secondly, it may be electrically impossible to install DFRs in certain substation scenarios due to their hardwiring requirements and size constraints.

The ESP32-DevKitC-32UE MCU was concluded as the optimal prototyping platform for the project. A range of other components were acquired, including two ADS1015 ADCs for taking voltage readings, a MicroSD card breakout for storing the useful live measurements, a GPS breakout for interrupt timer disciplining and a secondary ESP32 for handling external communications. Two signal conditioning circuits were then designed and engineered to attenuate the secondary-side voltage (VT) and current (CT) signals down to 1VRMS, which were then fed into the two ADCs. The 1200sps readings taken by the ADCs are then run through various validated algorithms to calculate the 12 final useful live measurements. These 12 near-real-time measurements are then displayed on the secondary ESP32 Wi-Fi Access Point & Station Web Servers, along with being sent through a backend pipeline to a cloud-hosted React JS website and stored in cloud data storage. The true frequency and VRMS algorithms were validated through a simulation in MATLAB and found to be over 33 times more accurate than originally required (+-0.3mHz and +-0.3mV, respectively).

## Table of Contents

## Specification

The ongoing operation of interconnected power systems is established by real-time measurement of network parameters, typically line voltage and current, frequency, power, protection battery voltage and current and digital contacts, and other quantities including cable temperatures. Measurements are typically derived from sensors, for example voltage and current transformers (VT and CT), embedded on the network normally on the secondary side (protection level) of transmission or distribution transformers. Traditionally embedded sensors are hard-wired and coupled to protection and fault-monitoring instrumentation using copper and twisted-pair wiring. Innovation in sensor technologies has explored alternative methods of coupling, including optical-fibres and wireless sensors.

This project will investigate wireless sensing and develop a hardware and software approach to measure low-voltage and current quantities by way of a proof-of-concept solution. The project will require use of traditional VT and CT sensors coupled by wireless transmission (Bluetooth/Wi-Fi) to a receiver (e.g. a PC or mobile phone) to send measurements in near-real-time for display and storage. The solution will be used to evaluate the potential for wireless sensing as a basis for smart metering (and billing) for single and potentially three-phase systems. Project outcomes may form a basis for determining energy consumption or assessing quality of supply. The solution may be implemented on a COTS (commercial-off-the-shelf) small-footprint platform (e.g. Raspberry Pi/Arduino/Teensy/Pi Zero) or suitable alternative platform. Operating scenarios will typically include: remote measurement of voltage, current, harmonics, transients, noise, and incomplete data. The project will require excellent knowledge of C/C++ programming, analogue/digital hardware, basic digital signal processing, wireless communications and power engineering.

## Project objectives

**Objectives**

1. Investigate power system measurement and wireless sensing
2. Design a near-real-time solution in hardware and software to measure low voltage/current signals
3. Develop appropriate hardware/host software for wireless PC/laptop/GUI control and interfacing
4. Write C/C++ software to implement the solution for near-real-time measurement
5. Write C/C++ software to implement near-real-time determination of other quantities (e.g. power/energy)
6. Evaluate and test the smart sensors with different operating scenarios

**Bonus MEng Extension**

1. Develop appropriate hardware and software for bi-directional sensor measurement interfacing
2. Develop a broad range of operating scenarios with different voltage/current attributes for smart metering
3. Implement and evaluate final solution with simulated/measured/laboratory signals

## Acknowledgements

I would like to express my special thanks of gratitude to Dr. Littler for his able guidance and support in completing this project.

I would also like to extend my gratitude to Tony Boyle for his help with securing the required components for this project.

## Declaration of originality

I declare that this report is my original work except where stated.

Sam Ross

01/04/2023

# List of Figures

# Introduction

## Introduction Overview

In power substations, a single electrical fault can cause power outages that cause millions of pounds worth of damage if not dealt with correctly. The circuit breaker/protection relay systems detect when there is a fault on the line and provide alternative voltage support to maintain the required voltage to the load. Digital fault recorders are used to record events before, during and after a disturbance to figure out what has happened on the grid and they facilitate a post-fault analysis to be made in order to make improvements to overall reliability. In Distributed DFR systems, the individual DFRs around the plant are hardwired to a Data Concentrator via Serial or Digital communications.

This architecture leads to two main flaws with DFRs. The DFR recordings and other useful live measurements – such as the number of faults, voltage, current, energy consumption, power consumption, etc. – can only be acquired by physically accessing the DFRs units themselves. This is impractical and may even be physically impossible. Secondly, due to the large size of the DFRs and their requirement for hardwired external communication connections, there may be scenarios within a substation where it would be electrically impossible to physically place the DFRs without breaching the electrical integrity of the system.

This project aims to develop an optimal hardware and software smart wireless sensor solution to solve these two stated problems that individual DFRs have in a distributed DFR system.

This project researches into power systems and wireless sensing and then utilises this knowledge to form the requirements needed for an optimal solution. Research is conducted into various platforms and breakouts, which are then analysed and compared to determine the optimal embedded circuit solution along with a signal conditioning attenuation circuit. The embedded circuit is then developed and algorithms are written (and validated through simulation in MATLAB) to calculate the 12 final useful near-real-time measurements, which are then made available to users via 4 different forms of external communications.

## Background

### What are electrical faults on power lines and what causes them?

In electrical power systems, faults are any abnormal electrical currents. Faults on power transmission lines are caused by anything that can cause conductors to clash and hence short circuit (current maximises and voltage drops) e.g. strong winds, lightning, bird strike. [1]

### Why are electrical faults undesirable?

In the scenario of an automated factory, undealt with faults on the power line would cause a voltage drop in factory machines and hence would cause them to malfunction/shut down. This would result in the whole factory line having to be reset, potentially costing millions. Therefore, it is crucial that the faults are detected and dealt with accordingly as well as prevented in future.

## How are faults detected and dealt with in power systems?

A **Protection Relay (PR)** and **Digital Fault Recorder (DFR)** are coupled to the secondary side (protection level) of the transmission and distribution transformers, using copper and twisted-pair wiring. [1]

The Protection Relay is a relay device that trips a circuit breaker when a fault is detected i.e. when the voltage drops by a specified amount (e.g. voltage drops by 10%) or if the current increases by a specified amount. Voltage support (e.g. using a Static VAR Compensator) will then be relied on to supply additional power to the loads to ensure that the required voltage is maintained.

A Digital Fault Recorder is a multi-channel, Intelligent Electronic Device (IED) that uses communications to retrieve fault, disturbance, and sequence of event records that are captured by the protection relays distributed throughout a substation. [2]

DFRs have two main purposes:

- Recording of system events, such as faults – these can be analysed later in a "post mortem" methodology to assess the performance of protection relays and circuit breakers during the fault
- Monitoring of system protection performance and quality of supply

## Modern industry DFR systems

DFR systems have evolved over time – the utility industry progressed from using a needle and paper to more complex systems [3]. The utility industry is currently split between two types of DFR systems: **Distributed DFRs** (DDFR) and **Distributed Recording** (Virtual DFR). [4]

Distributed DFRs involve having a separate DFR installed in each Protection Relay panel whereas Distributed Recording makes use of the fault recording functionality that is already built into the Protection Relays. Both of these systems are connected to some sort of **Data Concentrator** by means of serial or digital communication, which is used to then collect, synchronise, and store the event records [4].

A dedicated Distributed DFR system will be better at recording than a virtual DFR system (higher sampling rate and resolution, etc.), but a virtual DFR system has a much lower installation cost.

## Flaws of both DFR systems

For consistency, this project will talk about improving the individual DFR units in a DDFR system but the same improvements can be made for the protection relay units in a Virtual DFR system.

As described earlier, both the DDFR and Virtual DFR systems must be **hardwired** to a Data Concentrator by means of **serial** or **digital communication**. This leads to two flaws in both systems.

### *Problem 1 – Inaccessible Live Measurements*

The DFR recordings and other useful live measurements – such as number of faults, voltage, current, energy consumption, power consumption, etc. – can only be acquired by physically accessing the DFR units (or protection relay units in Virtual DFR systems) themselves. This is extremely impractical (especially in large substations) and it may even be physically impossible to access the individual DFRs to acquire these near-real-time measurements.

### *Problem 2 – Breach of electrical integrity*

Due to the size and mass of DFRs and their reliance on hardwiring for external communication, this leads to scenarios where it would be electrically impossible to physically use DFRs for fault recording. For example, putting in large DFRs up at the mastheads of transformers and hardwiring them up will completely breach the electrical integrity of the system.

## Smart Wireless Sensor solution to solve DFR problems

The high sampling rates, large memory capacity, flexible triggering, and improved datasets of standalone DFRs make them the optimal solution for studying network performance and fault analysis [4]. **However**, DFR systems have a couple of problems, as described. The DFR systems **can be augmented** with a smart wireless sensor to provide solutions to these problems.

**Easily accessible live measurements** – Once a user comes within close enough proximity to the sensor, the user will be able to read useful live measurements – such as number of faults, voltage, current, energy consumption, power consumption, etc. – via wireless communications through the platform wireless communications (Wi-Fi/BLE server). By using the sensor in combination with the DFR, this will provide both accessible live measurements from the platform and optimal fault analytics for later use from the DFR.

**An electrically-safe solution** – It's a non-invasive solution, the voltage and current connections just need to be connected with a clamp. The sensor is battery-powered, wireless, portable, and low maintenance, hence making it the perfect solution to be clamped on almost anywhere that it would be electrically impossible to do so with a DFR. [1] In these scenarios where it is electrically impossible to use a DFR, the fault records produced by the smart wireless sensor will be adequate, although they are inferior in terms of sampling and hence resolution for analysis, etc. compared to that of a DFR. [5]

## Smart Wireless Sensor COTS Platform solution

Augmenting DFRs with COTS small-footprint platforms should solve all the DFR problems previously described. As an end goal, the platform should be able to asynchronously read the 8 analogue channels from the 3-phase system for smart metering (and billing) then use wireless transmission (Bluetooth/Wi-Fi) to a receiver (e.g. a PC or mobile phone) to send measurements in near-real-time for display and storage. It should have some form of local storage for the readings too and the samples should be taken in sync, regardless of the geolocation.

### 6 functionalities required of the final COTS Platform Circuit, as shown in Figure 1

**Wi-Fi and Bluetooth:** These wireless transmission forms are needed for wireless transmission of the useful near-real-time measurements to nearby smart devices. If a Wi-Fi network is available, these values could be transmitted via a local Wi-Fi/BLE server or even to cloud storage in future.

**8 channel ADC:** The 8 analogue inputs from the star 3-phase power system need to be **asynchronously measured** so that faults can be detected and recorded. Useful live measurements can be calculated and shared to users – such as voltage, current, number of faults, energy, and power consumption.

**MicroSD card:** The useful ADC measurements and calculations will be stored on the microSD card.

**GPS:** Needed as a clock to discipline the interrupt timings. Works to 1PPS accuracy, regardless of the geo-location

**Battery:** The platform should have a low enough realistic power consumption to run sufficiently on external battery cells

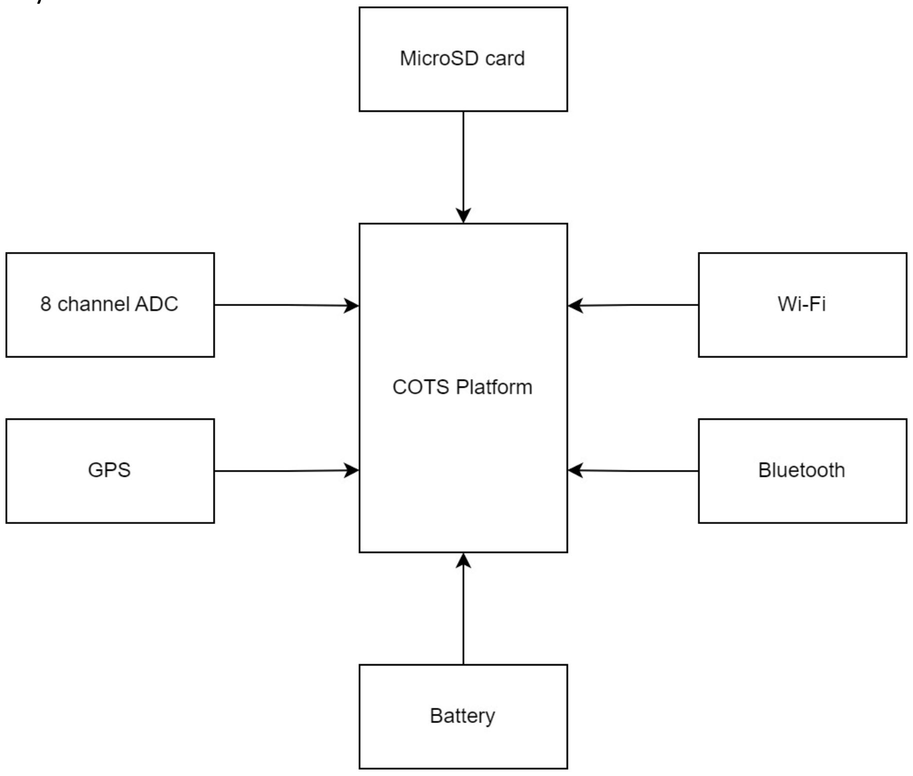


*Figure 1 - Block diagram to represent the requirements of the COTS solution that will facilitate the power supply quality monitoring and external communications*

## Signal Conditioning

In a power station the secondary side of the distribution transformer can be expected to be around 110V [6]. 110V is much a greater potential difference than what a standard COTS Platform ADC module can take as an analogue input. Most standard ADCs are unipolar and take analogue inputs of a range between 0 – 5V roughly.

This means that the 3-phase AC voltage inputs will need attenuated down to between 0 – 5V (depending on the ADC). Since the analogue inputs are AC waveforms, the inputs will also need to be shifted so that both the positive portion and the negative portion of the transient wave fit within the 0 – 5V range. The midpoint of the waveform should sit at 2.5V so that the wave sits perfectly in the center. The input signal is set at 1kHz for now – it will be set to the mains freq (50Hz) in semester 2.

The amplitude of the attenuated signal must be less than 2.5V and hence, a nice VRMS value of 1 would give a Vpeak value of sqrt(2) = 1.414V, which fits perfectly within the bounds.

To attenuate the analogue input signal down from 110V to 1.414V, an inverting op-amp circuit would be the perfect choice. The resistor values can be worked out as follows:

$$Gain = \frac{Vout}{Vin} = -\frac{Rf}{Rin}$$

Gain is negative for the inverting op-amp as it produces a negative output voltage compared to the input due to it being 180 degrees out of phase.

$$Gain = \frac{\sqrt{2}}{110} = \frac{1}{Rin}$$

Rf set to 1kΩ as a baseline to find value for R2 from there

$$Rin = \frac{110}{\sqrt{2}}$$

$$Rin = 77.78k\Omega \approx 78k\Omega$$

*Figure 2 - Signal conditioning gain equation*

In Figure 3, shown below, the 78kΩ resistor is used for Rin and a 1KΩ resistor is used for Rfb, as proven in the calculation previously. This theoretical 741 op-amp is being powered by positive and negative 12V supplies.

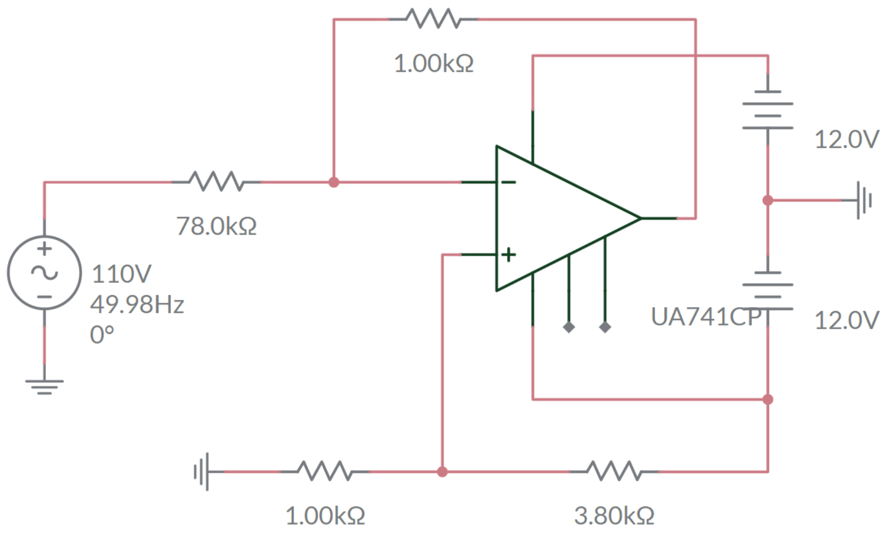


*Figure 3 - Schematic diagram of the inverting op-amp attenuation circuit.*

To obtain the 2.5V offset as required for the ADC range, a voltage divider circuit was coupled to the 741 positive supply voltage so that 2.5V is being supplied to the non-inverting input of the 741 op-amp. This will hence offset the output voltage by 2.5V. The resistor values needed for this voltage divider circuit were worked out using the calculation, as shown below in Figure 4.

$$Vout = Vin \ \frac{R2}{R1 + R2}$$

$$Vout\ (R1 + R2) = Vin \times \ R2$$

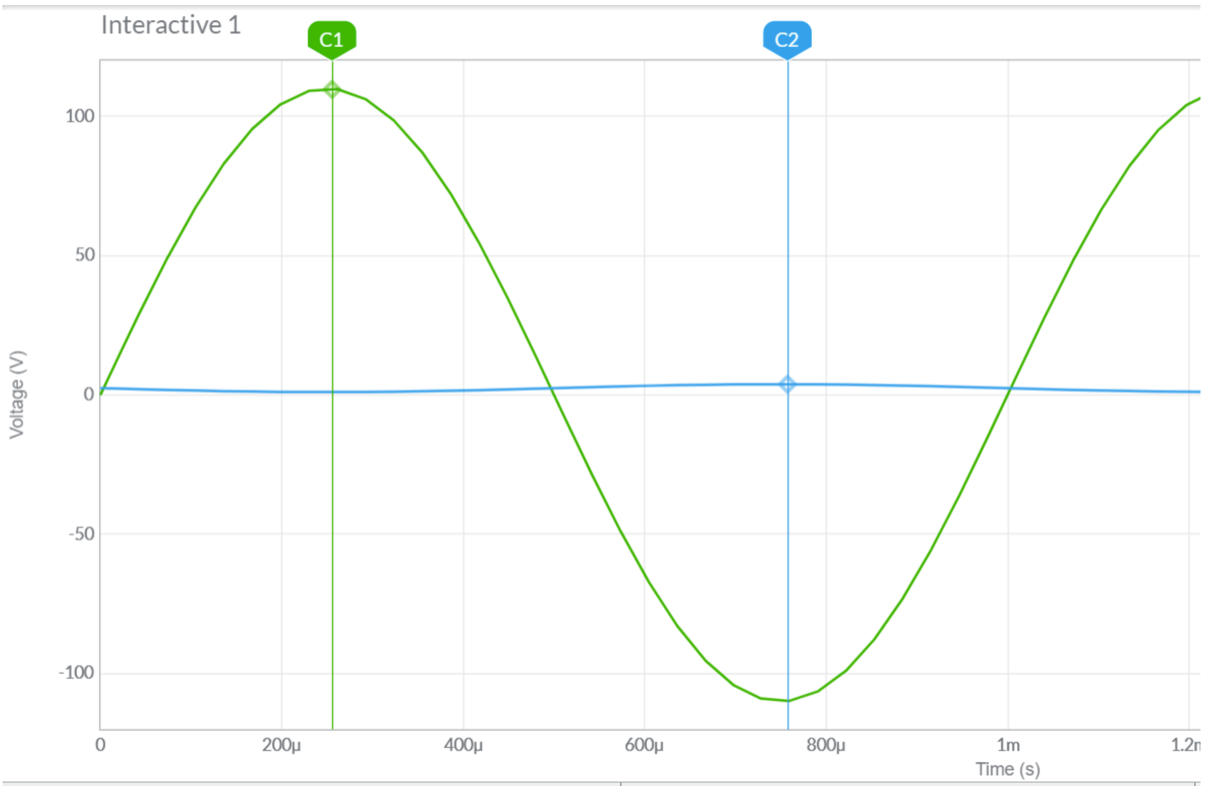
$$R1 = \frac{Vin \ \times R2}{Vout} - R2$$

$$R1 = \frac{12 \times 1}{2.5} - 1$$

$$R1 = 3.80k\Omega$$

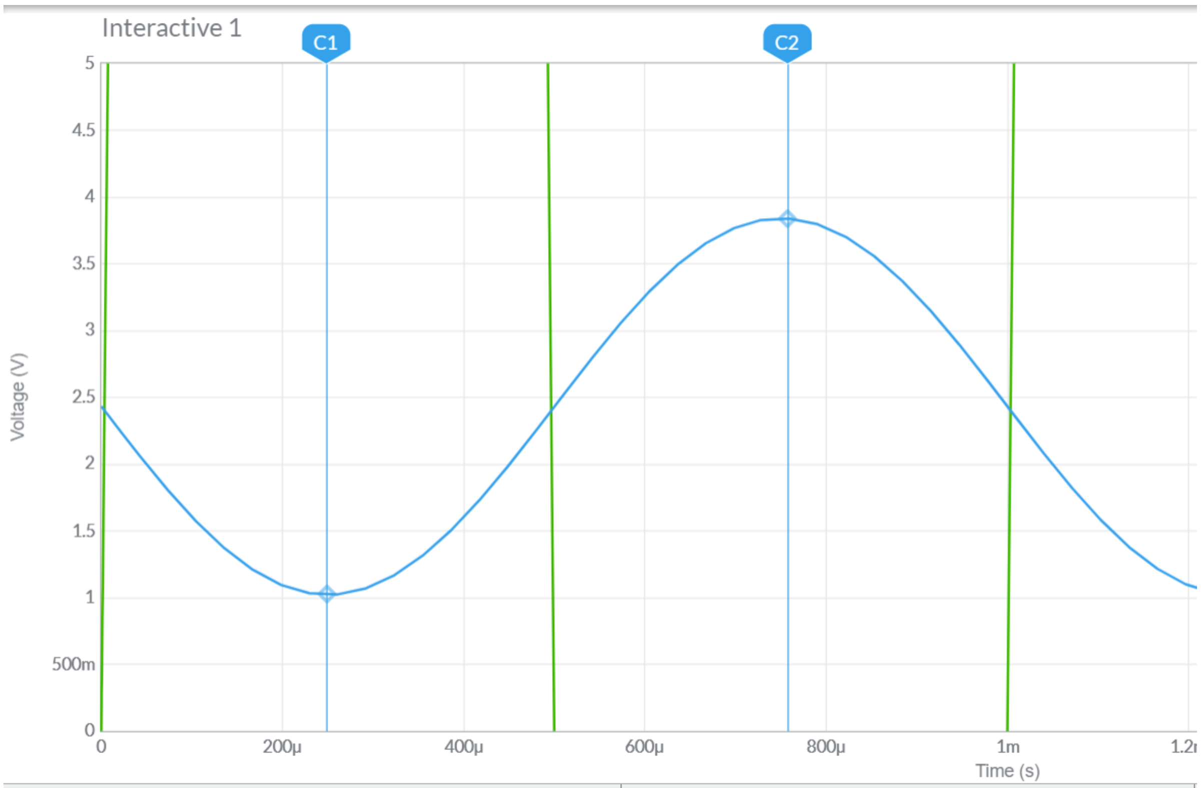
*Figure 4 - Voltage divider equation for calculating op-amp r1 value*

Shown below in Figures 5 and 6, are two grapher images of the transient waveforms outputted by the schematic. The green line represents the AC waveform for Vin, which has a Vp of 110V. The blue line represents the AC waveform for Vout which has a midpoint of 2.5V, a maximum voltage of 3.84V (3sf), and a minimum voltage of 1.03V (3sf). This gives a Vpp value of 2.81 which is approximately 2 x $\sqrt{2}$. Therefore, the schematic circuit output is as expected – it produces a Vout signal of Vpeak = $\sqrt{2}$ with an offset (and hence midpoint) of 2.5V. Figure 5 captures both the Vin and the Vout waveforms, whereas Figure 6 is magnified on the y-axis to give a more clear view of the Vout waveform.



| 🟩 Cursor 1 | 🟦 Cursor 2 |
|---|---|
| 255.26 μs, 109.63 V | 757.36 μs, 3.8379 V |

*Figure 5 - Grapher image of the transient waveform produced by the schematic diagram in Figure 4. This grapher image compares the maximum Vout value to the maximum Vin value.*



| 🟦 Cursor 1 | 🟦 Cursor 2 |
|---|---|
| 249.13 μs, 1.0269 V | 757.36 μs, 3.8379 V |

*Figure 6 - Grapher image of the transient waveform produced by the schematic diagram in Figure 4. This grapher image compares the maximum Vout value to the minimum Vout value.*

## Choosing the optimal COTS platform

There does not exist a single COTS Platform on the market that has all of the 6 required functionalities already built in. Therefore, external breakout boards can be used to facilitate any needed functionality.

At a bare minimum, the platform must be powerful enough to process all the various inputs, communications, calculations, and processes that will be needed as part of the project. It must also have enough pins to support all the various inputs. To narrow down the platforms a bit further, it would be ideal to use a board that already has Wi-Fi and Bluetooth connectivity built in. This is not essential but is an added bonus as it reduces unnecessary complexities.

A crucial decision had to be made between using a **microcontroller (MCU)** or a **single board computer (SBC)** for this project. The best-suited MCU and best-suited SBC were finalised upon and then compared.

## The best-suited single-board computer (SBC) option

Out of the Raspberry Pi SBC platforms, the **Pi Zero 2 W** was found to be the most suitable platform for this project. This is thanks to its best-suited processing potential per size ratio (compared to the other models), which will be able to handle all the inputs/processing/outputs required as part of this project. It also includes a built-in MicroSD card module which is an added bonus as it would remove the need for an external MicroSD card module.

The CPU in the Pi Zero 2 W is 5 times faster and has 40% faster single-core performance [7] than the Pi Zero W so it would be worth the extra £7 to go for the Zero 2 W over the old model Zero W.

The comparison table of the different Raspberry Pi SBC model specifications can be seen in Appendix 3, Figure 114.

## The best-suited microcontroller option

Microcontrollers from Arduino, Teensy, Espressif Systems, Beaglebones, and other manufacturers were deeply reviewed. After thorough research, the best-suited MCU from Arduino came out to be the **Arduino Nano RP2040 Connect**. Out of all the other manufacturers, the best microcontroller came out the be the **ESP32.**

Both of these 2 microcontrollers would be great options for the project due to their extremely good power/size ratio, however, there are a couple of key differences that set them apart:

- The Nano RP2040 has a slightly lower clock speed but it has a higher processing power due to the latest ARM architecture
- The Nano RP2040 came out in 2021 compared to the ESP32 coming out in 2016. Therefore, the ESP32 has extensive support available and is less buggy because of this
- The Nano RP2040 has a RAM of 264KB and ROM of 16MB compared to the ESP32 RAM 320KB and ROM 448KB
- ESP32 is much more power efficient – it also has ultra-low power management features

ESP32 is more suited to applications that require:

- Wi-Fi/BLE
- Some GPIOs
- Lower processing power
- Battery power source

Nano RP2040 is more suited to applications that require:

- Slightly higher processing power
- Multiple input peripherals
- Higher memory
- Wearable applications

Both of these MCUs have adequate processing power for this project and would make a great platform choice. Overall, the ESP32 would be a more suitable MCU for the project due to its much lower power consumption (which is crucial for this battery-powered sensor), extensive support available since it has been out for 6 years, its lower cost, and the fact that it doesn't have all the unneeded sensors onboard such as the microphone, etc. [8]

The comparison table of the different Arduino model specifications can be seen in Appendix 2, Figure 113.

## Best MCU vs best SBC – (ESP32 vs Raspberry Pi Zero 2 W)

Both the ESP32 and Raspberry Pi Zero 2 W are the best possible MCU and SBC options respectively.

The ESP32 is a dual-core microcontroller that generally runs sequential code in a do-while loop whereas the Pi Zero 2 is a quad-core general-purpose computer that runs under an operating system (OS) and hence can have interrupts due to OS priority suspensions. The overlaying operating system found in SBCs adds complexity but allows multiple tasks to be run at once due to the concurrent nature, whereas the MCUs are mostly sequential in nature and are intended to run one program repeatedly. [9]

To reduce complexities and to be as efficient as possible with available resources eg. battery power – it would make sense to use the more powerful, complex, and resource-intensive SBCs only if it is absolutely necessary i.e. if the MCU couldn't handle the required load. Therefore after detailed research, a powerful MCU such as the **ESP32** should be more than capable of handling the ADC sampling, Wi-Fi, BLE, MicroSD card storage, GPS interrupt timing syncing, etc. that is required by the chosen platform. If one ESP32 is insufficient to handle all of these processes at once, a second ESP32 could be used in tandem, with one for ADC sampling and the other for the rest of the processes.

## ESP32 Devkit model choice

ESP32 chips are created and developed by Espressif but a range of different companies produce ESP32 devkits (using Espressif's ESP32 chip) such as Adafruit, Sparkfun, and a range of other companies [10]. Since Espressif actually create the ESP32 chip, a devkit directly developed by Espressif was the best option for this project as it guarantees the best reliability, best documentation, largest userbase, and hence will keep complexities to a minimum.

The ESP32-DevKitC-32UE (ESP32-WROOM-32UE) was the devkit of choice due to its ease of use, built-in pinout, and low footprint – which is important for preserving battery life. The UE model was specially chosen compared to the E model so that an external U.FL antenna could be connected rather than relying on the E board’s weak internal antenna.

The full specifications and pinouts of this ESP32 devkit can be seen in Appendix 5, Figure 18.

## Choice of breakout boards

The specifications for each of these breakout boards can be found in Appendices 6 - 8, along with photos of each board in Figures 19 – 21.

### 8-Channel Analogue to Digital Converter Breakout

The ESP32 and most other MCUs have a built-in ADC. The ESP32 has 18 possible ADC channels, however, the ESP32 only has one actual hardware ADC so the channels must be multiplexed rather than read asynchronously [11]. An external multi-channel ADC could be used to measure multiple channels both asynchronously and accurately, as required.

A 16-bit ADC has 15 usable bits – which is a great resolution. There aren't very many 8-channel 16-bit ADCs on the market, so the best option found was to use two Adafruit ADS1115 4-channel ADCs and then connect them both up over I2C at a maximum sps of 860 samples/second. This ADC is unipolar and has an input voltage range of -0.3 – 5.3V (when powered by a 5V supply) which is perfect for the conditioned signal inputs.

### MicroSD Card Breakout

The Adafruit MicroSD card breakout board+ came out to be the best option for the MicroSD card module due to its renowned reliability and available open-source software. The software was originally designed for SDHC MicroSD cards, however, an SDXC MicroSD card was ordered to use with the breakout initially by mistake. The maximum size of SDHC cards is 32GB so a 32GB card was ordered at a later stage. This MicroSD card breakout board communicates with the ESP32 via SPI communications, which offers a much faster solution than "bit-banging" using other pins. [12]

### GPS Breakout

The U-Blox NEO-6m was the best available GPS breakout board on the market due to its size, popularity, and renowned reliability. This particular GPS module communicates with the ESP32 using the TX and RX serial communication pins.

| Brand | Component | Wholesaler | Individual Price (inc VAT) | Notes |
|---|---|---|---|---|
| Espressif Systems | ESP32-DevKitC-32UE (ESP32-WROOM-32UE) | Mouser | £10.56 | Free shipping over £33 |
| Adafruit | MicroSD adapter breakout board – 254 (485-254) | Mouser | £7.92 | Free shipping over £33 |
| Adafruit | ADS1115 16-Bit ADC – 4 Channel with Programmable Gain Amplifier | Coolcomponents | £14.89 | Excludes £2.99 shipping |
| U-Blox | GPS Module GY-NEO6M NEO-6M + Antenna | 247Geek | £6.99 | Excludes £2.00 shipping |
| SanDisk | Ultra microSDXC UHS-I memory card 128 GB + adapter | Amazon UK | £12.99 | Free shipping |

*Figure 7 - Bill of Materials order that was sent on 28th November 2022*

## Plan of action for development

On arrival of all the separate components, there were many possible orders of development that could have been taken when working on this project. The most logical and efficient way to work through this project would be to go through, component by component, and get them all working individually before trying to combine their functionalities together. This builds the understanding of each component better, allows more room for trial and error, and makes debugging/problem-solving exponentially easier. Figure 8 shows an updated block diagram of the COTS platform and breakout boards, specifically for the ESP32 MCU. (The specific battery model will be chosen at a later stage in the project).

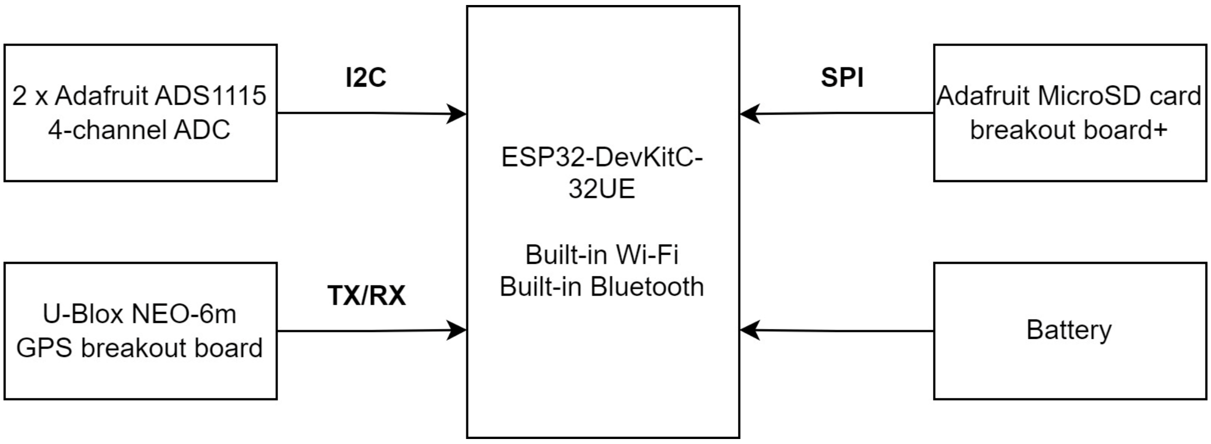

2 x Adafruit ADS1115 4-channel ADC
I2C
U-Blox NEO-6m GPS breakout board
TX/RX
ESP32-DevKitC-32UE
Built-in Wi-Fi
Built-in Bluetooth
SPI
Adafruit MicroSD card breakout board+
Battery

*Figure 8 - Updated block diagram of the COTS platform and breakout boards, along with each breakout's communication format*

Logically, it made the most sense to start the development with the ESP32 devkit since all the other components need to be connected to the MCU before they can actually be used and tested. From there, the breakout boards can be tested individually, then they can be tested in combination with each other as shown in the flow diagram order in Figure 9, below.

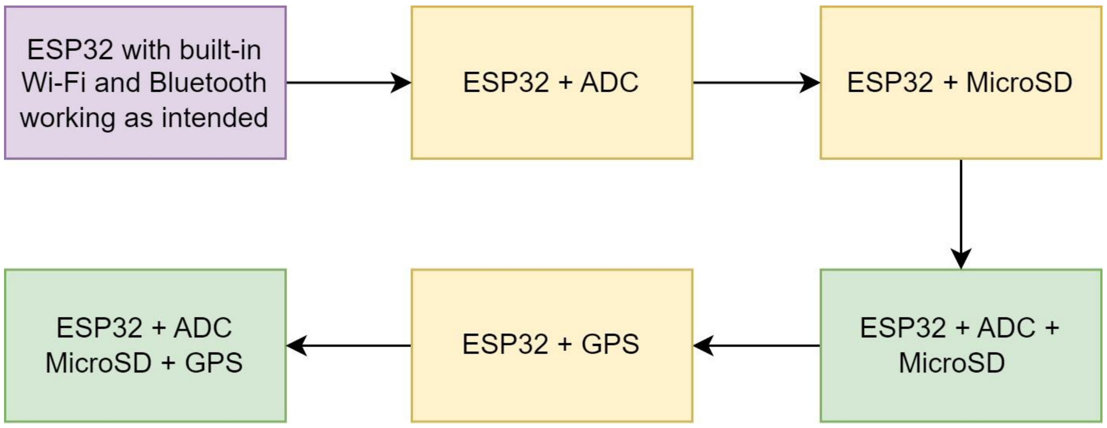

ESP32 with built-in Wi-Fi and Bluetooth working as intended
ESP32 + ADC
ESP32 + MicroSD
ESP32 + ADC + MicroSD
ESP32 + GPS
ESP32 + ADC MicroSD + GPS

*Figure 9 - Flow diagram order for development of ESP32 platform and breakouts*

## ESP32 Development

The ESP32 can be used with the Arduino IDE so the first goal was to get a simple LED flickering circuit working in the Arduino IDE.

To test the Bluetooth functionality of the board, a simple BLE server was set up on the ESP32 to allow data to be transferred to a smartphone in range.

To test the Wi-Fi functionality and antenna range of the board, a simple program was used to scan for nearby networks and display them in the serial monitor. A simple Wi-Fi server for turning on and off an LED was run on the board to test the IoT server functionality. An external antenna had to be connected to the U.FL port on the board to facilitate a strong enough connection to the local router to set up the server.

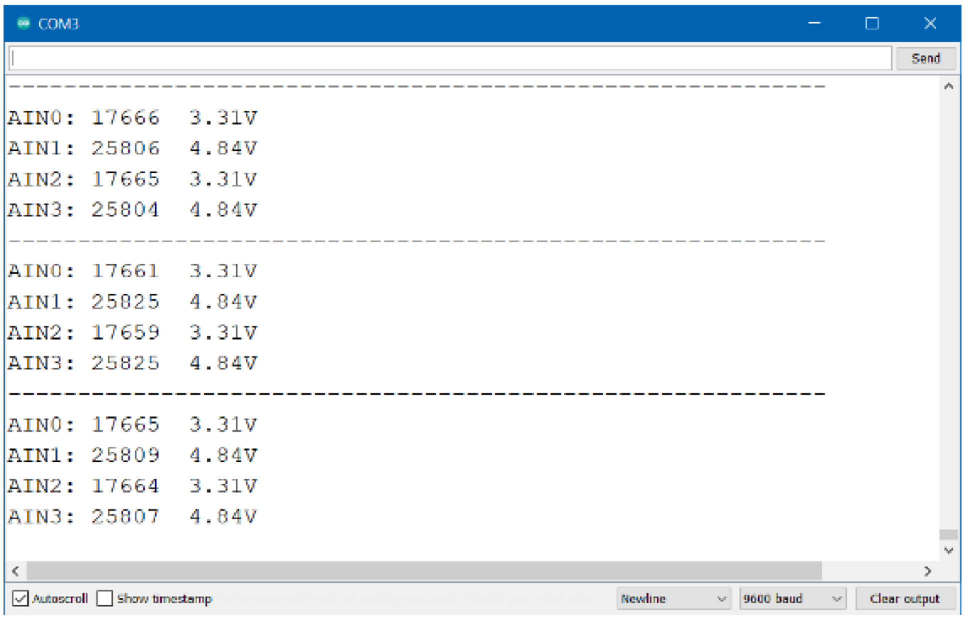
Once the ESP32 had been tested thoroughly, the next logical step involved getting the ADC breakout working with the ESP32.

## ESP32 + ADC Breakout

The Adafruit ADS1115 was connected via I2C to the respective I2C pins on the ESP32. I2C normally operates at 100-400kHz which is far beyond the samples per second (sps) needed for this project and hence the sps will not be limited by I2C, but by the maximum sps of the ADC. [13] The circuit diagram can be seen in Figure 15.

The ADS1115 can be powered by either a 3V or 5V supply. Since the analogue input range of the ADS1115 is -0.3 – (Vdd + 0.3), then a 5V supply must be used as a 3.3V supply would not provide a great enough range (-0.3 - 3.6V) for the conditioned analogue signals (which need a range of 1 – 4V).

A program was then engineered to take single-ended readings roughly once per second for all of the 4 channels, which were then printed out in the serial monitor. In the example shown below, the 3.3V DC output from the ESP32 was connected to pins A0 and A2 and the 5V DC output from the ESP32 was connected to pins A1 and A3, for demonstration purposes. The serial monitor output can be seen below, in Figure 10



```
-----------------------------------------------------------
AIN0: 17666   3.31V
AIN1: 25806   4.84V
AIN2: 17665   3.31V
AIN3: 25804   4.84V
-----------------------------------------------------------
AIN0: 17661   3.31V
AIN1: 25825   4.84V
AIN2: 17659   3.31V
AIN3: 25825   4.84V
-----------------------------------------------------------
AIN0: 17665   3.31V
AIN1: 25809   4.84V
AIN2: 17664   3.31V
AIN3: 25807   4.84V
```

*Figure 10 - Screenshot of the Arduino IDE Serial Monitor tool, displaying the ADC measurements for the 4 channels*

## ESP32 + MicroSD Card Breakout

The Adafruit MicroSD card breakout was connected via the hardware SPI pins on the ESP32. The hardware SPI pins offer a much faster solution than “bit-banging” the interface code by using another set of pins. [12]

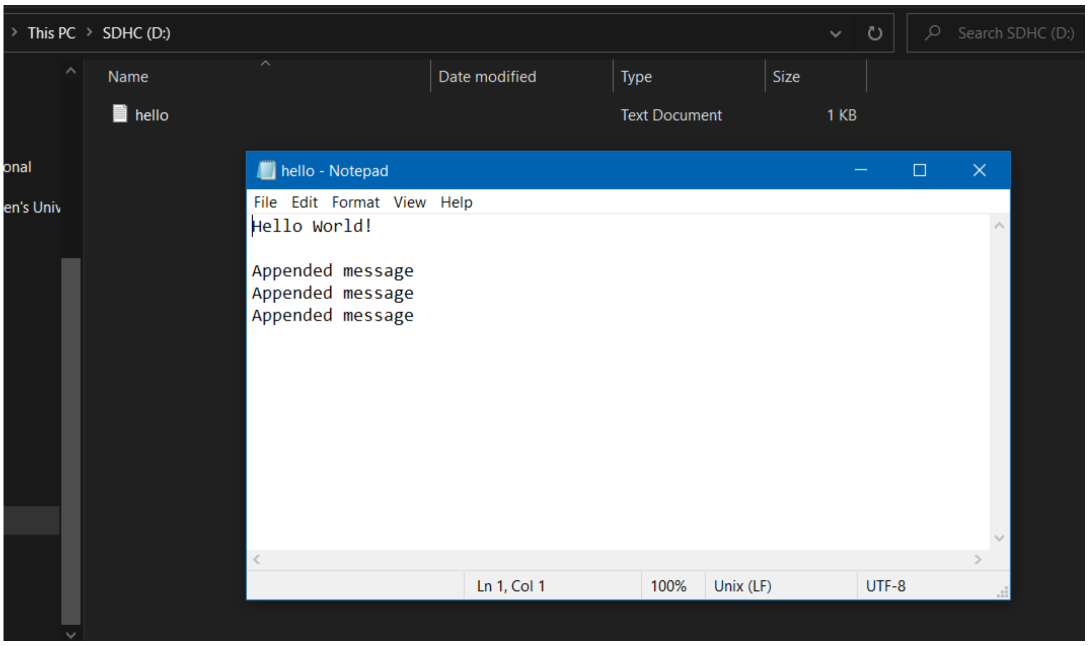
The MicroSD breakout board has proper 3V level shifting built in. The board is native at 5V so, to avoid unnecessary level shifting, a voltage of 5V from the ESP32 was supplied to the MicroSD card breakout.

The board itself can be used for tasks such as creating directories, listing directories, removing directories, writing files, appending files, reading files, renaming files, deleting files, and testing files IO. [14] This functionality of the board was all manually tested to ensure correct functionality.

A program was then engineered that creates a file then appends values to it. This functionality will be needed later for storing the ADC measurements. Figure 11 shows a snippet of the program and Figure 12 shows a screenshot of the MicroSD card contents that were created once the full program was run.

```
writeFile(SD, "/hello.txt", "Hello World!\n\n");
int counter = 0;
while (counter++ < 3) {
  appendFile(SD, "/hello.txt", "Appended message\n");
}
```

*Figure 11 - Snippet of the program code to demonstrate how the writing and appending of text files on the MicroSD works*



```
Hello World!

Appended message
Appended message
Appended message
```

*Figure 12 - Screenshot to show the contents of the MicroSD card after the program had been run*

Figures 11 and 12 demonstrate the basic MicroSD card functionality needed for storing the ADC measurements working as intended. The next logical step was to combine the functionality of the ADC and the MicroSD card breakouts.

## ESP32 + ADC Breakout + MicroSD Card Breakout

Once the individual ADC and MicroSD breakouts had been tested individually, their functionality was combined together into one single program. This program was engineered to take 4 single-ended voltage measurements from the ADC and then append these values to a text file on the MicroSD card. To simulate voltage signal inputs, the 3.3V output from the ESP32 was connected to A0 and A2 of the ADC and the 5V output from the ESP32 was connected to A1 and A3 of the ADC.

The full C++ code can be seen in Appendix 1. As the project progresses, the GPS module will be used as a clock for disciplining the timing interrupts to specify when the samples should be taken from the ADC. For now, however, this current program implementation doesn't consider timings. It takes the 4-channel readings and appends them to the text file on every loop iteration (averaging around 16sps). This program is a good step towards the later functionality of storing useful values on the MicroSD card, such as vrms/frequency/power/energy consumption, etc.

Figure 13 shows the breadboard circuit developed for use with this program. The MicroSD card breakout is shown on the left, the ESP32 in the middle, and the ADS1115 ADC on the right.

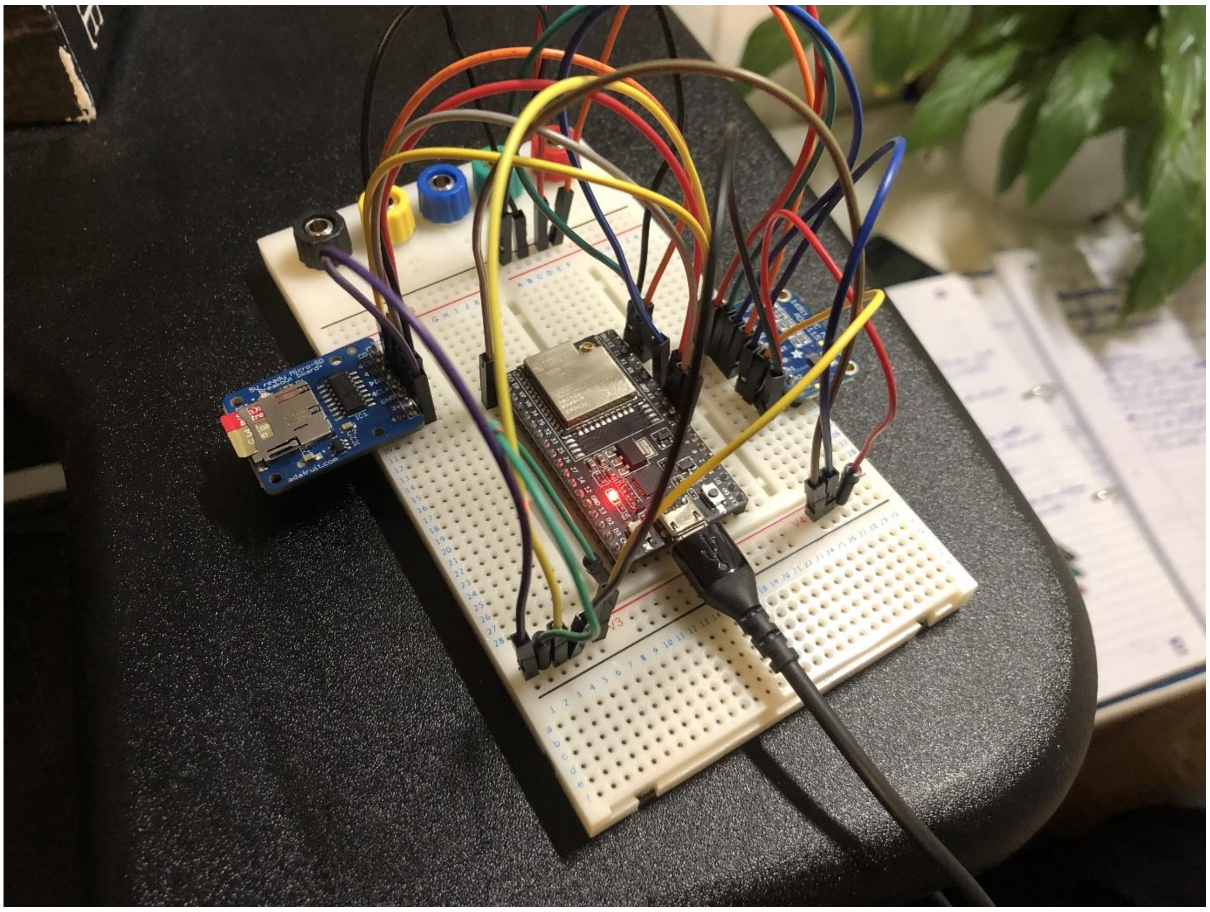
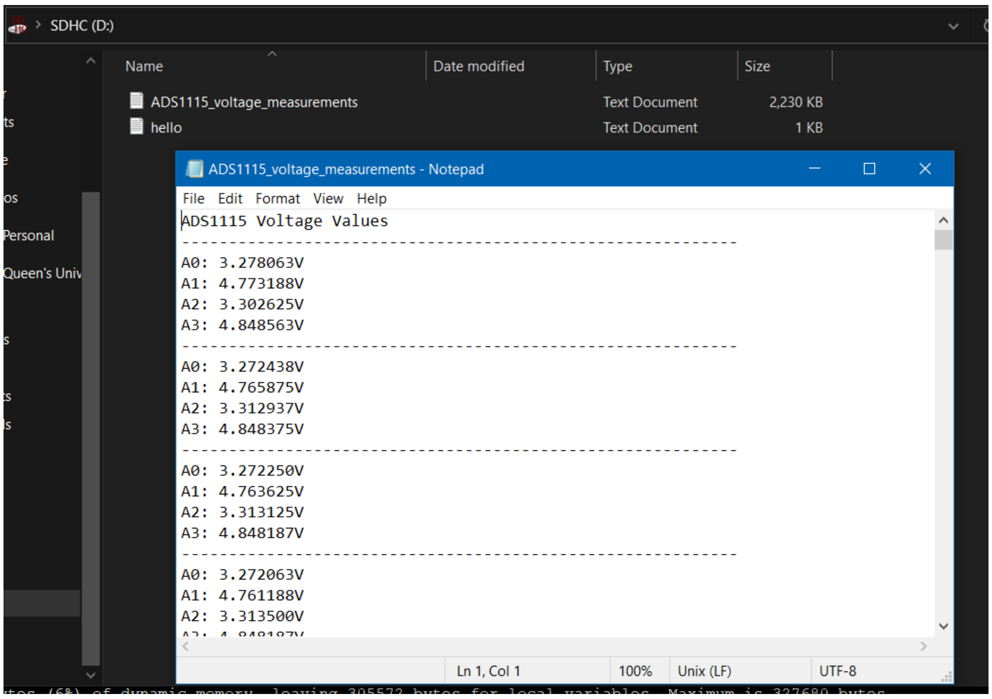


*Figure 13 - Photo of the ESP32 + ADC Breakout + MicroSD Card Breakout circuit used to store the 4 analogue input measurements on the MircoSD card*

Figure 14 shows a screenshot of the text files contained on the MicroSD card. It also shows the contents of the ADS1115_voltage_measurements text file, which contains the appended measurements outputted from the program, after the program had been run for a few minutes.


SDHC (D:)
Name
Date modified
Type
Size
ADS1115_voltage_measurements - Notepad
File Edit Format View Help
Ln 1, Col 1
100%
Unix (LF)
UTF-8

| Name | Type | Size |
|---|---|---|
| ADS1115_voltage_measurements | Text Document | 2,230 KB |
| hello | Text Document | 1 KB |

```
ADS1115 Voltage Values
-----------------------------------------------------------
A0: 3.278063V
A1: 4.773188V
A2: 3.302625V
A3: 4.848563V
-----------------------------------------------------------
A0: 3.272438V
A1: 4.765875V
A2: 3.312937V
A3: 4.848375V
-----------------------------------------------------------
A0: 3.272250V
A1: 4.763625V
A2: 3.313125V
A3: 4.848187V
-----------------------------------------------------------
A0: 3.272063V
A1: 4.761188V
A2: 3.313500V
```

*Figure 14 - Screenshot to show the content of the MicroSD card after the program had been run*

## Circuit Diagram

Figure 15, as shown below, shows the circuit diagram for the ESP32 + ADC + MicroSD circuit that was used previously in figure 13 – except for 1 slight change. Earlier, the ADC took 4 analogue inputs directly from the 3V and 5V output supplies of the ESP32. The ADC in this circuit diagram, however, uses the output voltage signal from the inverting op-amp attenuation circuit. In this signal conditioning circuit, the 741 op-amp is powered by a positive and negative 12V supply, however, this is not very practical due to size and power constraints. This will be fine-tuned at a later stage in the project.

The address pin of the ADS1115 is currently being set to GND. This address pin is used to set the address of the I2C connection (to 0x48), to allow up to 4 ADCs to be used simultaneously on the I2C, i.e. the second ADS1115 will need to have a different address set for it to be used in tandem with the first ADS1115 over the I2C interface..

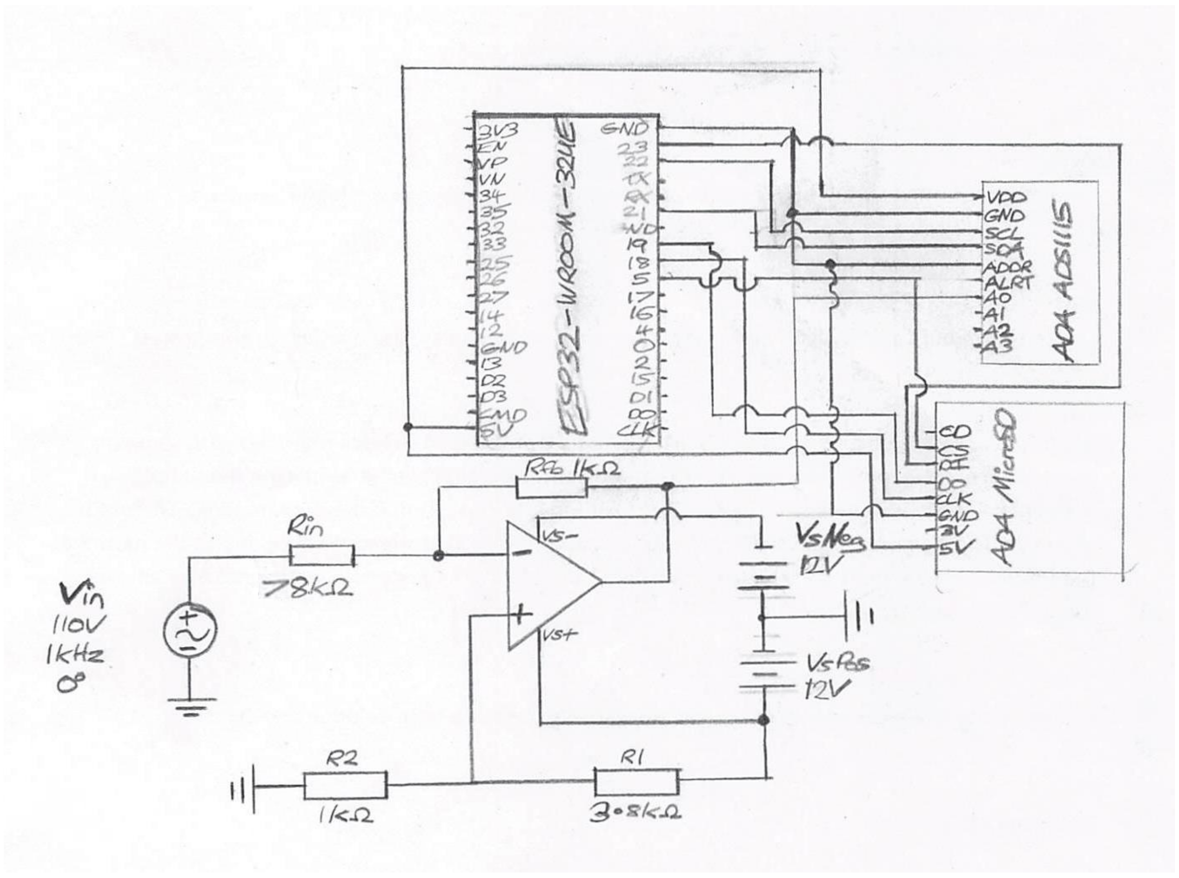


Figure 15 - A hand drawn circuit diagram for the ESP32 + ADC + MicroSD circuit that was used previously in Figure 13

## ADC Sampling Rate Problem

The UK mains frequency is around 50Hz [15]. Following Nyquist’s Theorem [16], when sampling the analogue inputs with the ADC, 24 samples per cycle should be an adequate number of samples to calculate accurate VRMS values, frequency values, etc.

$$f_s = N \times f_0$$

$$f_s = 24 \times 50$$

$$f_s = 1.2kHz$$

$$t_s = \frac{1}{f_s}$$

$$t_s = \frac{1}{1200}$$

$$t_s = 833\mu s$$

*Figure 16 - Nyquist's Theorem sampling rate equation for UK mains frequency of 50Hz*

Shown above, in Figure 16, are the calculations for determining the required frequency and time period of the ADC samples. It was calculated that the ADC should take 1200 samples per second (sps), i.e. it should take samples every 833µs.

The ADS1115 has a high resolution of 16-bits (15 usable bits) but as a result, it only has a maximum sample rate of 860sps. This maximum sampling rate of 860sps is not high enough to be able to take the 1200sps readings that are required for this project and hence a faster sampling rate ADC is needed.

The ADS1015 ADC has a slightly lower resolution of 12 bits (11 usable bits) but has a much higher maximum sampling rate of 3300sps, hence facilitating the 1200sps measurements that are needed for the 50Hz signal. Two Adafruit ADS1015 ADC breakouts were researched and then ordered.

## Future work to be completed in semester 2

The ESP32 circuit is at the stage where 4 single-ended inputs measurements are read and then stored on the MicroSD card.

- The next step will be to wire up the GPS breakout individually with the ESP32, to grasp how the breakout works and to test that the desired functionality works as expected separately
- After that, the GPS breakout should be included in the ADC & MicroSD circuit then use the GPS's 1PPS timing functionality as a clock to discipline the timing interrupts for taking the readings, at 1sps to begin with.
- The circuit should then be amended to use the ADS1015 instead of the ADS1115 then try to get 1200sps readings working successfully – as needed for a 50Hz signal with 24 samples per cycle.
- The program should be altered to work out the VRMS from those measurements, then store the values to the MicroSD card at appropriate regular intervals.
- Using the samples taken and basic trigonometry, the two true zero points within each cycle should be calculated and then used to work out the true frequency of each cycle.
- Using these frequency measurements, the energy consumption and power can be worked out and stored on the MicroSD card too. The frequency should be roughly 50Hz but the true frequency must be worked out to determine accurate energy consumption and power statistics as required for metering/billing.
- The signal conditioning circuit currently takes in a voltage input of 110V, to simulate the secondary side of the distribution transformer. In the development process, using a lower voltage input such as 1/2V would be more practical so the signal conditioning circuit should be amended to facilitate this lower voltage input

Once all the desired useful measurements are being read, calculated, and stored correctly locally, the external communications should be set up.

- First, a Wi-Fi server should be set up on the ESP32 to host a static web page for displaying the near-live-time useful measurements.
- Hosting a Wi-Fi server requires a constant/stable Wi-Fi connection so this may be unsuitable for many substation scenarios. A BLE server doesn't need a separate network, can communicate with most modern smartphones, and has a very low energy usage [17]. Users in the substation in close enough proximity to the ESP32 could access the near-real-time readings through their smartphone using BLE. This may be a superior alternative to a Wi-Fi server – this will need further research into to compare and decide on the best solution.
- Hosting a Wi-Fi/BLE server while taking the ADC readings at a high frequency may require too much processing power for a single ESP32 to handle. If so, a second ESP32 could be used solely for taking the high-frequency ADC measurements while the other ESP32 focuses on the measurements and external communications.

Once the external communications of the smart wireless sensor have been set up successfully, there is additional bonus functionality that could be implemented to further improve the functionality of the smart sensor.

- A suitable battery solution could be designed and developed to support the portable functionality of the platform and signal conditioning circuit
- The useful measurements could be stored in a suitable cloud database storage provided by services such as Amazon Web Services (AWS) or Google Cloud Platform (GCP), etc.
- Even further, a static website could be hosted on these cloud platforms to display the readings online, to be accessed worldwide.
- Rather than using a breadboard, a more semi-permanent solution such as Veroboard Stripboard could be used
- A suitable model case could be designed in CAD modeling software and 3D-printed to hold all of the circuitry and components

# Semester 2

## ESP32 + GPS Breakout

After the ESP32, ADC and MicroSD breakout circuit was working as intended, the next logical step was to get the ESP32 working with the GPS breakout. The U-Blox NEO-6m was hooked up to the ESP32 via UART (TX/RX). UART is a very simple communication form, however, a bug was found with the ESP32 devkit - if the GPS breakout was connected to the ESP32 default tx/rx pins then the code would fail during the upload process to the MCU.

This was solved by manually setting-up the UART serial communications with different tx/rx pins than the default ones. Figure 17, as shown below, showcases the line of code used to set up the GPS serial communications using pins 16 and 17 as the RX and TX pins of the ESP32, respectively. The ESP32 general serial communication line with the PC was set up with a baud rate 2000000Hz, then communications with the GPS breakout were set up at a baud rate of 9600Hz, as shown below.

```
Serial2.begin(9600, SERIAL_8N1, 16, 17);
```

*Figure 17 - Line of code to initialise the secondary serial, for the GPS breakout UART communications*

What data does the GPS breakout send to the ESP32?

The GPS breakout sends National Marine Electronics Association (NMEA) sentences [18] to the ESP32 via UART. Each of these NMEA sentences start with a "$" symbol and the data fields are separated by commas. The TinyGPSPlus library [19] was utilised to easily extract usable data from these NMEA sentences. Figure 18, as shown below, showcases the serial monitor output which includes 5 NMEA sentences and some corresponding extracted data from those sentences – time, latitude and longitude.

```
--------------------------
$GPRMC,122351.00,A,5435.17211,N,00556.24868,W,0.588,,060423,,,A*67
$GPVTG,,T,,M,0.588,N,1.088,K,A*27
$GPGGA,122351.00,5435.17211,N,00556.24868,W,1,08,1.53,62.8,M,52.5,M,,*7C
$GPGSA,A,3,04,25,28,31,29,05,18,20,,,,,2.73,1.53,2.26*0B
$GPGSV,4,1,13,02,51,0

Time: 12235100
Lat: 54.59
Long: -5.94
```

*Figure 18 - NMEA sentence response sent over UART to the ESP32, along with useful extracted metrics*

## GPS breakout for ADC timing disciplining

Using a GPS breakout for disciplining/syncing the ADC interrupt may seem like a strange choice at first, however, GPS breakouts are unique in the fact that the 1PPS signal sent from them is perfectly synced, no matter the geo-location. The NMEA sentences are send via UART to the ESP32 perfectly at the start of each GMT second. This in-sync 1PPS signal of the GPS breakout can then be utilised to ensure that multiple smart wireless sensors will be taking readings perfectly in sync, as will be explained in more detail later.

When working with the GPS breakout, a mistake was made of trying to utilise the actual extracted NMEA time for the ESP32 timing interrupt disciplining. After some self-education on timing interrupts, it then became clear that the 1PPS signal itself coming from the GPS should be utilised for the interrupt syncing.

To test and validate the 1PPS interrupt timing functionality of the GPS, a timing interrupt was set up on the ESP32. This timing interrupt was set up to take ADC readings on every falling edge of the signal hitting that pin. The TX line of the GPS breakout was connected to pin 25 (yellow wire) so that a burst of falling signal edges would hit the interrupt pin once every second, hence causing an ADC reading to be taking every second. This circuit can be seen below, in Figure 20.

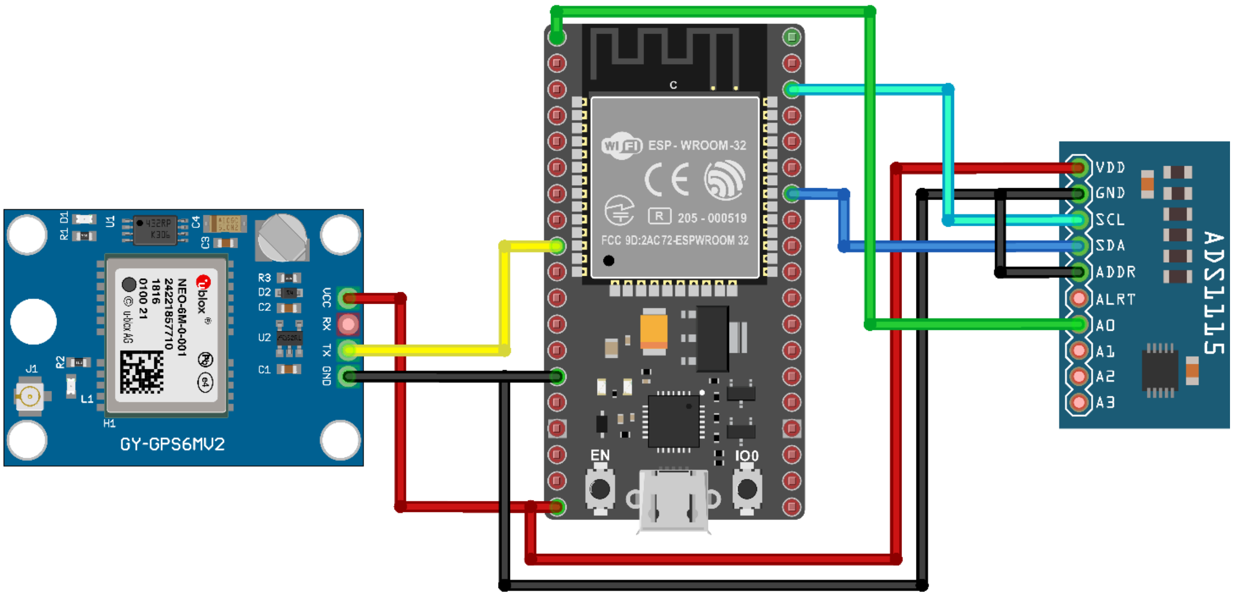
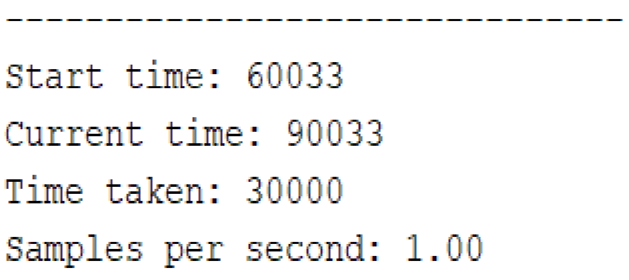


*Figure 20 - ESP32, ADS1115 and NEO-6M Fritzing circuit diagram*



```
-------------------------------

Start time: 60033
Current time: 90033
Time taken: 30000
Samples per second: 1.00
```

*Figure 19 - Serial Monitor output of the GPS ADC benchmark program*

The Serial Monitor screenshot, as shown above in figure 19, shows that the GPS timing interrupt circuit was able to successfully achieve 1PPS ADC sample readings. In the end-goal circuit, the GPS won't be used in this exact way but this circuit proved that the GPS 1PPS signal is extremely accurate and will facilitate the disciplining of the ADC timing interrupts perfectly.

The full details of how the GPS breakout is used for interrupt timing in the final circuit is explained in later sections.

## ADS1015 for 1200sps readings

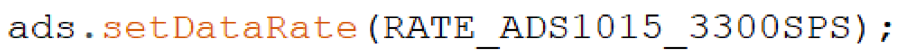
As explained in the Semester 1 section, the UK mains frequency is roughly 50Hz and 24 samples should be taken per cycle (following Nyquist's Theorem). [16] This calculates to 1200 samples needing to be taken every second (1200sps). As explained previously, the ADS1115 only has a maximum sampling rate of 860sps.

Two Adafruit ADS1015 ADCs were ordered and the ADS1115 in the circuit was swapped out for one of the ADS1015s. When setting up the 1200sps interrupt timer, duplicate readings could possibly have been taken if the ADC was unable to keep up with the 1200sps sampling, therefore, it was deemed necessary to thoroughly benchmark the ADC1015 to try to achieve its maximum sampling rate.

Single-ended readings are used by the MCU to take ADC readings every so often, however, they are inefficient and slow as it was later found that they have an in-built delay that cannot be overridden. Continuous mode is the ADS1015's fastest mode. Once the ADC has taken a new reading, the alert pin on the ADC sends out a high signal – if this alert pin is hooked up to an interrupt pin on the ESP32, the alert signal will tell the ESP32 when the ADC is ready to take another reading. Using continuous mode in this way ensures that the fastest possible rate of unique values can be read from the ADC at one time.

The ADC was thoroughly tested in continuous mode with the alert pin hooked up to an interrupt on the ESP32 to achieve the maximum sampling rate possible for the ADC. The ESP32 was timed for how long it took to take 30,000 samples and the results were printed in the serial monitor. In this testing period a maximum rate of 1700sps was achieved with ease, but it took a substantial effort investment to achieve the final sampling rate.

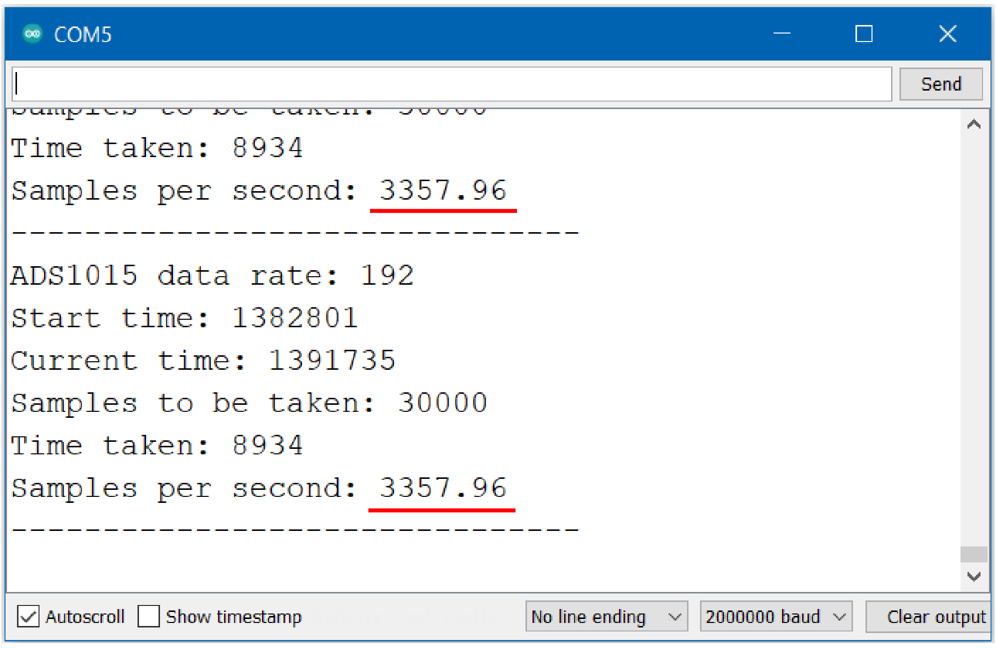
The first limiting factor of the continuous measurements was the default data rate. By modifying Adafruit's ADS1015 library to include useful additional functions, this default value could now be set using the simple line shown below, in Figure 21.



```
ads.setDataRate(RATE_ADS1015_3300SPS);
```

*Figure 21 - C++ code used to set to ADS1015 sampling rate to 3300sps*

The second main limiting factor of the ADS1015 was eventually found to be the I2C bus frequency. The ESP32 I2C bus clock speed is set to 100kHz by default, however, through modifying the Adafruit library, the I2C clock speed was increased to 400kHz which facilitated the ES32 and ADC to reach their final maximum speed.



```
Time taken: 8934
Samples per second: 3357.96
-------------------------------
ADS1015 data rate: 192
Start time: 1382801
Current time: 1391735
Samples to be taken: 30000
Time taken: 8934
Samples per second: 3357.96
-------------------------------
```

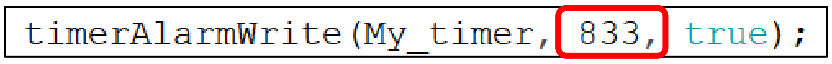
*Figure 22 - Serial monitor output of the final ADC benchmark speed test*

Shown above, in Figure 22, is the serial monitor output of the final ADC speed test. It can clearly be seen that a sampling rate of 3356sps was consistently taken through the continuous readings.

As shown in the previous program, the ADS1015 can successfully take readings at a sampling rate of up to 3300sps. This proves that the ADC will be able to take unique readings at 1200sps without any duplicate readings – ensuring that the ESP32 only retrieves new readings rather than re-reading the same ADC reading twice.

The next step was try to achieve the desired ADC sampling rate of 1200sps. To do so, interrupts were again utilised but this time the interrupts were to be triggered by ESP32 built-in clock. The ESP32 has four 64-bit timers [20]. The timer interrupt was specified at 1200sps by passing the specified time period as a parameter to one of the function calls, as shown in Figure 23 below.

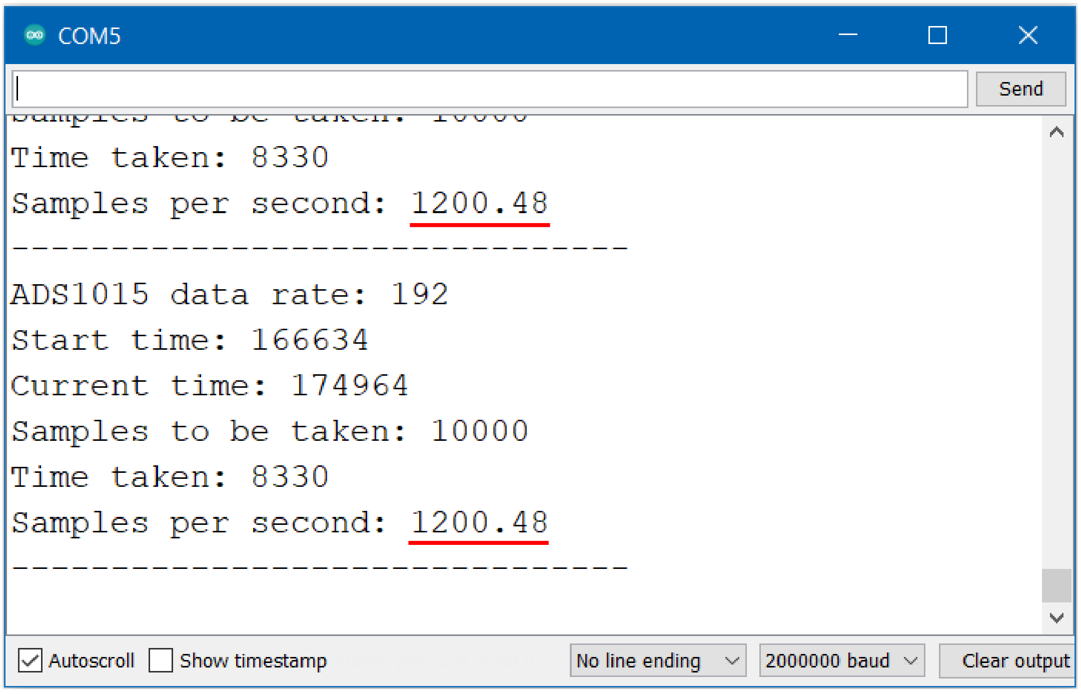
The interrupt ADC reading code was then updated from previously using the alert pin from the ADC



```
timerAlarmWrite(My_timer, 833, true);
```

*Figure 23 - C++ code used to set up the ESP32 interrupt timer at a sampling rate of 1200Hz (833us time period)*

for interrupts to using the ESP32 interrupt timer, as shown above in Figure 23. In summary, this meant that the ESP32 should take readings from the ADC every 833us, hence achieving 1200sps. This was tested thoroughly and a benchmark screenshot can be seen below in figure 24.



```
Samples to be taken: 10000
Time taken: 8330
Samples per second: 1200.48
-------------------------------
ADS1015 data rate: 192
Start time: 166634
Current time: 174964
Samples to be taken: 10000
Time taken: 8330
Samples per second: 1200.48
-------------------------------
```

*Figure 24 - Serial monitor output of the ADC sampling rate when the interrupt timer is set to 1200sps*

## ESP32 Input Analog Signal

Now that 1200sps ADC readings were being consistently taken, it was time to find an analogue signal input source. Up until this point in the project, the ESP32 5V and 3.3V DC outputs were being used as inputs into the ADC. Since these signals are DC signals, the output waveforms plotted by the ADC were horizontal lines. The end goal AC source was intended to use a mains supply/function generator, however, in the meantime the ESP32 digital to analogue converter (DAC) was set up and utilised as an input into the ADC. The external DacESP32 library [21] was utilised to obtain a consistent sine/cosine waveform with just a couple of function calls. The DAC output pin was set at pin 25 and the analogue output signal was set to a cosine wave with 50Hz frequency. Some of this code can be seen below, in Figure 25.

```
DacESP32 dac1(GPIO_NUM_25);
void setup(void)
{
  dac1.outputCW(50);
```

*Figure 25 - ESP32 code that utilises the DacESP32 library to setup a DAC cosine waveform output (of supposed frequency 50Hz) to pin 25*

The library was extremely valuable to this portion of the project, however, the frequency of the output waveform was not as precise as it was made out to be. When setting a frequency of 50Hz, the actual frequency of the analogue waveform was tested and calculated to be around 51Hz, which is a substantial difference. Setting an accurate frequency with this, however, was not quintessential as the frequency algorithm would be later be validated through simulated waveforms in MATLAB, as will be shown at a later point in the project.

Figure 26, as shown below, showcases the Serial Plot output (in the Arduino IDE) of the ESP32 DAC 51Hz cosine analogue waveform. These ADC readings were taken at 1200sps and the plot shows a lovely, consistent cosine waveform being sampled from the ADC. Through thorough testing, it was validated that there were no duplicate values or random zero values being read from the ADC.

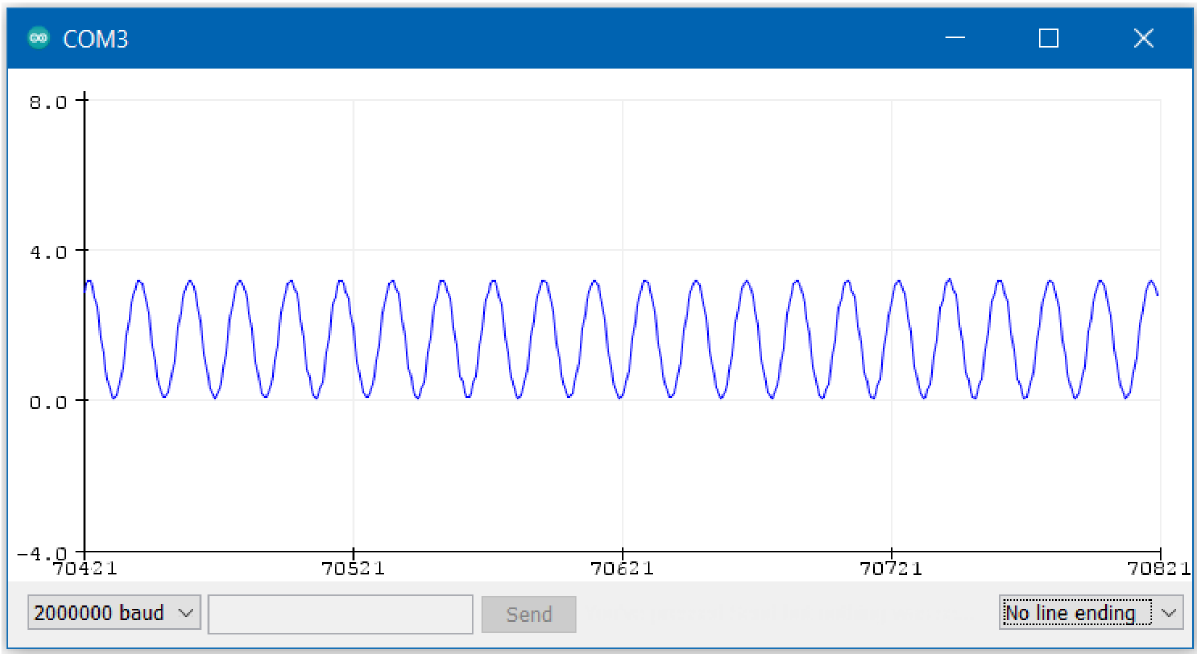


*Figure 26 - Serial plotter output of the ESP32 51Hz DAC waveform being read at 1200sps by the ADC*

## True Frequency

To work out the frequency of an analogue (sine/cosine) waveform, the time period must be calculated. The time period of a waveform is the time it takes for the wave to complete one full cycle. To determine the exact times when a cycle both starts and finishes, a voltage value should be chosen. Once the waveform crosses from the negative side to the positive side (or vice versa), the cycle can be concluded as having completed.

With non-shifted waveforms, it would make sense to use 0V as the defined crossing voltage, however, since the wave has to be shifted due to the unipolar nature of the ADC a different midpoint value must be chosen. The midpoint of 2.5V is a good starting point, however, what happens if the midpoint of the waveform shifts to 2.6V? Later in the project it will be proven that the zero crossing voltage value does not necessarily have to be the midpoint of the wave, it could technically be anywhere between Vmin and Vmax (e.g. 1.1V and 3.9V) and the frequency values will remain unaffected. Later in the report, there is a section to show additional functionality that was added to calculate a running midpoint (as required for the VRMS).

Therefore, the initial zero crossing voltage value can be set to 2.5V to begin with – as it is a safe value at roughly the expected midpoint of the wave. Figure 27, as shown below, showcases an analogue waveform with annotations pointing to the true zero points that need to be accurately measured.

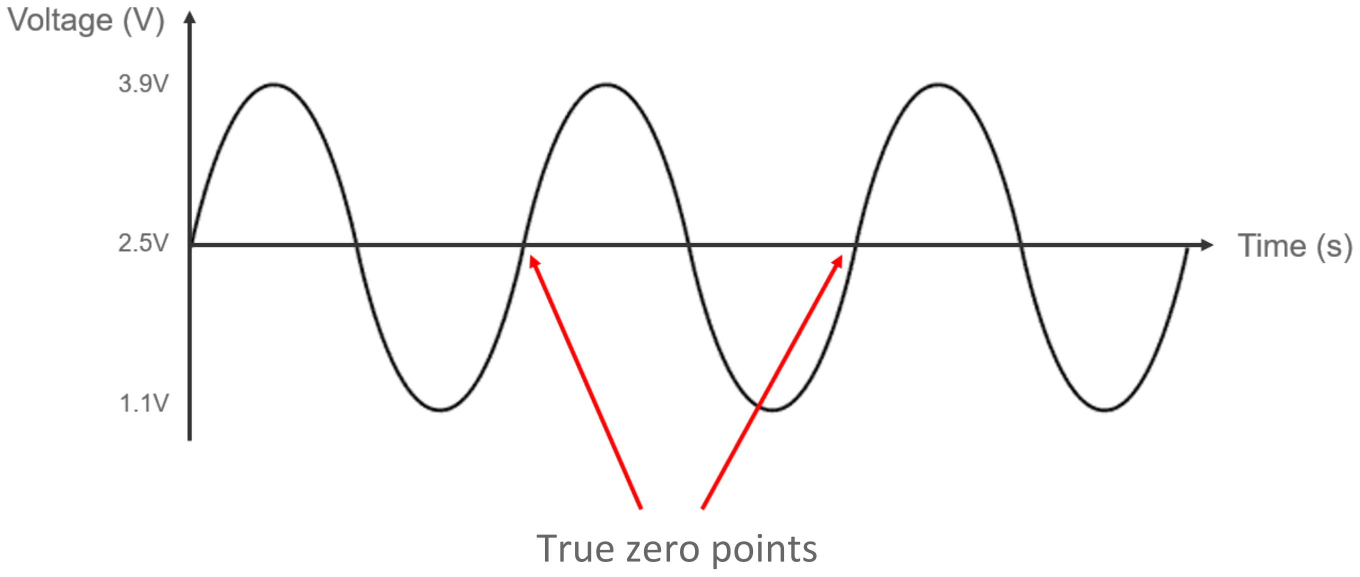


*Figure 27 - Analogue waveform with annotations pointing to the crucial true zero points of a particular cycle in the wave*

The ADC takes samples of the input analogue signal, with almost exactly 24 samples per cycle. This can be visualised in the analogue waveform plot, as shown below, in Figure 28.

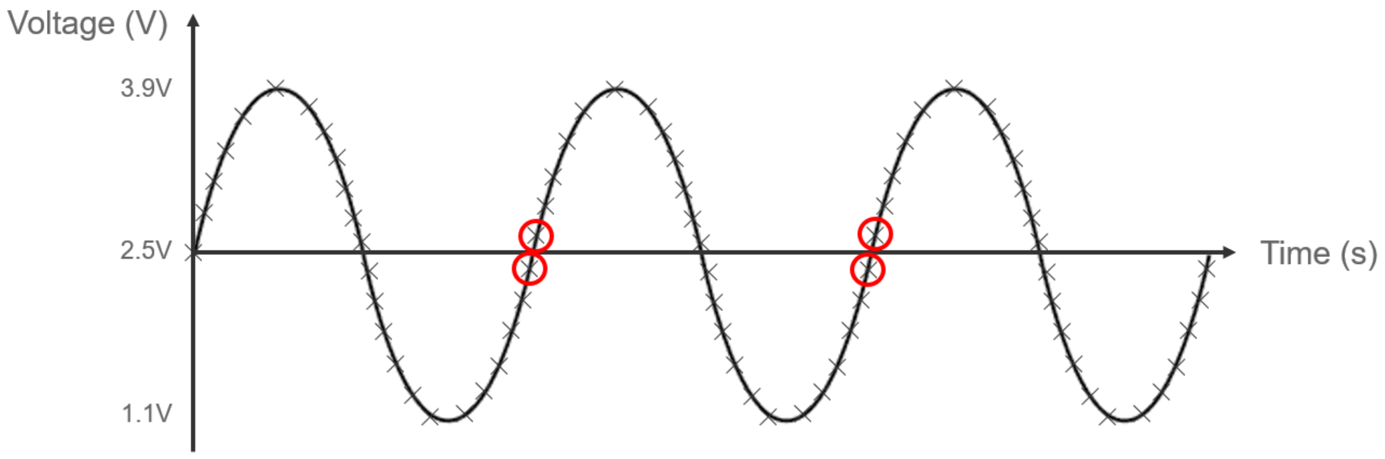


*Figure 28 - Analogue waveform with important true zero crossing sample points highlighted*

To find these true zero points, the frequency algorithm checks for scenarios where any two consecutive samples match the following criteria, as shown in figure 29 below:

$$previous\ sample\ voltage \leq zero\ crossing\ voltage < current\ sample\ voltage$$

*Figure 29 - Sampling pair voltage scenario that the algorithm checks for*

When this scenario occurs, then the algorithm will know that the analogue waveform has just crossed the zero crossing. Knowing this, the algorithm will then multiply these two voltage values then use some basic trigonometry to determine the exact time of the crossing. This can be visualized by extrapolating these two values into their linear triangle formation, as shown below, in Figure 30.

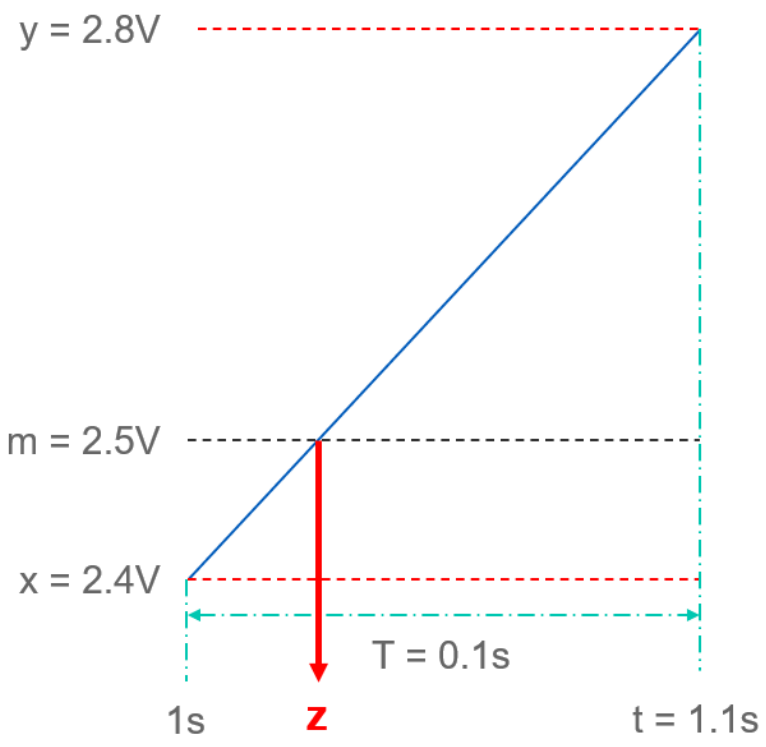


*Figure 30 - Extrapolated triangle diagram of the two zero crossing points*

Figure 30, as shown above, is the extrapolated triangle of the two zero crossing points, as explained previously. The “y” voltage value of 2.8V is the current sample reading and the “x” voltage value of 2.4V is the previous sample reading. The true line between these two sample points will have a very slight bend, however, this bend is so miniscule that a straight line is assumed between the points instead. “m” is the defined zero crossing voltage value of 2.5V.

The key to this frequency algorithm is the fact that it’s trying to figure out the time when the zero crossing value is crossed by the waveform, as defined by the value “z”. Since both the voltage and time values of x and y are stored in variables, a trigonometry equation, as shown below in Figure 31, can be utilised to determine the exact time of the zero crossing – “z”.

$$z = t - \left(T \times \frac{y - m}{y - x}\right)$$

$$z = 1.1 - \left(0.1 \times \frac{2.8 - 2.5}{2.8 - 2.4}\right)$$

$$z = 1.1 - \left(0.1 \times \frac{0.3}{0.4}\right)$$

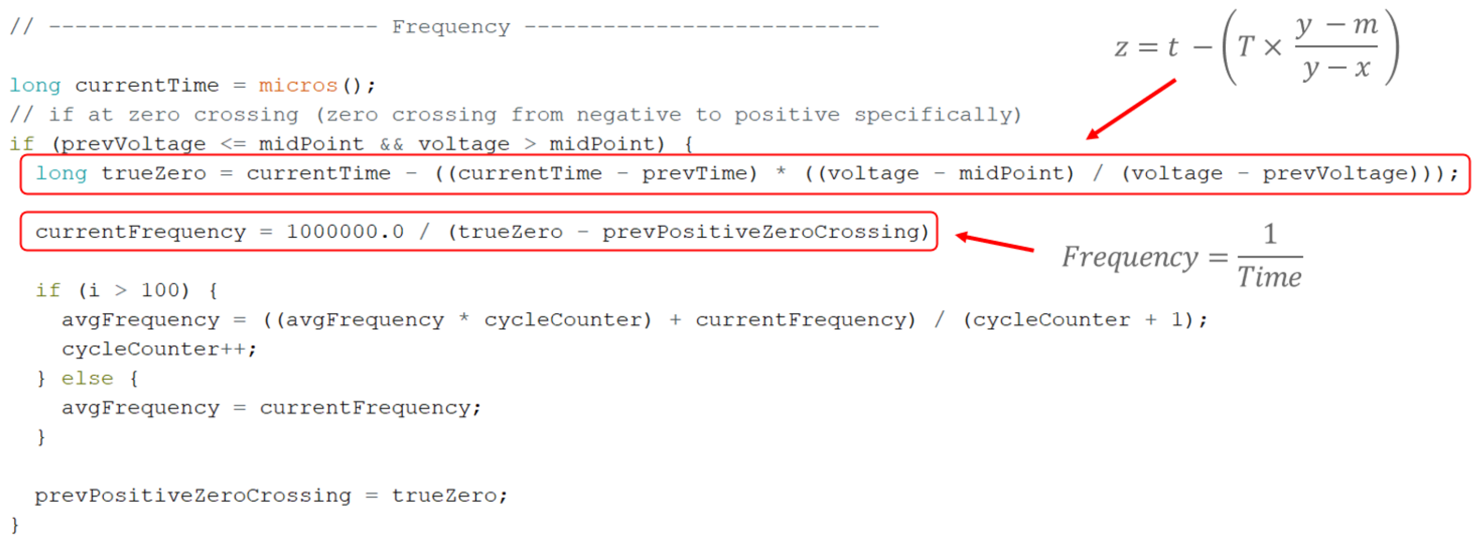
$$z = 1.1 - 0.075$$

$$z = 1.025\ seconds$$

*Figure 31 - Trigonometry equation used to determine the exact time that the zero crossing occurs at*

As shown in the equation in Figure 31 above, the zero crossing time was calculated to be 1.025 seconds, which is a quarter of the time between the example samples – “x” and “y”. This is validated by the fact that the zero crossing point, “m”, is a quarter of the way between the previous and current values – “x” and “y” respectively. This has proven that the zero crossing time and the voltages are directly proportional to each other in nature, hence validating that the triangle trigonometry can be utilised for the calculations.

Figure 32, as shown below, showcases the true frequency algorithm C++ ESP32 code. The first highlighted line is equivalent to the equation shown in Figure 31 and the second highlighted line is equivalent to the frequency time equation. The lower half of Figure 32 contains code for the running average frequency. The “i > 100” if statement is used as a precautionary warmup for the ADC breakout to allow time for any unexpected values to be cycled through – as this is important for the running averages. The ESP32 “micros()” function is used to obtain the current time in microseconds. The timings must be very precise, therefore the micros() function was used over the millis() function. The full code can be seen in Appendix 9.



```
// ------------------------- Frequency ---------------------------

long currentTime = micros();
// if at zero crossing (zero crossing from negative to positive specifically)
if (prevVoltage <= midPoint && voltage > midPoint) {
  long trueZero = currentTime - ((currentTime - prevTime) * ((voltage - midPoint) / (voltage - prevVoltage)));

  currentFrequency = 1000000.0 / (trueZero - prevPositiveZeroCrossing)

  if (i > 100) {
    avgFrequency = ((avgFrequency * cycleCounter) + currentFrequency) / (cycleCounter + 1);
    cycleCounter++;
  } else {
    avgFrequency = currentFrequency;
  }

  prevPositiveZeroCrossing = trueZero;
}
```

$$z = t - \left(T \times \frac{y - m}{y - x}\right)$$

$$Frequency = \frac{1}{Time}$$

Figure 32 - ESP32 C++ code for the true frequency algorithm

## VRMS Algorithm

When using a voltage/time plot to work out the voltage of a sinusoidal waveform, this can easily be done by working out the value of Vpeak (Vp) using one of the 3 equations shown in Figure 33, below.

$$\text{Vpeak} = \text{Vmax} - \text{midpoint}$$

$$\text{Vpeak} = -(\text{midpoint} - \text{Vmin})$$

$$\text{Vpeak} = (\text{Vmax} - \text{Vmin}) / 2$$

*Figure 33 - Three possible equations for calculated Vpeak*

To use any of these methodologies for calculating the voltage of the input analogue signal, the Vmax and/or Vmin values of each wave cycle have to be determined. (The midpoint of the input analogue waveform could change but this is solved by using the running midpoint value, which is explained later in the report). The problem lies with obtaining the Vmax or Vmin value – the analogue waveform is sampled with around 24 samples per cycle so it's unlikely that an individual sample will be taken on either/both of the very peak and trough points of the analogue waveform. This presents an issue where the calculated voltages using these values will likely be slightly less than the true voltage due to the nature of the timings of the samples.

The first proposed solution to this described issue was to use the Vmin and Vmax values of the previous "n" cycles (n=5 for example) and then find the voltage based on the highest Vmax value and the lowest Vmin value of those previous "n" number of samples. This solution seemed to work to a passable standard, however, the voltage values were not very accurate/precise (as they depended on the exact timings of the max and min samples) and hence, the algorithm failed as a whole.

The final solution found was to first work out the voltage root-mean-square value (VRMS) then calculate the voltage from this VRMS value.

$$x_{rms} = \sqrt{\frac{{x_1}^2 + {x_2}^2 + {x_3}^2 + \ldots + {x_{24}}^2}{24}}$$

*Figure 34 - VRMS equation*

The VRMS equation is shown above, in Figure 34. This equation can be utilised to work out the VRMS of the current cycle at each new sample point. At each new sample point, the current sample point and the 23 previous sample points should each be squared, then summed, then divided by 24, then square rooted. To store the 23 most recent sample readings into the ESP32 memory, a circular buffer should be utilised. This can be performed by using an array of size 23, along with a simple integer variable to hold the current position. On each new sample reading, the most recent reading will replace the oldest reading.

On each new sample reading, the full circular buffer can be iterated over then processed using the equation shown in Figure 34. Iterating over the circular buffer is fine in theory, however, in practice it would be quite a computationally expensive process as all 24 values need to be iterated over on each new sample reading (23 in circular buffer + 1 most recent sample reading). This comes out at an algorithm time complexity of 24O(N).

What if there was an algorithm that could be used instead of the circular buffer algorithm? An algorithm that is more computationally efficient, more memory efficient, more readable and has shorter overall code than the previously described circular buffer algorithm?

This wonder-algorithm is known as the sliding window algorithm. The sliding window algorithm simply utilises a single variable for recalculating the VRMS, without having to iterate over the whole circular buffer again each time. The premise of the sliding window algorithm is that on each new sample value, the previous VRMS value is extrapolated out back to the sum square values then the oldest squared sample voltage is subtracted from the sum square voltages and the newest sample voltage is added to the sum square voltages. These sum square voltages are then divided by 24 then square rooted again to obtain the new VRMS. This can be visualised using Figure 35, as shown below.

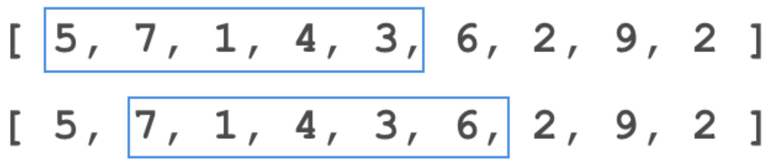


*Figure 35 - Sliding window visualisation*

The sliding window algorithm has an identical output to the circular buffer algorithm, however, it is much more efficient – with an algorithm time complexity of O(N) rather than 24O(N). This substantial increase in efficiency was a critical turning point for the project as the ESP32 taking the 1200sps ADC readings doesn’t have that much available time and computing power to spare between taking each of the ADC sample readings (especially for 2 ADCs, as shown later).

```
int pos = i % bufferSize;
float oldM = bufferM[pos];
float oldV = bufferV[pos];
float oldC = bufferC[pos];

// midPoint
midPointAvg = ((midPointAvg * bufferSize) + voltage - (oldV + oldM)) / bufferSize;
midPoint = midPointAvg;
bufferM[pos] = midPoint;

// vrms
sumSquareVoltages = (pow(vrms, 2) * bufferSize) - pow(oldV, 2) + pow(voltage - midPoint, 2);
vrms = sqrt(sumSquareVoltages / bufferSize);
bufferV[pos] = voltage - midPoint;
```

*Figure 36 - VRMS sliding window algorithm ESP32 C++ code*

Figure 36, as shown above, showcases the ESP32 C++ code for the sliding window VRMS algorithm. “M” represents midpoint, “V” represents voltage and “C represents current. The midpoint and current functionality will be explained in later sections. Initially, the old buffer positions are found using a modulus calculation then the old midpoint, old voltage (and old current values) are obtained from the three buffers. The midpoint has to be recalculated before obtaining the voltage, as will also be explained in a later section. Once the midpoint has been calculated, it can then be utilised for the VRMS code.

Firstly, the VRMS value is squared then multiplied by the bufferSize to extract out the sumSquareVoltages value. Once the sumSquareVoltages value has been extrapolated out, then the old value can be subtracted and the new value can be added to the sumSquareVoltages. The oldV variable holds the oldest voltage value, which was just obtained from the buffer. The red line highlights the oldest value in the sliding window that is being subtracted from the sumSquareVoltages and the green line highlights the current value that is being added to the sumSquareVoltages. The VRMS is then recalculated and the new value is stored in the first point in the voltage buffer. The full code can be found in Appendix 9.

$$V_p = \sqrt{2} \times V_{rms}$$

*Figure 37 - VRMS to Vpeak equation*

Once the VRMS has been recalculated for the new sample, the V-peak voltage value can then be worked out by using the VRMS-to-voltage conversion equation, as shown in the equation in Figure 37, above.

## Circular Buffer Size

The bufferSize variable is set earlier in the program. Altering the bufferSize value alters how the circular buffer functionality works. The minimum bufferSize for a 50Hz wave is 24 (according to Nyquist's Theorem [16]), to allow for a full cycle to be used in the calculations. In practice, this VRMS value will fluctuate slightly due to the analogue signal itself not being exactly 50Hz. These slightly fluctuations can be smoothed out by increasing the bufferSize. Increasing the bufferSize to 1200, for example, means that the VRMS of the new sample is calculated based upon the values taken in the whole last second. This will produce a very accurate and smooth VRMS, however, it will also mean that the VRMS has a slight delay when tracking the changing VRMS – as it is the antagonist to its smoothness. The perfect bufferSize must be set depending on the use case of the smart wireless sensor. If a 1Hz signal has to be measured, then a bufferSize of 1200 is the absolute minimum that can be used at this 1200sps sampling rate – to ensure that at least one full wave is sampled. In the UK mains scenario with a signal of frequency 50Hz, vigorous testing and analysis proved a buffer size of anywhere between 24 and 1200 to be satisfactory – and should be tailored to the end user need, whether they need a snappier or smoother VRMS value.

The ESP32 does have limited on-board memory and computational power so this must factored in when determining the buffer size. Through testing and analysis, a total bufferSize of 1200 was found to be computationally feasible and hence, when using three buffers (for midpoint, voltage and current) a bufferSize of 400 each buffer was found to be optimal for this project in terms of performance, smoothness and delay.

## Frequency and VRMS in action

At this stage in the project, the midpoint, frequency, and VRMS were all being successfully measured and calculated.

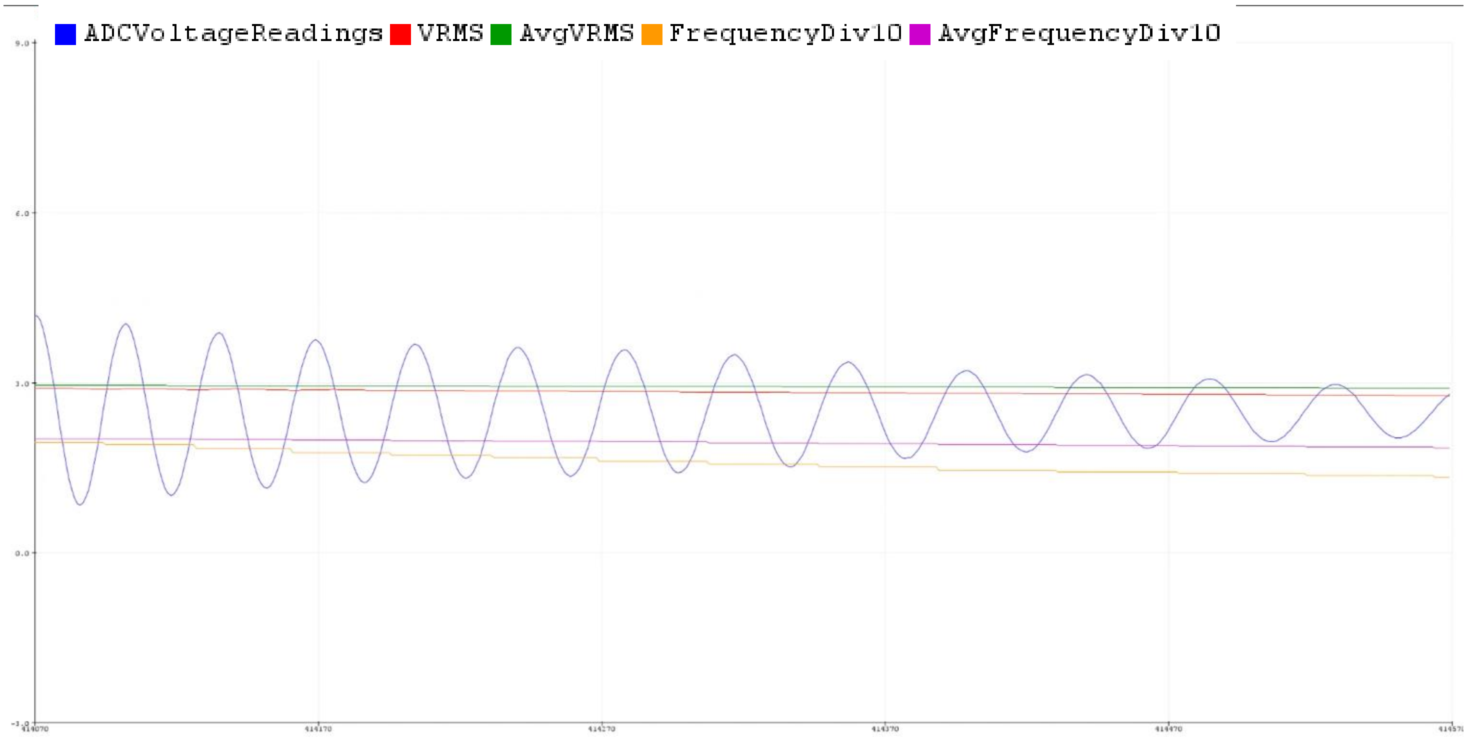


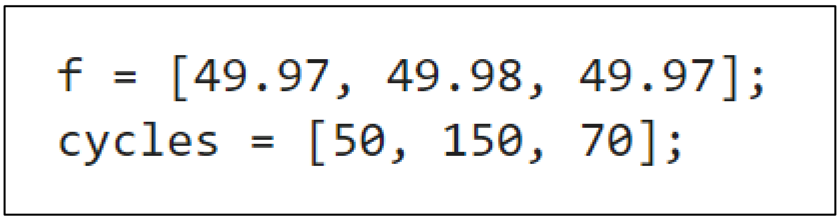
*Figure 38 - Serial plotter waveform of the ADC readings, VRMS and frequency while the amplitude and frequency are being reduced*

Figure 38, as shown above, showcases a screenshot of the Arduino IDE Serial Plotter for a function generator analogue sinusoidal signal. It’s very clear from the analogue input sine wave in the figure that both the frequency and voltage of the sine wave are both decreasing rapidly. Hence, the measured frequency and VRMS lines of the plot decrease accordingly, as shown by the yellow and red line, respectively. A running frequency and VRMS average were also being calculated and plotted, as can be seen by the pink and green lines, respectively. This wave had been running for approximately five seconds prior to this screenshot so the running average lines can be seen to decrease at a more gradual rate than the rate of the frequency and VRMS lines.

## Frequency and VRMS Algorithms Validation in MATLAB

Up to this point in the project, all of the project code had been written for the ESP32 in C++ in the Arduino IDE. The algorithms that were engineered seemed to work perfectly, however, there was no way of validate the algorithms using the real analogue signals due to slight inconsistencies in both values and timings, etc. The only way that the algorithms could truly be validated is through a simulated scenario, where the inputs, timings, readings, etc. could be controlled down to pinpoint accuracy and consistency.

MATLAB was chosen as the simulation program for the algorithm validation task as it was known to be precise and consistent enough for the task. To actually validate the algorithms, the input analogue sine wave had to be constructed then set at a certain frequency for a specified number of cycles then the frequency would be altered slightly and the sine wave would run for another number of cycles, and so on.



```
f = [49.97, 49.98, 49.97];
cycles = [50, 150, 70];
```

*Figure 39 - MATLAB sine wave frequency simulation values*

Figure 39, as shown above, showcases the finalised desired sine wave frequency values for the validation program. The sine wave would run at 49.97Hz for 50 cycles, then its frequency would be increased to 49.98Hz for 150 cycles then its frequency would be decreased back down to 49.97Hz for the final 70 cycles.

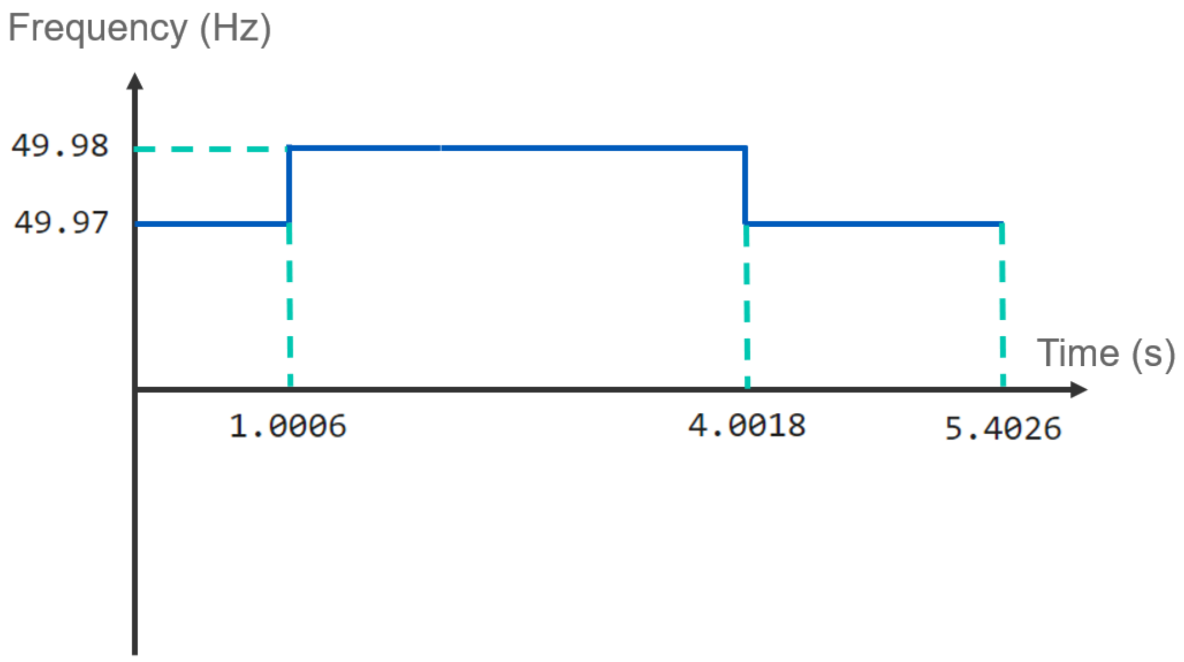


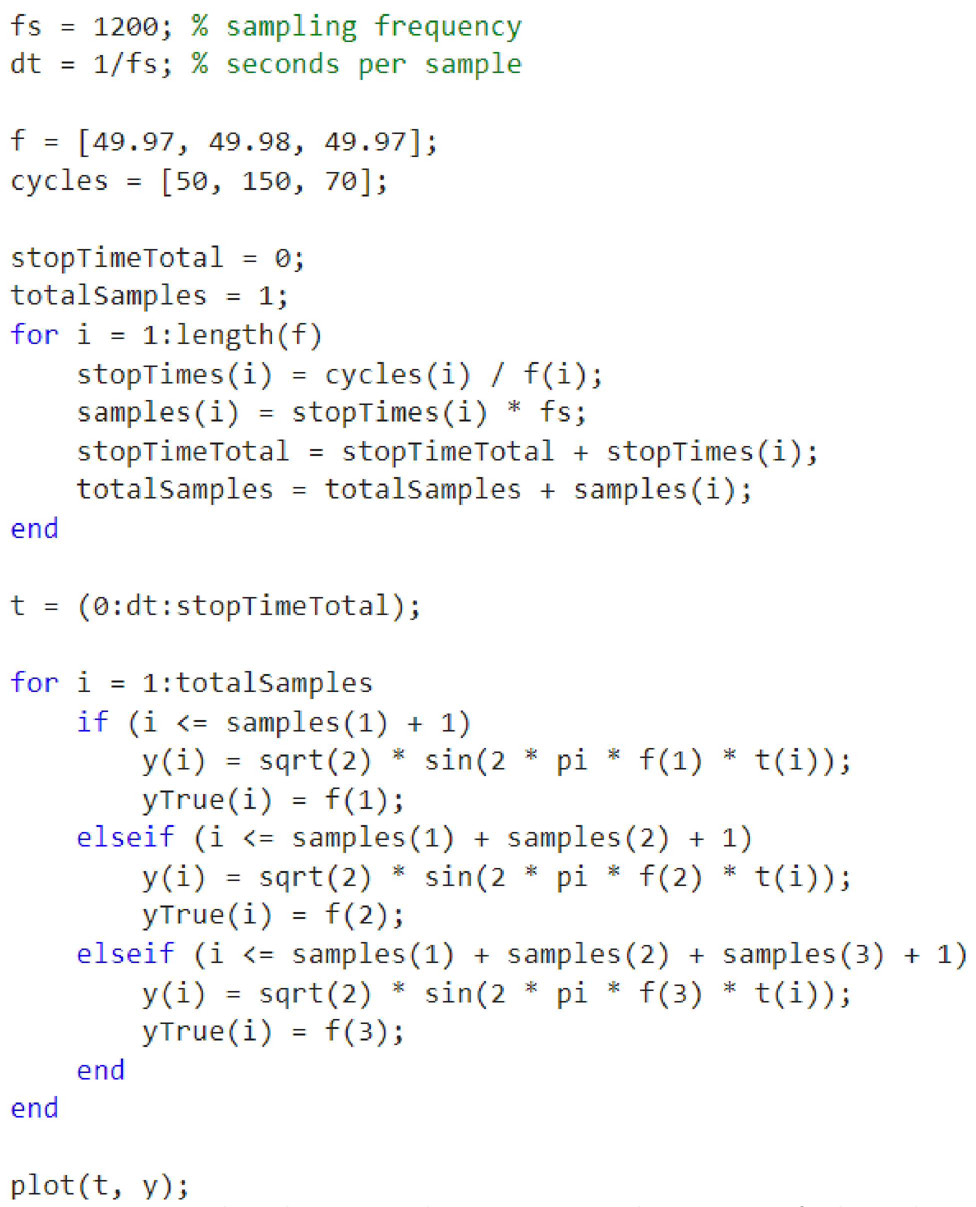
*Figure 40 - Rough frequency time plot of the expected varying frequency input analogue sine wave*

Figure 40, as shown above, showcases a rough frequency time plot used to represent the expected varying frequency of the simulated input analogue sine wave. To validate the frequency algorithm, both the input and output sine wave frequency waves should be plotted and compared. They should be compared in shape and values to determine if the measured frequencies have met the required frequency accuracy and precision. The desired frequency accuracy of this project was set out at +-10mHz and the desired VRMS accuracy of this project was set out at +-10mV accuracy.

The expected VRMS and measured VRMS should be plotted against time and compared to validate the VRMS algorithm, similar to validating the frequency algorithm.

## MATLAB Sine Wave Population

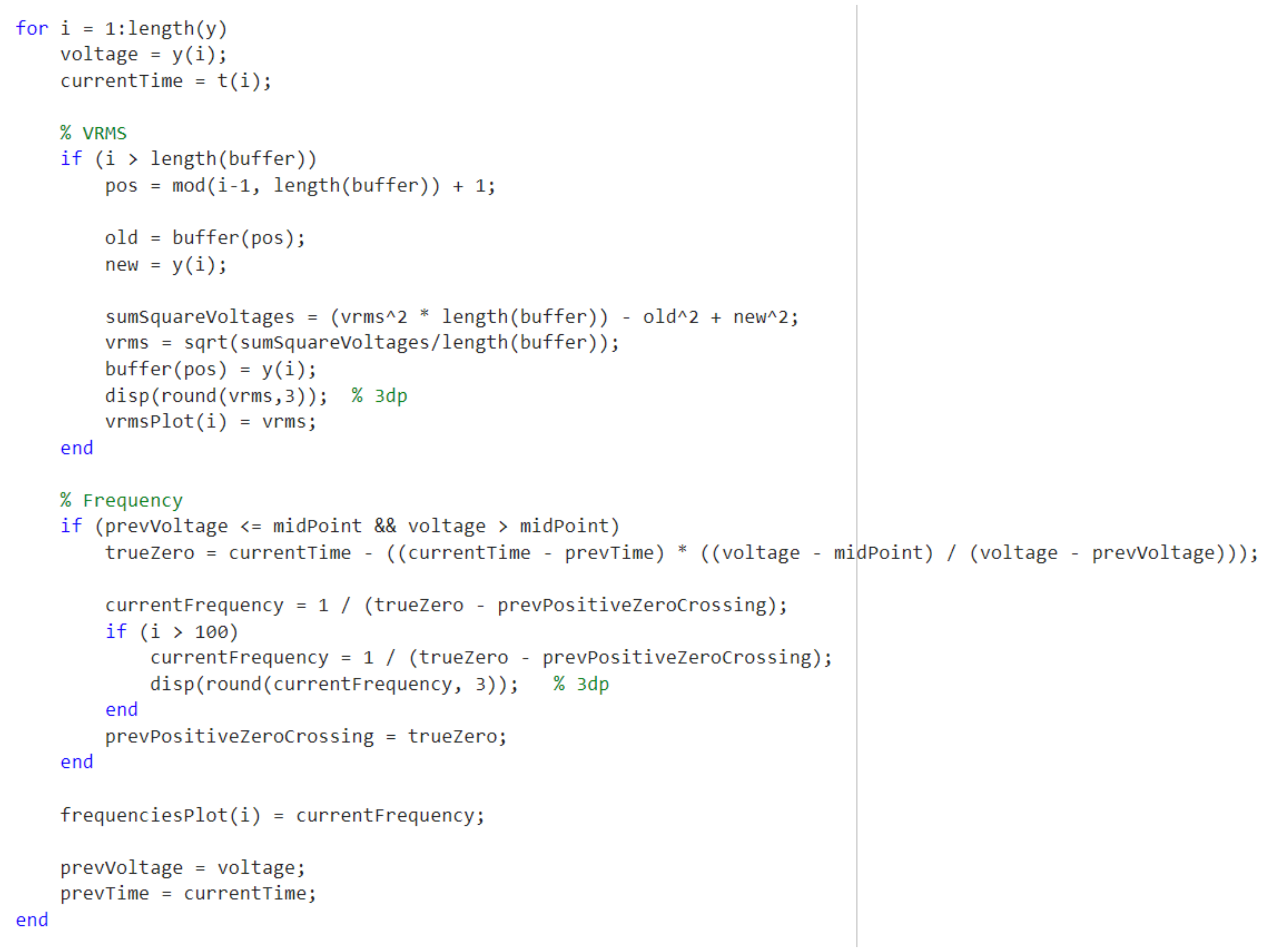
Before being able to validate the frequency and VRMS algorithms, the sine wave specified in Figure 39 had to be constructed.



```
fs = 1200; % sampling frequency
dt = 1/fs; % seconds per sample

f = [49.97, 49.98, 49.97];
cycles = [50, 150, 70];

stopTimeTotal = 0;
totalSamples = 1;
for i = 1:length(f)
    stopTimes(i) = cycles(i) / f(i);
    samples(i) = stopTimes(i) * fs;
    stopTimeTotal = stopTimeTotal + stopTimes(i);
    totalSamples = totalSamples + samples(i);
end

t = (0:dt:stopTimeTotal);

for i = 1:totalSamples
    if (i <= samples(1) + 1)
        y(i) = sqrt(2) * sin(2 * pi * f(1) * t(i));
        yTrue(i) = f(1);
    elseif (i <= samples(1) + samples(2) + 1)
        y(i) = sqrt(2) * sin(2 * pi * f(2) * t(i));
        yTrue(i) = f(2);
    elseif (i <= samples(1) + samples(2) + samples(3) + 1)
        y(i) = sqrt(2) * sin(2 * pi * f(3) * t(i));
        yTrue(i) = f(3);
    end
end

plot(t, y);
```

*Figure 41 - MATLAB code used to construct the input MATLAB analogue sine wave for the simulation*

Figure 41,  as shown above, showcases the MATLAB code used to construct the input MATLAB analogue sine wave. The sampling frequency, the specified frequencies and the cycles were set up in a parameterised fashion to ensure that these values could easily be altered. The sampling rate was set at 1200sps, as required for a 50Hz sine wave with 24 samples being read per cycle. The wave VRMS was set at exactly 1V. The stopTimes and samples vectors were populated then they were used to create the final sine wave vector, as specified by identifier "y". This final sine wave was plotted on the frequency time graph, in preparation for a later comparison. The full code for the MATLAB algorithm validation can be found in Appendix 11.



```
for i = 1:length(y)
    voltage = y(i);
    currentTime = t(i);

    % VRMS
    if (i > length(buffer))
        pos = mod(i-1, length(buffer)) + 1;

        old = buffer(pos);
        new = y(i);

        sumSquareVoltages = (vrms^2 * length(buffer)) - old^2 + new^2;
        vrms = sqrt(sumSquareVoltages/length(buffer));
        buffer(pos) = y(i);
        disp(round(vrms,3));  % 3dp
        vrmsPlot(i) = vrms;
    end

    % Frequency
    if (prevVoltage <= midPoint && voltage > midPoint)
        trueZero = currentTime - ((currentTime - prevTime) * ((voltage - midPoint) / (voltage - prevVoltage)));

        currentFrequency = 1 / (trueZero - prevPositiveZeroCrossing);
        if (i > 100)
            currentFrequency = 1 / (trueZero - prevPositiveZeroCrossing);
            disp(round(currentFrequency, 3));   % 3dp
        end
        prevPositiveZeroCrossing = trueZero;
    end

    frequenciesPlot(i) = currentFrequency;

    prevVoltage = voltage;
    prevTime = currentTime;
end
```

*Figure 42 - MATLAB code for the VRMS and frequency algorithms*

Figure 42, as shown above, showcases the MATLAB code for the VRMS and frequency algorithms. This code works by iterating over each voltage value in the sine wave vector "y", as if the value was being passed from the ADC to the ESP32. Since this signal is simulated, the midpoint of the signal is set at 0, rather than 2.5V for the ADC breakout – due to the signal having to be shifted. The syntax of the algorithm code in MATLAB is obviously slightly different to the ESP32 C++ code, however, the overall functionality is kept to be exactly the same in both programs.

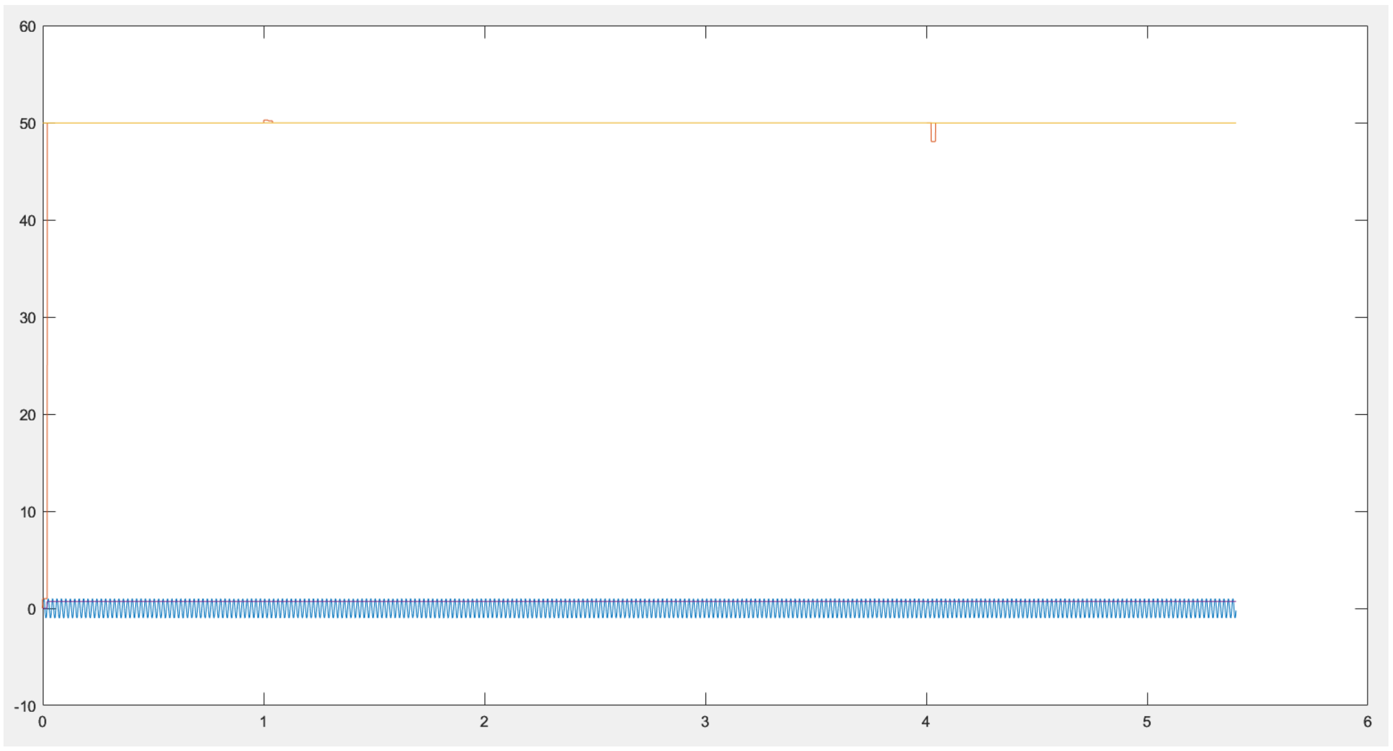
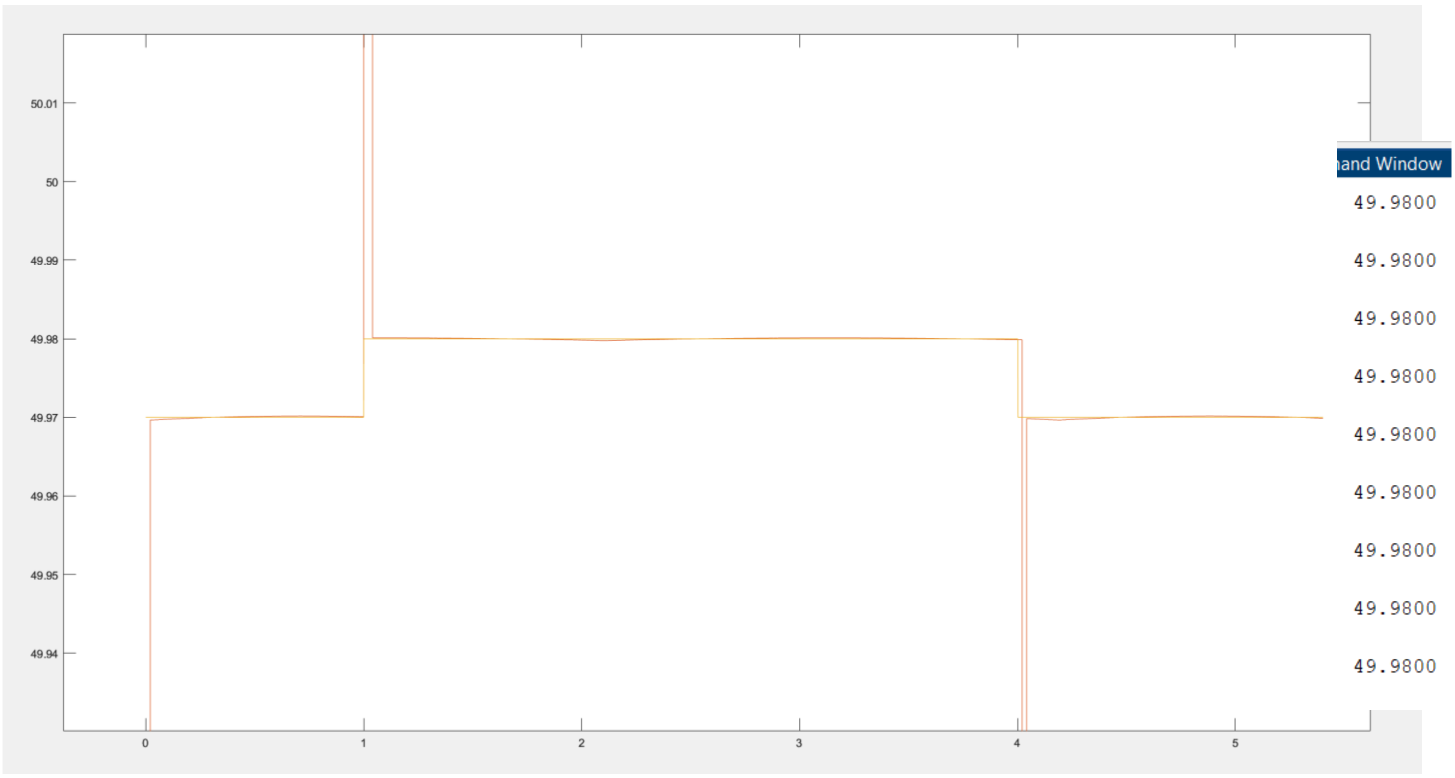


*Figure 44 - MATLAB output waveform of the algorithm validation simulation program*

Figure 43, as shown above, showcases the MATLAB signal wave output of the algorithm validation program. This image is very zoomed out to give a basic overview of the signal outputs. The blue input sine wave can be seen at the bottom of the graph and the expected and actual frequency can be seen plotted in yellow and orange, respectively.



```
49.9800
49.9800
49.9800
49.9800
49.9800
49.9800
49.9800
49.9800
49.9800
```

*Figure 43 - MATLAB output waveform zoomed (on y-axis) of the frequency section*

Figure 44, as shown above, showcases a zoomed in (with respect to the y axis) screenshot of expected and actual frequency lines. The actual frequency output (orange line) clearly shows that the measured frequency follows the expected frequency line extremely precisely. At the frequency changes, the analogue sine wave is cut mid-way through the cycles, and hence causes the slight blip in frequency at those frequency change points. These blip lines may look to be fairly significant, however, this is due to this graph being zoomed in to such a high y-axis magnification. A goal of the this project was to achieve a frequency measurement accuracy of +- 10mHz, however, after testing and analysis using these MATLAB simulations, the frequency algorithm was found to have achieved an even-better accuracy of **+- 0.3mHz – which is over 33 times more accurate than originally required!** This accuracy can be seen in the command window frequency values to the right hand side of figure 44.

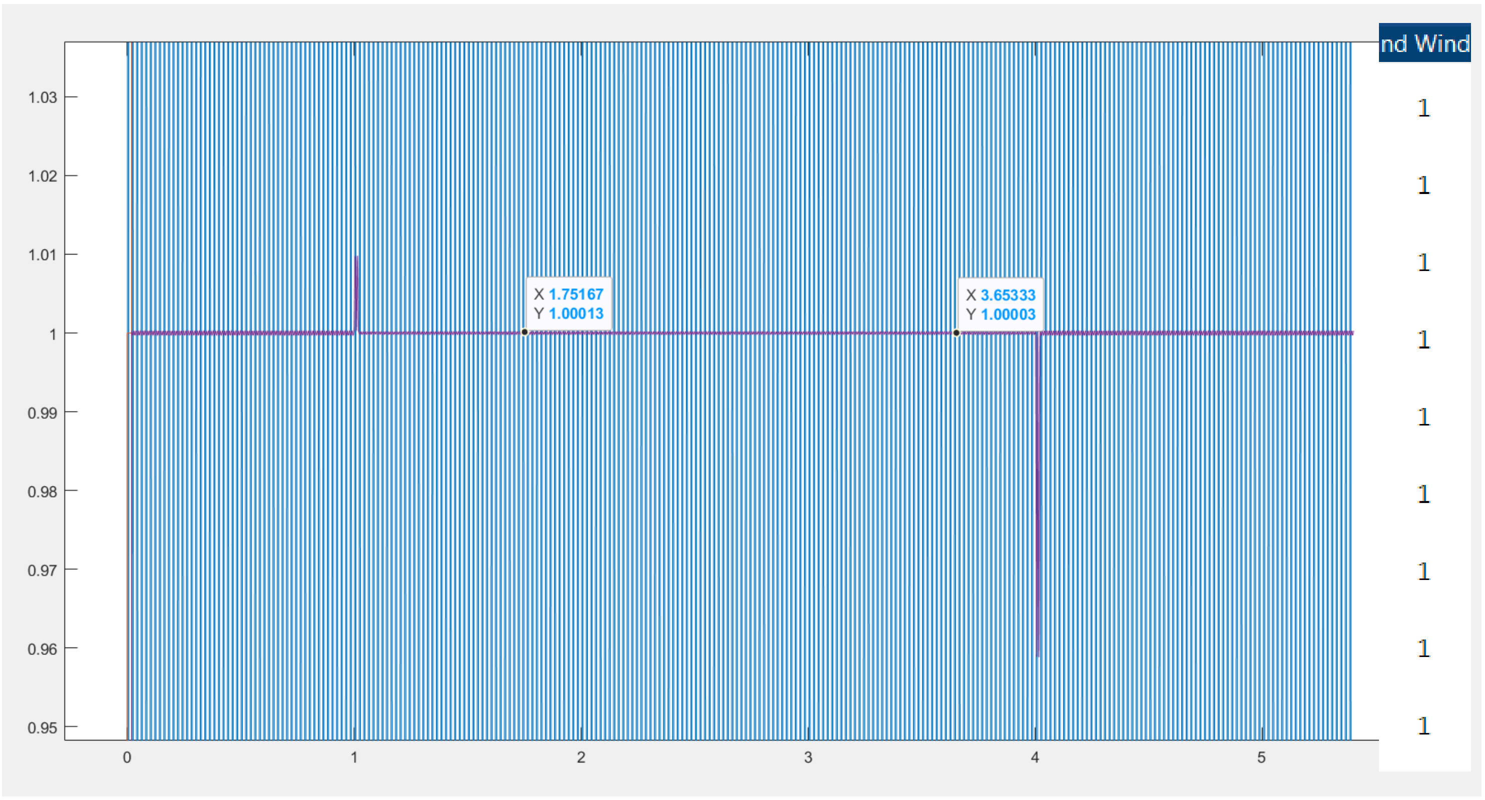


*Figure 45 - MATLAB output waveform zoomed (on y-axis) of the VRMS section*

Figure 45, as shown above, showcases a very magnified (on the y-axis) screenshot of the sine wave/VRMS section of the MATLAB algorithm validation output. The sine wave can be seen in blue and the measured VRMS is plotted as the horizontal purple line. The VRMS line clearly shows that the measured VRMS follows the expected VRMS of 1V almost perfectly. There are slight blips at the sine wave frequency change points due to the sine wave being cut mid-way through the cycles, hence causing the VRMS for that sample fluctuate very slightly. This blip fluctuation does not exceed 40mV, as shown in the plot. A goal of the this project was to achieve a VRMS measurement accuracy of +- 10mHz, however, after testing and analysis using these MATLAB simulations, the VRMS algorithm was found to have achieved an even-better accuracy **of +- 0.3mV – which is over 33 times more accurate than originally required!** This accuracy can be seen in the command window voltage values to the right hand side of Figure 45. These values have been rounded to 4 decimal places.

Overall, the MATLAB algorithm validation program has successfully proven that both the frequency and VRMS algorithms are 100% valid. It has proven that the algorithms have achieved an accuracy that is has greatly exceeded expectations – at 33 times more accurate than originally required! Knowing that the frequency and VRMS algorithms were completely valid and extremely accurate, the MATLAB simulation route was complete and the project moved back to focusing on the ESP32 program and functionality.

## Midpoint

In an ideal scenario, the ADC would be bipolar and hence, would be able to measure both the positive and negative voltage. However, the ADS1015 is unipolar so the analogue signal has to be shifted by 2.5V to ensure that the signal lies between the ADC input range of 0-5.3V.

For a non-shifted signal, the midpoint would simply be at 0V, however, the midpoint of the shifted signal will lie approximately at 2.5V. Due to the nature of the voltage attenuation and shifting op-amp circuit, the midpoint is very unlikely to lie at exactly 2.5V. This fact is crucial, as the midpoint being slightly off of the expected 2.5V could throw off the VRMS very slightly. Therefore it is quintessential to this project to accurately track the midpoint. The final solution formulated for doing so was to use a sliding window circular buffer, similar to the implementation for the VRMS. This can be clearly seen in the VRMS ESP32 C++ code shown earlier, in Figure 36. As shown in Figure 36, the midpoint buffer (bufferM) is set to the same size as the VRMS buffer to ensure that the midpoint value is updated via the sliding window algorithm in exact sync with the VRMS buffer (bufferV). This ensures that the midpoint and VRMS maintain their accuracy and precision down to a perfect degree.

Figure 46, as shown below, showcases the live midpoint tracking, in action, for a waveform of varying DC offset. It's clear to see from the screenshot that the DC offset is being increased dramatically by adjusting the DC offset knob of the function generator. As a consequence, the ADC1Voltage readings shift vertically upwards in response. The yellow line, representing the tracking midpoint, can be seen very precisely tracking the midpoint of the waveform, with a very slight delay due to having a buffersize of 400 (hence 0.333s delay).

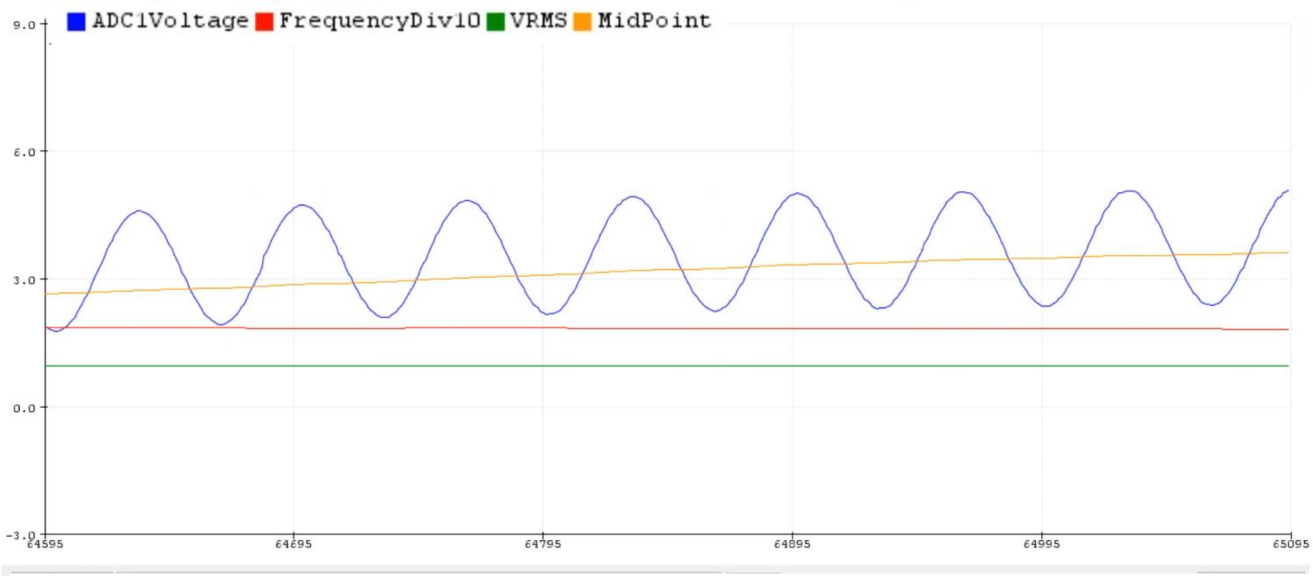
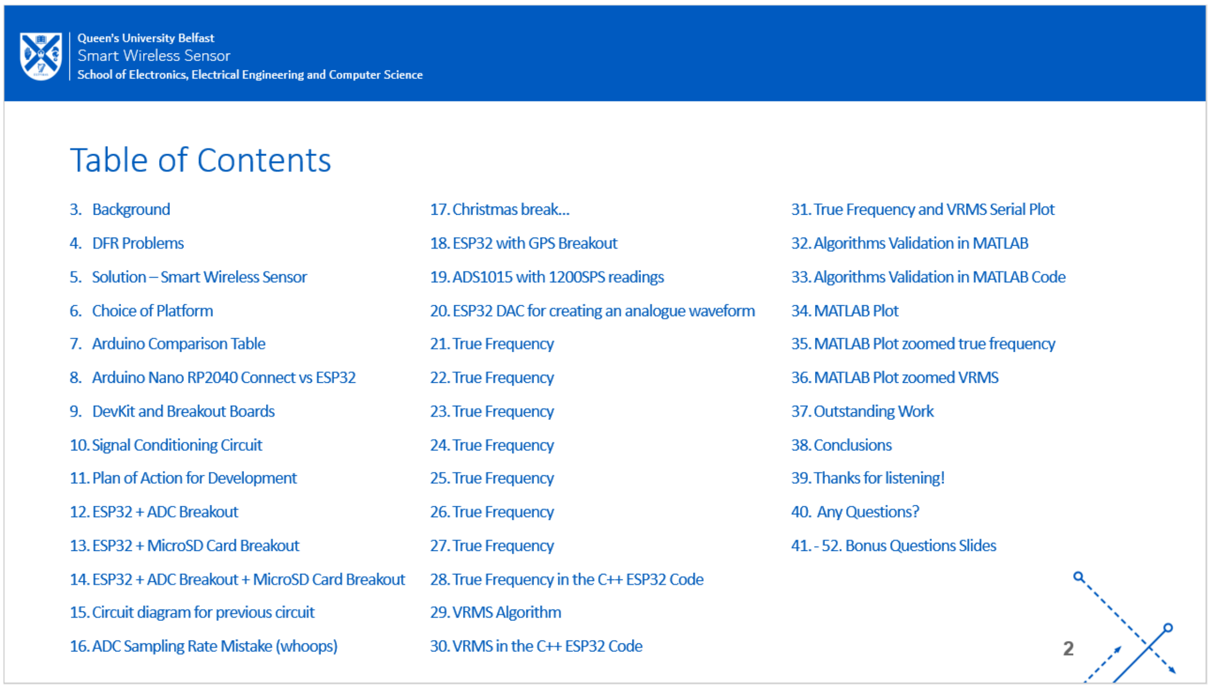


*Figure 46 - Serial plotter showcasing the live midpoint tracking*

Figure 47 most importantly shows that both the frequency and the VRMS are completely unaffected by the varying midpoint. This is a crucial point to having the tracking midpoint algorithm and hence, has been proven to work successfully.

The midpoint of the analogue input wave should not vary much at all, however, the midpoint algorithm successfully tracks the midpoint in a very similar way to the VRMS being tracked, thanks to the same buffer size. The smoothness and delay can be fine-tuned by respectively increasing or decreasing the shared buffer size.

## Oral Presentation and Demo



Queen's University Belfast
Smart Wireless Sensor
School of Electronics, Electrical Engineering and Computer Science

### Table of Contents

2

*Figure 47 - Presentation table of contents slide*

On Friday 24th March, a presentation was delivered on the Smart Wireless Sensor project to two senior lecturers in the EEECS faculty. The presentation covered the following topics, as shown above in Figure 47. A successful demo was given at the end of the presentation.

The demo set-up included a function generator, the ESP32 and the microSD card breakout. The function generator analogue signal was fed into one of the ADC analogue inputs and readings were taken at 1200sps. At this stage in the project, the midpoint, VRMS, average VRMS, voltage, frequency, average frequency and a few other measurements were shown with live-measurements from the function generator analogue input signal. The function generator analogue signal amplitude, frequency and DC offset were varied as part of the demonstration – to showcase the minimal effects that exaggerated fluctuations would have on the measurements.

A photo of the circuit and function generator used in the demo can be seen below, in Figure 48. The function generator can be seen with a frequency set to around 49.6Hz. This was hooked up to the breadboard via 2 lead cables, which were then fed into the red and black leads ports of the breadboard then into the ADC and ESP32 ground, respectively. The demo included a brief showcase of how varying the signal frequency, amplitude and DC offset affected the waveform. The ADC voltage readings, as well as the useful live measurements, were plotted on the Serial Plotter and then printed in the Serial Monitor, for demonstration purposes. The demo also showcased the smart wireless sensor taking readings from the function generator signal outputted as a square wave and a triangular wave. The sensor was able to take these readings with ease and was unaffected, no matter the type of function wave being read.

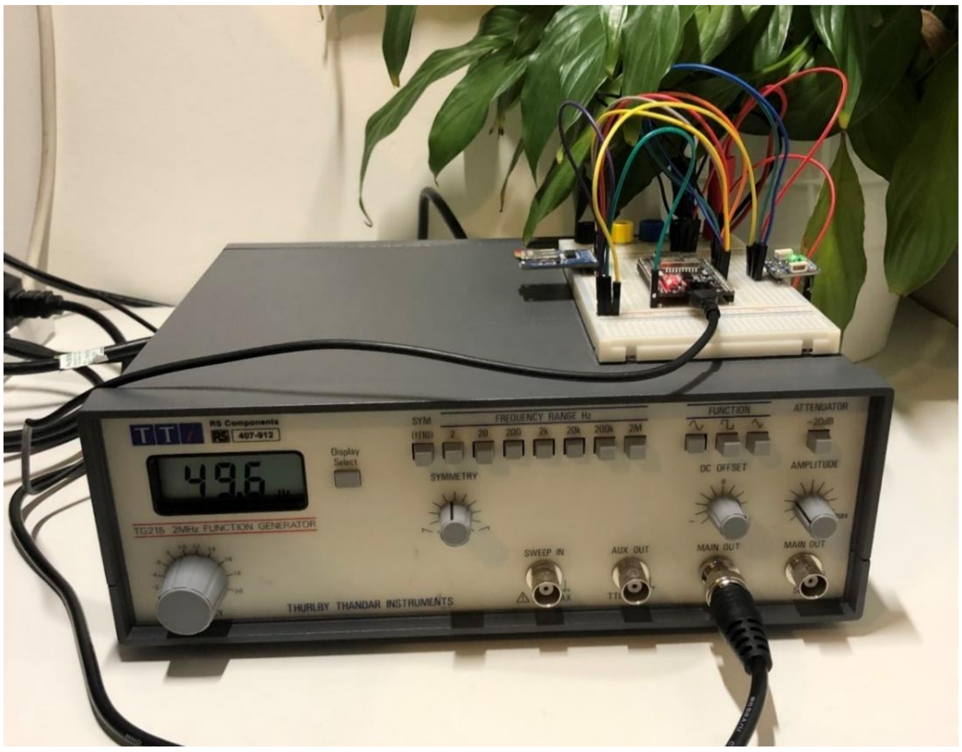


*Figure 48 - Function generator and circuit used in the oral presentation demo*

The Fritzing circuit diagram for the demo circuit can be seen below, in Figure 49.

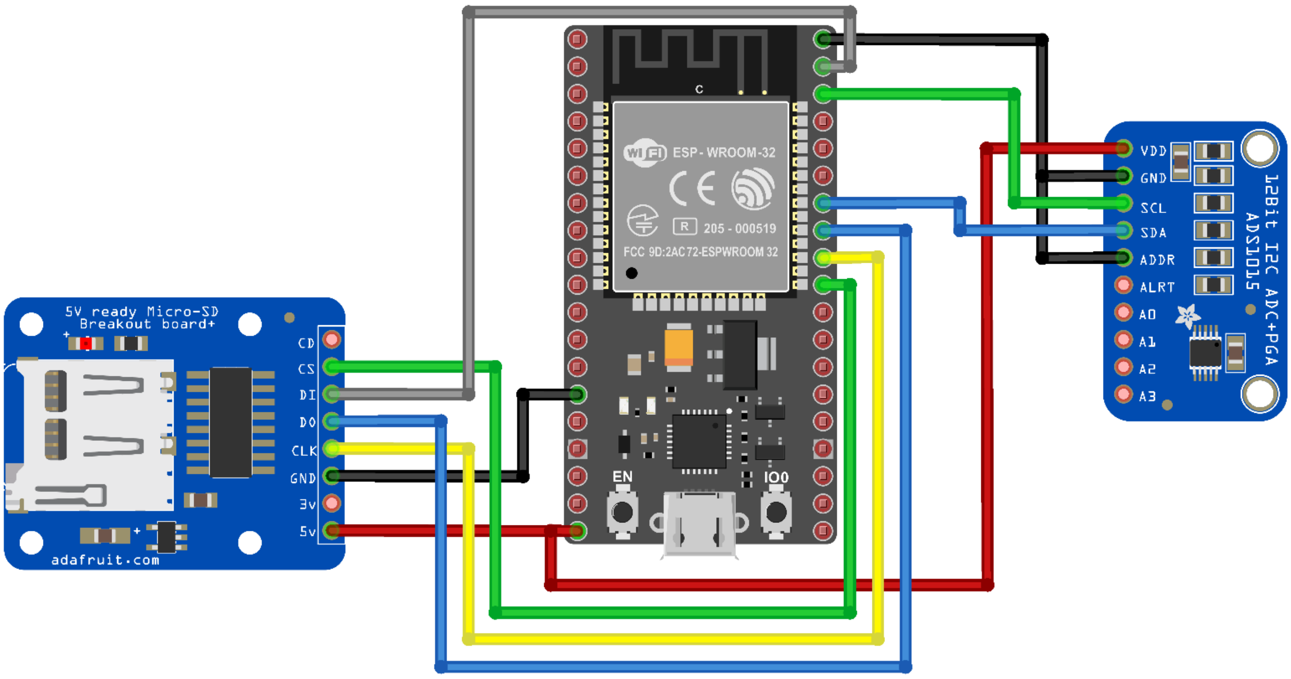


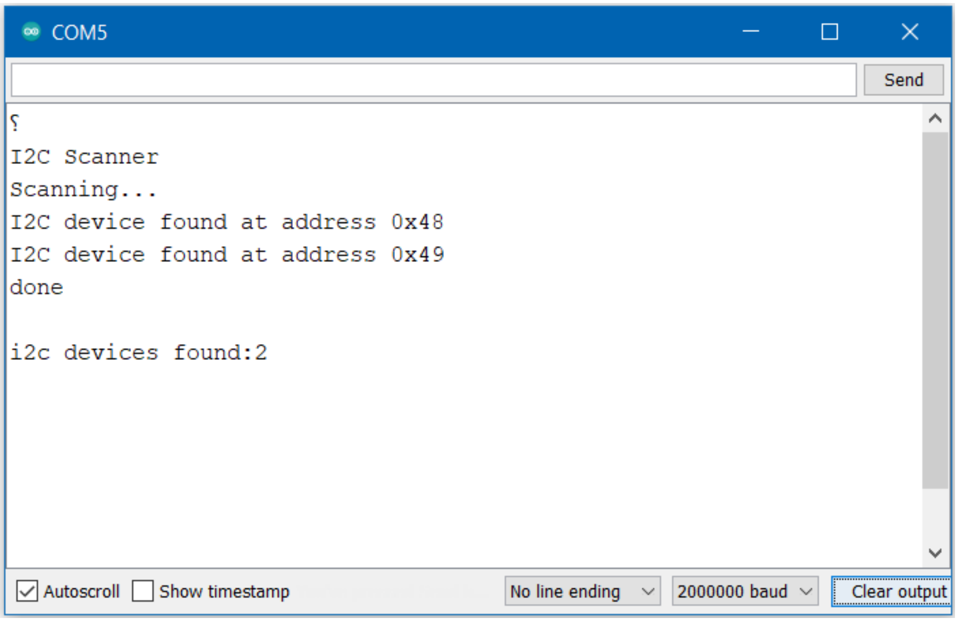
*Figure 49 - Oral presentation demo circuit diagram*

## Two ADS1015 ADCs

To recap – at this stage in the project, readings were being taken from one ADS1015 ADC at a sampling rate of 1200sps. Through some minor tweaks to the Adafruit ADC library earlier, the maximum sps rate achieved was 3300sps. Since the ADS1015 was being used in continuous mode, only one analogue input could be used on that ADC at one time. The first ADC was being used for the voltage measurements and hence it was critical to this project to get the second ADC working in tandem, to facilitate voltage measurements (representing current) to be taken simultaneously.

The ADS1015 comes with an address pin, which facilitates the ADC I2C address to be changed, depending on what is hooked up to the address pin.

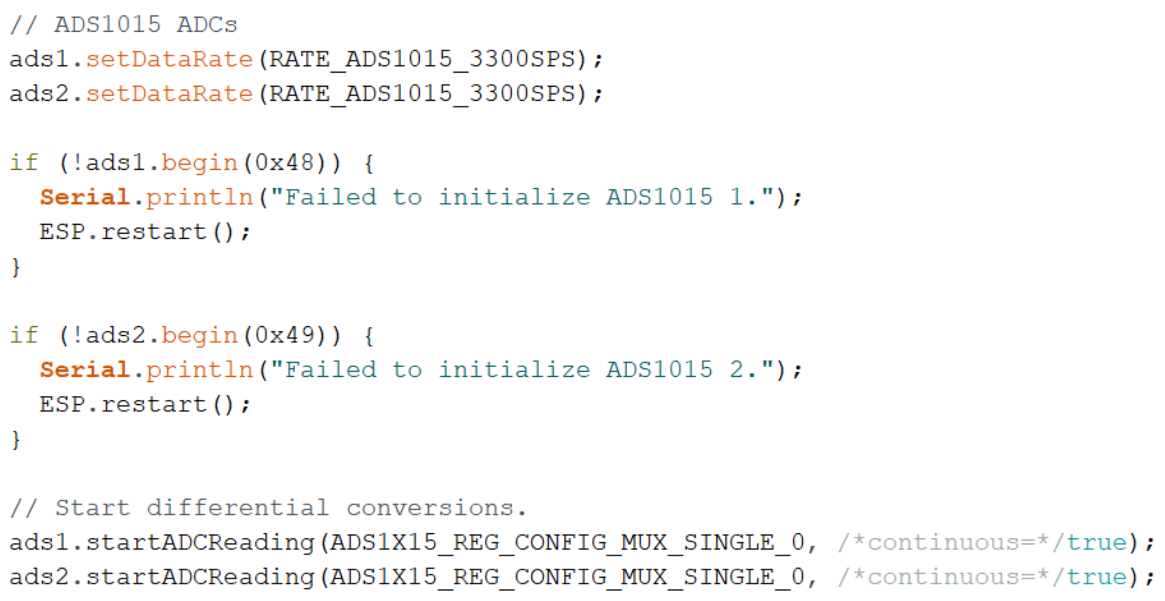
For ADC 1, the GND signal was hooked up to the address pin, giving ADC 1 an I2C address of "0x048". For ADC 2, the Vin signal was hooked up to the address pin, giving ADC 2 an I2C address of "0x049".



```
ʕ
I2C Scanner
Scanning...
I2C device found at address 0x48
I2C device found at address 0x49
done

i2c devices found:2
```

*Figure 50 - Serial monitor output for I2C Scanner*

An I2C Scanner program was run on the ESP32 to check that both ADCs were being picked up by the ESP32 over the I2C communication interface successfully. Figure 50, as shown above, clearly shows that the ESP32 was able to detect the two ADCs over the I2C interface. To differentiate between the two ADCs in the ESP32 code, the ADS1015 objects were initialised with their I2C device addresses, as showcased previously.



```
// ADS1015 ADCs
ads1.setDataRate(RATE_ADS1015_3300SPS);
ads2.setDataRate(RATE_ADS1015_3300SPS);

if (!ads1.begin(0x48)) {
  Serial.println("Failed to initialize ADS1015 1.");
  ESP.restart();
}

if (!ads2.begin(0x49)) {
  Serial.println("Failed to initialize ADS1015 2.");
  ESP.restart();
}

// Start differential conversions.
ads1.startADCReading(ADS1X15_REG_CONFIG_MUX_SINGLE_0, /*continuous=*/true);
ads2.startADCReading(ADS1X15_REG_CONFIG_MUX_SINGLE_0, /*continuous=*/true);
```

*Figure 51 - ESP32 code to initialise the ADCs*

The code to initialise the ADCs can be seen in Figure 51, above. Initially, both ADCs are set to a sampling rate of 3300sps then the two ADCs are initialised with their I2C address passed as the begin() argument. If both ADCs are successfully initialised, then the continuous readings for both ADCs are started.

The ESP32 was previously reading samples from ADC 1 at 1200sps and then the coded was updated to take samples from both ADC 1 and 2 at 1200sps each. This means that the number of samples that the ESP32 had to handle and compute with was doubled. Through some program efficiency improvements, the ESP32 was successfully able to handle voltage readings and calculations from both of the ADCs simultaneously. This is thanks to the strong computing power of the ESP32 and its large amount of memory – this was a significant reason behind choosing the ESP32 as the MCU of choice originally.

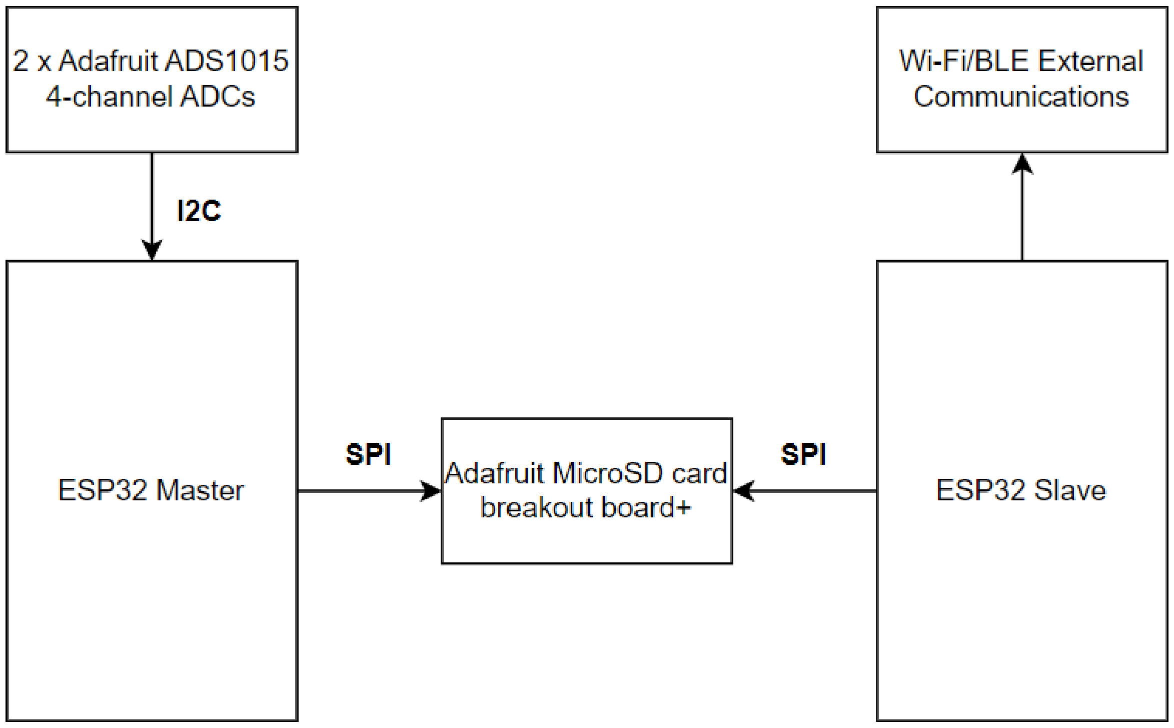
## External Communications – Introducing a slave ESP32

As discussed in the last section, the ESP32 can handle 1200sps readings from two ADCs simultaneously, but the desired functionality of the project involves further functionality – microSD card storage, GPS, Wi-Fi/BLE server, cloud storage, etc. It is clear that, no matter how strong the ESP32 is, it will not be able to handle the ADC readings and Wi-Fi tasks simultaneously. Therefore, a second ESP32 was sourced and the potential architecture between the two ESP32s was theorised.

Architecture Requirements:

- The first desired requirements of the interlinking architecture followed the fact that the master ESP32 should take the ADC readings and transfer them to the slave ESP32 by whatever means, which in turn will use those readings for Wi-Fi-related purposes.
- The GPS interrupt timing disciplining is required to be wired to the master ESP32 as it is the one actually taking the readings.
- Storing the values on the microSD card breakout could potentially be completed by either of the ESP32s

The first theorised attempt at communication between the master and slave ESP32s involved the master sending the useful live measurements to the MicroSD card via SPI then the slave would in turn read those values from the MicroSD card. This would mean that the master would only have to store the values to the MicroSD card, without having to directly transfer them to the slave ESP32, hence allowing the master to dedicate more processing potential to the ADC readings. Since the master is already storing the readings to the microSD card in a suitable long-term format, the slave ESP32 would only have to read from the microSD breakout. Essentially, this proposed architecture would cover two essential bases, with only one process.



2 x Adafruit ADS1015 4-channel ADCs
I2C
ESP32 Master
SPI
Adafruit MicroSD card breakout board+
SPI
ESP32 Slave
Wi-Fi/BLE External Communications

*Figure 52 - Flowchart of the first proposed master slave architecture*

Figure 52,  as shown above, showcases a flowchart of the first proposed master slave architecture. This architecture would have been ideal, however, on deeper research it was found that SPI communication protocol can only have one master. This proposed architecture, however, had both ESP32s set up as masters to the microSD card breakout slave. Due to the nature of SPI – particularly the CS timing [22] – this architecture was found to be infeasible.

| Protocol | UART | I2C | SPI |
|---|---|---|---|
| Complexity | Simple | Easy to chain multiple devices | Complex as number of devices increases |
| Speed | Slowest due to no clock signal | Faster than UART | Fastest |
| Number of wires | 1 | 2 | 4 |
| Number of devices | Up to 2 devices | Up to 127, but gets complex | Many, but gets very complex |
| Number of masters and slaves | Single to Single | Multiple slaves and masters | 1 master, multiple slaves |

*Figure 53 - Communication protocol comparison table*

To decide on the optimal communication protocol, the three types of possible communication protocols were researched thoroughly. Figure 53, as shown above, showcases a comparison table of the 3 protocols.

Since the microSD card could not be used as a passthrough for the data, it was decided that the slave ESP32 should be the MCU that actually stores the values to the microSD card, to relieve the master ESP32 from additional functionality – as the master has to allocate a lot of computational power to the ADC readings. Since the slave will then be using SPI to store the values to the microSD card, it became clear from the research that using SPI directly between the master and slave as well would not be feasible, due to the nature of the clock timings requiring one clock signal only. This may be possible to do, however, this project is limited in terms of time, so it is crucial to reduce complexities where possible.

That leaves I2C and UART as the other possible communication protocols to be used between the master and slave. As shown in Figure 53, it is clear that I2C is faster than UART. This is partly due to the fact that UART does not have a clock signal [22]. In this project, it is quintessential that the speed of the data transfer between the master and slave is as high as possible, to reduce delays from affecting the ADC readings and Wi-Fi/BLE functionality of the master and slave, respectively. Therefore, the next proposed architecture was to utilise the I2C communications interface to send data from the master ESP32 to the slave ESP32.

Figure 55, as shown below, showcases the second proposed architecture, as discussed. The first step towards engineering this architecture was to wire the master to the slave and see if the master could detect the slave on the I2C bus.

Figure 54, as shown below, showcases the Fritzing circuit diagram for connecting the ESP32 master and slave via I2C.

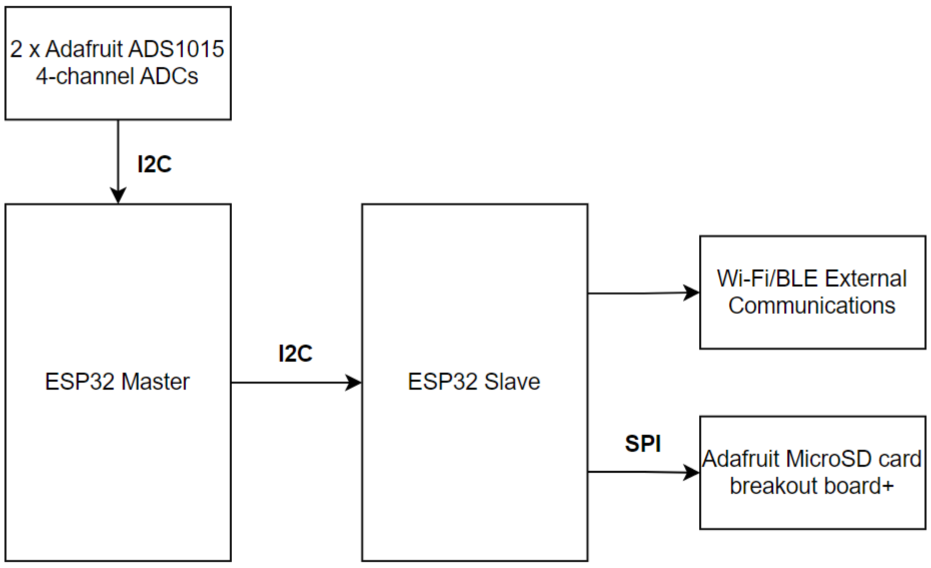

2 x Adafruit ADS1015 4-channel ADCs
I2C
ESP32 Master
I2C
ESP32 Slave
Wi-Fi/BLE External Communications
SPI
Adafruit MicroSD card breakout board+

*Figure 55 - Second proposed external communications architecture*

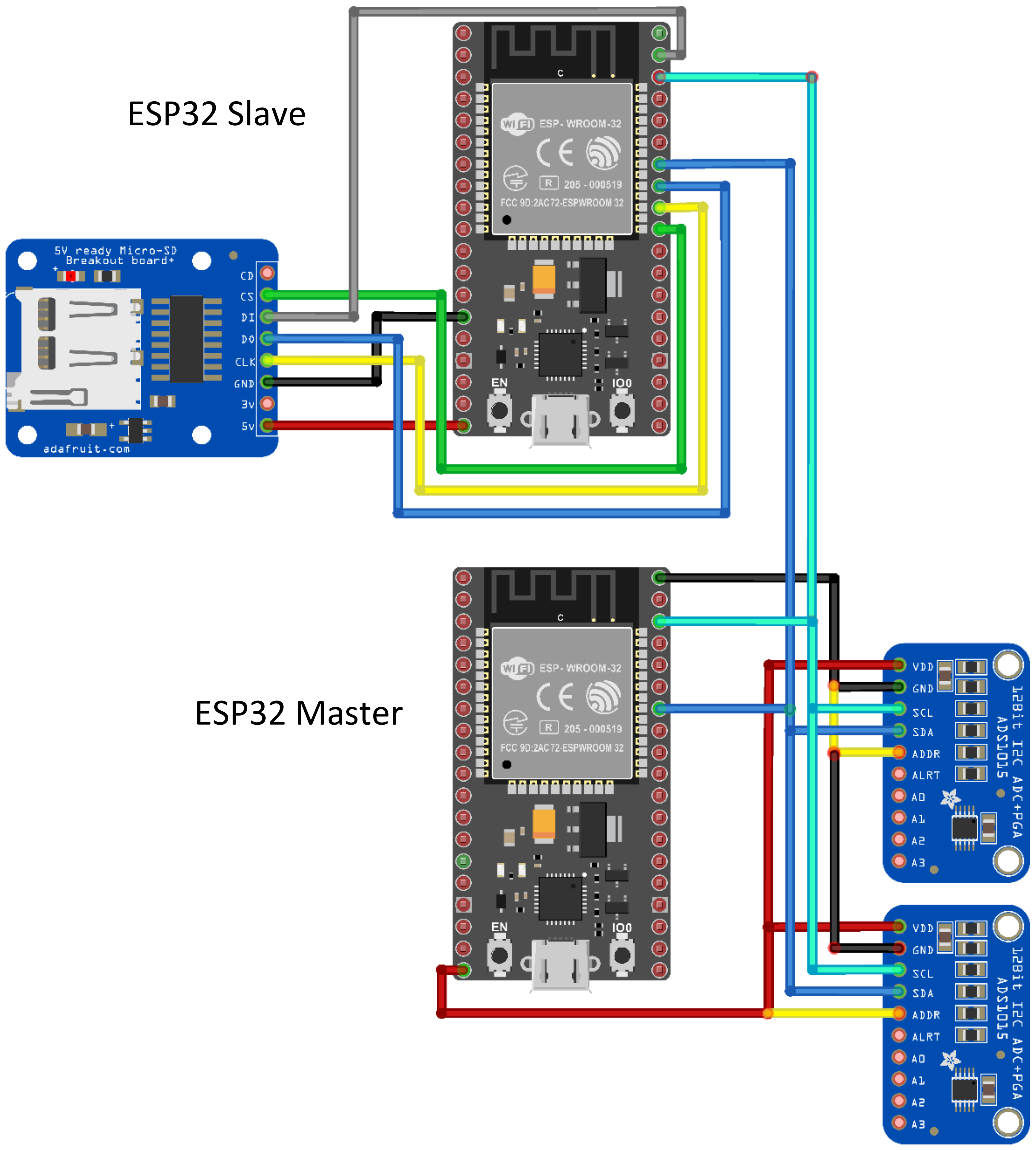
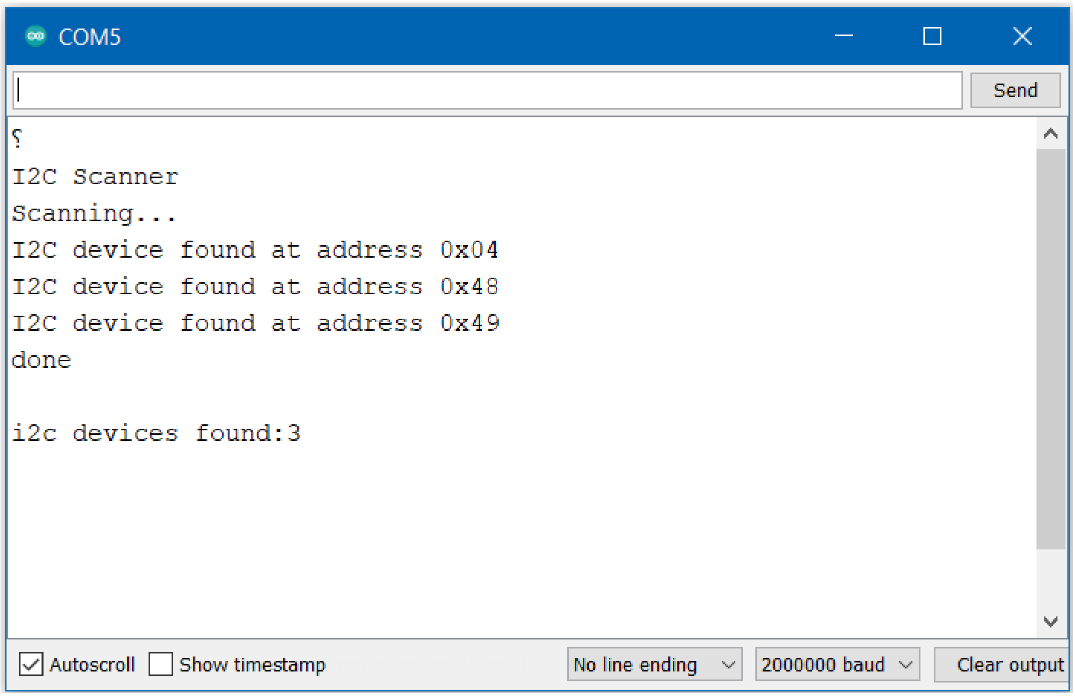


*Figure 54 - Fritzing circuit diagram for connecting the ESP32 master and slave via I2C.*



```
ʕ
I2C Scanner
Scanning...
I2C device found at address 0x04
I2C device found at address 0x48
I2C device found at address 0x49
done

i2c devices found:3
```

*Figure 56 - Arduino IDE Serial Monitor output of the I2C Scanner program*

Figure 56, as shown above, showcases the Arduino IDE Serial Monitor output of the I2C Scanner program. 0x48 and 0x49 are the two ADS1015 ADCs and it's clear from the Serial Monitor that the slave ESP32 is under the address 0x04.

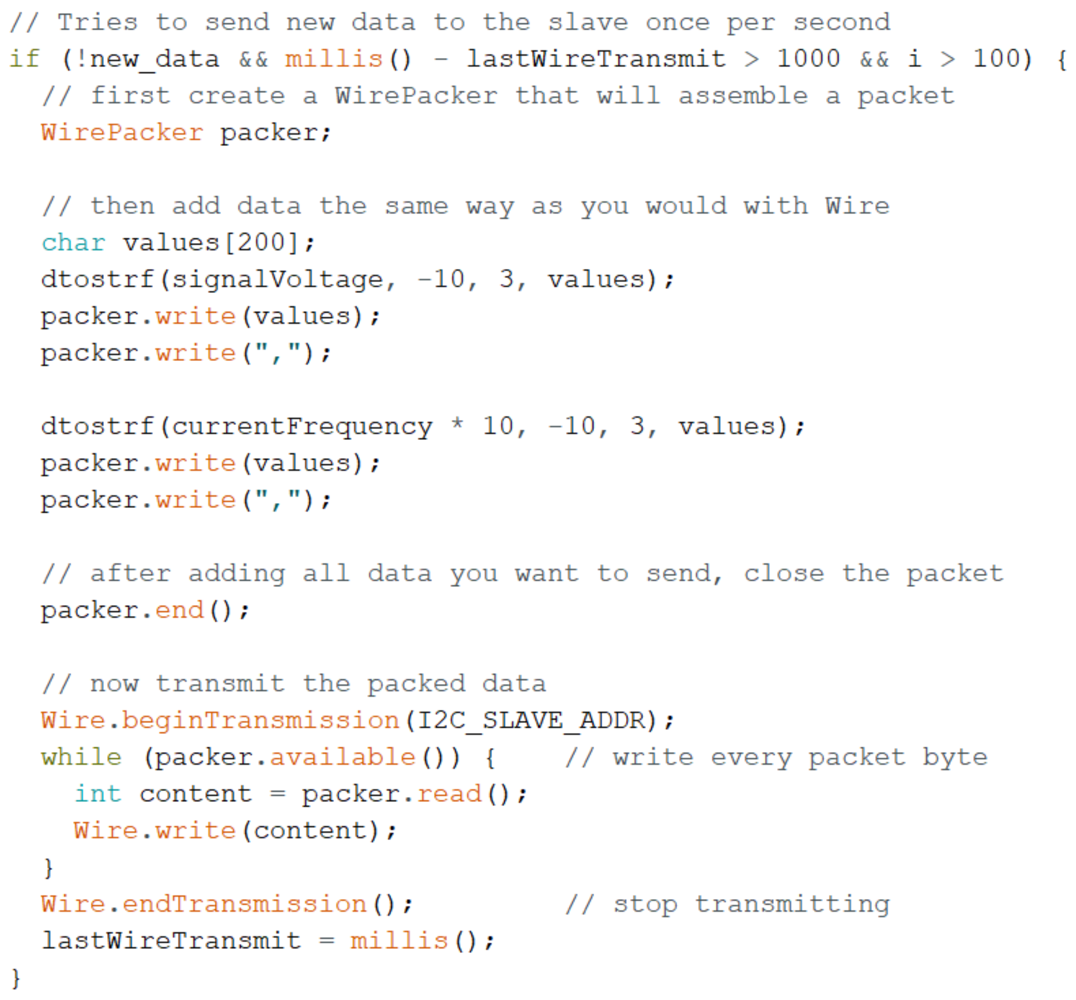
The next goal was to get simple data transfers working from the master ESP32 to the slave ESP32. A small library called "ESP32_I2C_Slave" [23] was utilised to get some very basic data transferring between the master and slave. This library was archived in 2021 and it isn't very popular, so the library was used with skepticism initially, however, the functionality of the library worked perfectly and hence, didn't warrant the need to use a different library.

Figure 57, as shown below, showcases the I2C transfer code written for the master ESP32. It is key to note that most of the values have been excluded from this screenshot. For demonstration purposes, only "signalVoltage" and "currentFrequency" are included in this example. The full code can be found in Appendix 9.

Once the master ESP32 has taken readings from the ADCs and calculated all the useful live measurements, it will check if it has been more than a second since the last set of values was sent to the slave. The WirePacker class will be utilised to store a string of characters as a char array. The packet will then be read and written to the I2C bus by utilising the Wire class.

The slave code is a bit simpler. The slave essentially checks the I2C bus repeatedly for new updates. If the master has sent data to the slave via I2C, then the slave will parse over that data to extract the values into variables, then will use those variables in the code. The slave I2C code can be seen in Appendix 10.

The ADC readings were then thoroughly tested and analysed to ensure that the new I2C communications, between the master and slave, had no effect on the ADC readings themselves.



```
// Tries to send new data to the slave once per second
if (!new_data && millis() - lastWireTransmit > 1000 && i > 100) {
  // first create a WirePacker that will assemble a packet
  WirePacker packer;

  // then add data the same way as you would with Wire
  char values[200];
  dtostrf(signalVoltage, -10, 3, values);
  packer.write(values);
  packer.write(",");

  dtostrf(currentFrequency * 10, -10, 3, values);
  packer.write(values);
  packer.write(",");

  // after adding all data you want to send, close the packet
  packer.end();

  // now transmit the packed data
  Wire.beginTransmission(I2C_SLAVE_ADDR);
  while (packer.available()) {    // write every packet byte
    int content = packer.read();
    Wire.write(content);
  }
  Wire.endTransmission();         // stop transmitting
  lastWireTransmit = millis();
}
```

*Figure 57 - I2C transfer code written for the master ESP32*

## MicroSD Card writing from the slave ESP32

The slave ESP32 was wired to the MicroSD card breakout via SPI, in the same format as shown in semester 1 of the project. The values being received over I2C from the master are parsed into variables and then they are sent to the microSD card via SPI. To format these variables, the C++ snprintf function was utilised. Initially, these values were being written to a singular microSD card text file called Smart_Wireless_Sensor_Readings.txt. This file could very quickly become populated with data and hence reach an impractical size. A suitable solution to this problem was solved later in the project.

## External Communications Overview

Now that the master was sending data to the slave – which was in turn sending formatted data to the microSD card – it was time to connect the slave ESP32 to the internet. The UE model of this ESP32 devkit comes with a U.FL antenna port, to facilitate an external antenna to be connected to the board. It was found that the built-in internal antenna had very poor signal and wall penetration, therefore an external antenna was connected via the U.FL port. The slave could successfully scan for nearby wireless networks, however, it struggled to connect to a Wi-Fi network. This was solved by swapping the master ESP32 board with the slave ESP32 board. The ADC circuit functionality was re-tested to ensure that swapping the boards had no effect on the readings and communications between the boards and breakouts. The now-swapped slave board was successfully able to connect wirelessly to the local network via a nearby router.

Now that the slave was able to connect to the internet, it was time to design the external communications architecture. Throughout the project, the type of server had been described as potentially either being a Wi-Fi server or a BLE server. Through some brief research, it was found that the ESP32 cannot run Wi-Fi and Bluetooth server capabilities simultaneously [24]. Having access to the internet for this project was decided as an essential component, due to future functionalities such as cloud uploads and updating the board time to be in sync with GMT+1. Therefore, the idea of using a BLE server was eradicated.

A goal of this project is to have easily accessible readings for the smart wireless sensor in all scenarios. If the ESP32 was set up as a station for a web server, this means that a user connected to the same network as the ESP32 would be able to access the ESP32 web page through the ESP32 Ip address. This was seen as a valid potential option, however, if the smart wireless sensor was in a scenario where it would not have access to a local network or the internet, then this solution would be totally useless.

First, a solution was needed for this scenario where the smart wireless sensor would not have access to a local network/the internet. The initial solution found was to have the ESP32 act as an access point, rather than as a station.

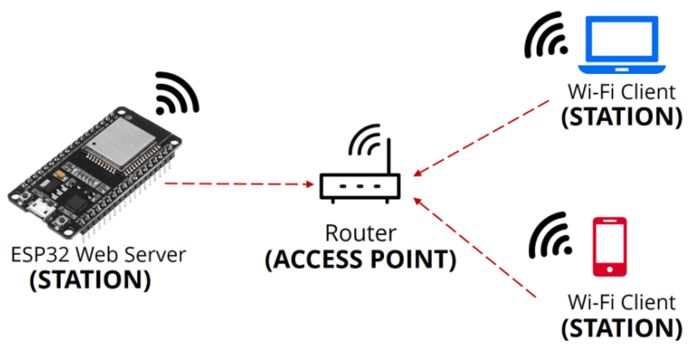


*Figure 58 - Flowchart for the ESP32 as a station*

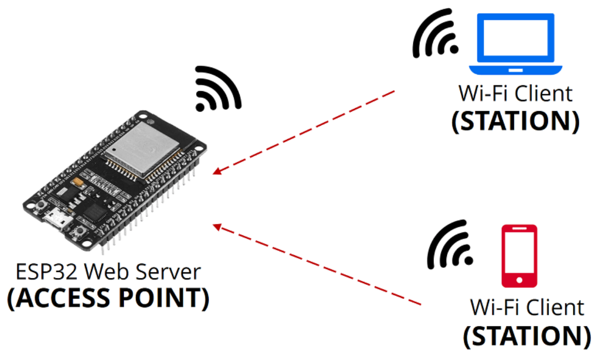


*Figure 59 - Flowchart for the ESP32 as an access point*

Figure 59 shows the flow chart for the ESP32 as a station (hotspot) and Figure 58 shows the flow chart for the ESP32 as a soft access point. With the access point architecture, this removes the need for clients to connect to the router to access the ESP32 web server – the clients can directly connect to the 2.4GHz Wi-Fi signal being transmitted from the ESP32 itself if they are within the 2.4GHz signal range. Using the ESP32 as an access point was found to be optimal solution for those scenarios where the smart wireless sensor would be unable to connect to the local network, however, it was found that the ESP32 can be in both access point mode and station mode simultaneously by using the "WIFI_MODE_APSTA" mode. It was also found that it would take no real extra computing power to set up the ESP32 web server over the local network too – so the ESP32 was set up as both an Access Point and a Station, simultaneously.

This meant that users would be able to access the ESP32 readings via the ESP32 Web Server by both of the architectures shown in Figures 59 and 58. The users will be able to access the Web Server from anywhere in the substation, either by standing nearby to the ESP32 and connecting to its own network or by connecting to the local Wi-Fi network and accessing it that way.

However, what if someone wanted to read the sensor values from anywhere outside the substation, or, even, from a different country? The solution is to utilise the ESP32 Wi-Fi features to send the useful live measurements to a cloud-hosted website, thus enabling users to access the website and measurements worldwide, at any time and any geo-location. The live measurements can then be easily stored in cloud data storage, for later access.

In summary, users will be able to read the useful live measurements through multiple different ways, as explained in these two following sections:

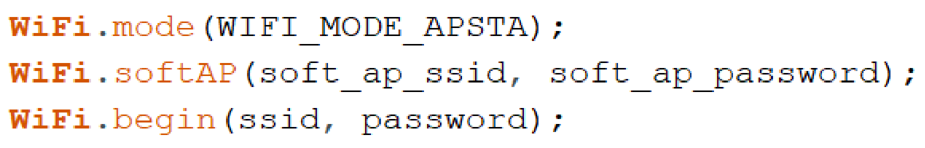
- First half: The ESP32 Web Server which can be accessed via the ESP32 Station and/or the ESP32 Access Point
- Second Half: The cloud-hosted website and cloud web storage

## External Communications First Half – Simultaneous Access Point and Station Wi-Fi Web Servers

A simple asynchronous web server was set up and tested in semester 1, however, this would type of web server would be inadequate for this external communications architecture. The way the web server in semester 1 was set up, was where a client would directly connect to the ESP32 via means of TCP, then the ESP32 would repeatedly send html code to the client via HTTP requests so that a web page would be displayed on the client device.

The slave ESP32 has a lot of functionality to deal with. It has to constantly check the I2C interface for new readings from the master ESP32, it has to take those readings then parse them and store them in the microSD card, it has to connect to the Wi-Fi to obtain the current time for the file/folder name system and in future it will be uploading the sensor measurements to a cloud website and data storage. This is clearly a lot functionality weighing on the slave ESP32 already. The web server that was set up in semester 1 needs the ESP32's full attention due to the nature of the greedy TCP connection – as the ESP32 is constantly sending new html code the client to up the asynchronous properties on the page – and hence, the ESP32 would not be to perform its other required tasks while its greedily connected to the client.

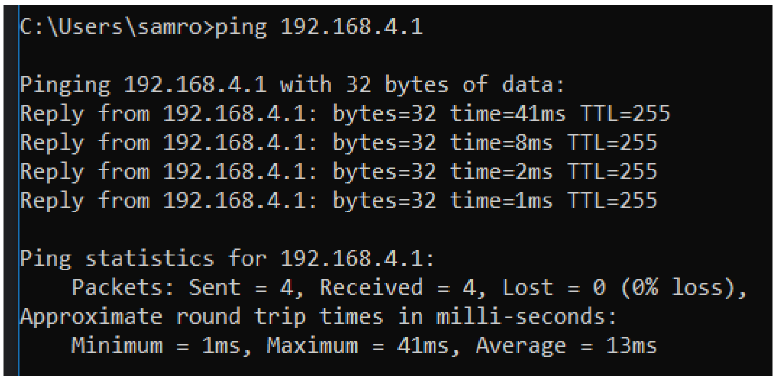
The solution found is to still use an ESP32 web server, however, the web server will not function as an asynchronous web page, but rather as a simple HTTP endpoint that will return the useful live measurements to the user in the HTTP response body.



```
WiFi.mode(WIFI_MODE_APSTA);
WiFi.softAP(soft_ap_ssid, soft_ap_password);
WiFi.begin(ssid, password);
```

*Figure 60 - ESP32 server setup code*

Figure 60, as shown above, showcases some of the ESP32 setup code. The ESP32 was setup in "WIFI_MODE_APSTA" mode to specify that the ESP32 should be able function as both a station and access point. The WiFi.softAP() method sets up the soft access point with a custom ssid and password. The WiFi.begin() method then sets up the station with the local network ssid and password that were defined in the code earlier. Once the three lines of code in Figure 60 have run, the access point and the station should be up and running, however, there has not been an web server functionality defined for those points so they must be health-checked in a specific way, as shown below.



```
C:\Users\samro>ping 192.168.4.1

Pinging 192.168.4.1 with 32 bytes of data:
Reply from 192.168.4.1: bytes=32 time=41ms TTL=255
Reply from 192.168.4.1: bytes=32 time=8ms TTL=255
Reply from 192.168.4.1: bytes=32 time=2ms TTL=255
Reply from 192.168.4.1: bytes=32 time=1ms TTL=255

Ping statistics for 192.168.4.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 41ms, Average = 13ms
```

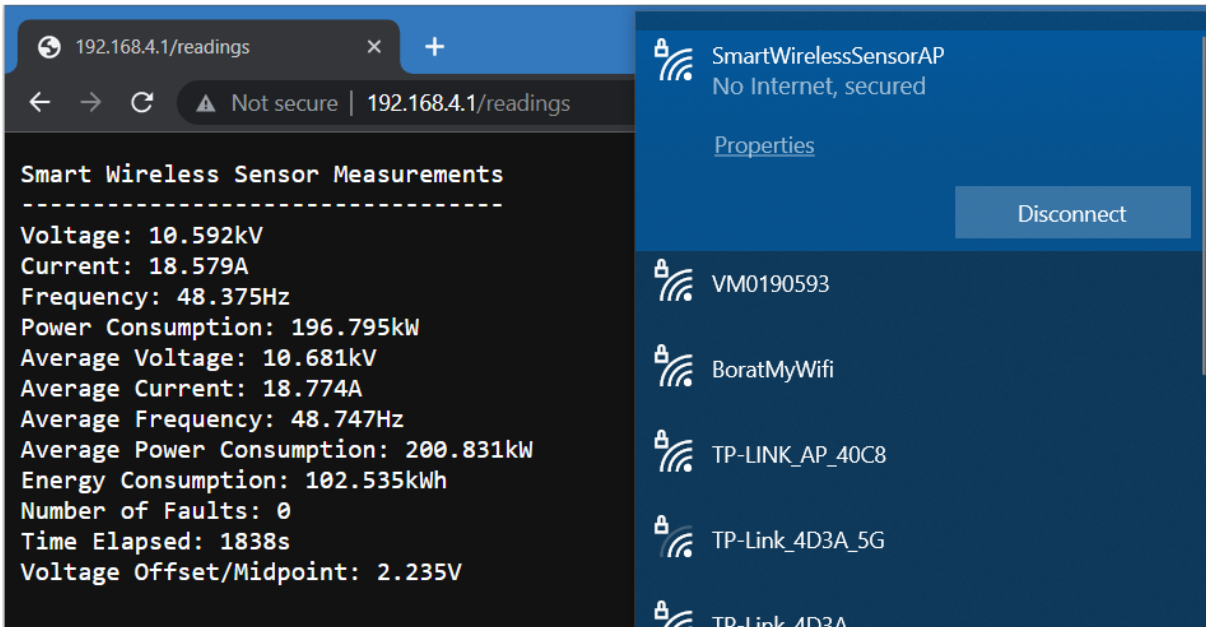
*Figure 61 - Command line output after the soft AP IP address has been pinged successfully*

The AP and the station IP addresses can be pinged at this stage, to ensure that they are up and running. Figure 62, as shown above, showcases the command line output after the soft AP IP address has been pinged successfully.

```
if (ON_STA_FILTER(request)) {
  request->send(200, "text/plain", message);
  return;

} else if (ON_AP_FILTER(request)) {
  request->send(200, "text/plain", message);
  return;
}
```

*Figure 62 - Some of the AP and station web server setup code*

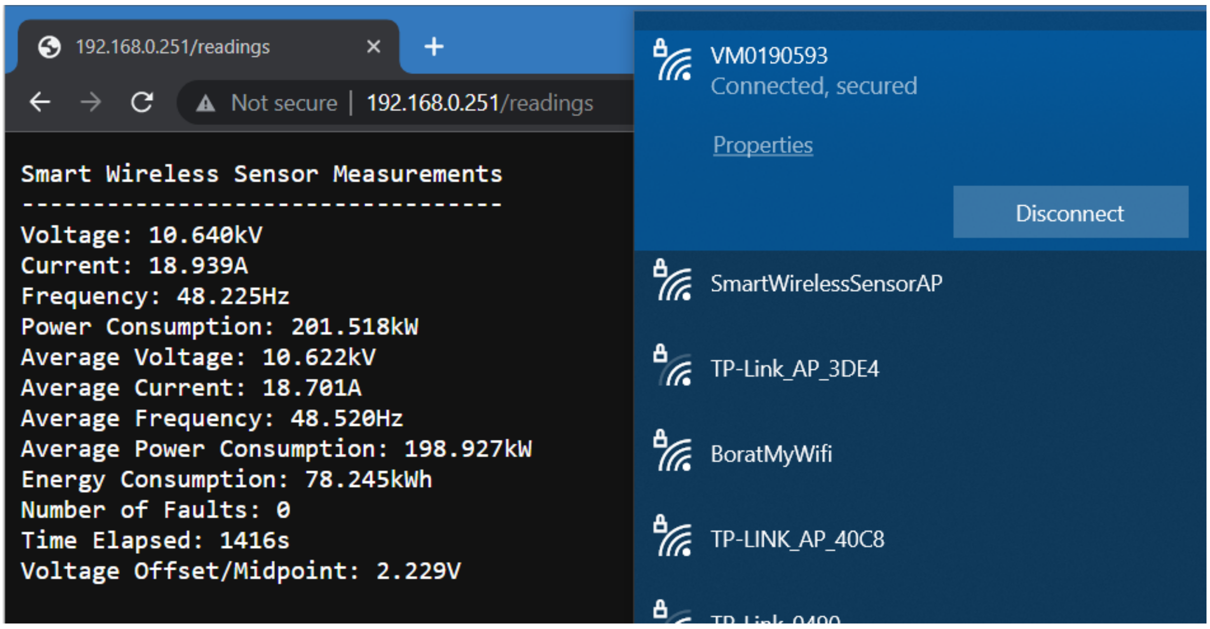
Figure 62,  as shown above, showcases some of the AP and station web server setup code. This code is part of the server.on() function call. This code essentially specifies what the ESP32 should do whenever an HTTP request in the correct format is sent to either the network IP address (through the station) or to the ESP32 IP address (through the access point). In both cases, the HTTP response should be the same. A HTTP 200 “OK” response is returned with the 12 useful live measurements returned in the HTTP body, as earlier defined by the “message” variable.



```
Smart Wireless Sensor Measurements
----------------------------------
Voltage: 10.592kV
Current: 18.579A
Frequency: 48.375Hz
Power Consumption: 196.795kW
Average Voltage: 10.681kV
Average Current: 18.774A
Average Frequency: 48.747Hz
Average Power Consumption: 200.831kW
Energy Consumption: 102.535kWh
Number of Faults: 0
Time Elapsed: 1838s
Voltage Offset/Midpoint: 2.235V
```

*Figure 63 - Laptop connected directly to the ESP32 Access Point*

Figure 63, as shown above, showcases a screenshot of a laptop **connected directly to the ESP32 Access Point** through the laptop’s Wi-Fi peripherals. It then shows the HTTP request sent to the ESP32 in the chrome browser to “192.168.4.1/readings”. Finally, it shows the HTTP body response in the chrome main page, which was returned directly from the ESP32 to the laptop. All 12 readings are returned in a clear, concise format for easy readability.



```
Smart Wireless Sensor Measurements
----------------------------------
Voltage: 10.640kV
Current: 18.939A
Frequency: 48.225Hz
Power Consumption: 201.518kW
Average Voltage: 10.622kV
Average Current: 18.701A
Average Frequency: 48.520Hz
Average Power Consumption: 198.927kW
Energy Consumption: 78.245kWh
Number of Faults: 0
Time Elapsed: 1416s
Voltage Offset/Midpoint: 2.229V
```

*Figure 64 - Laptop connected to the same local Wi-Fi network that the ESP32 was connected to*

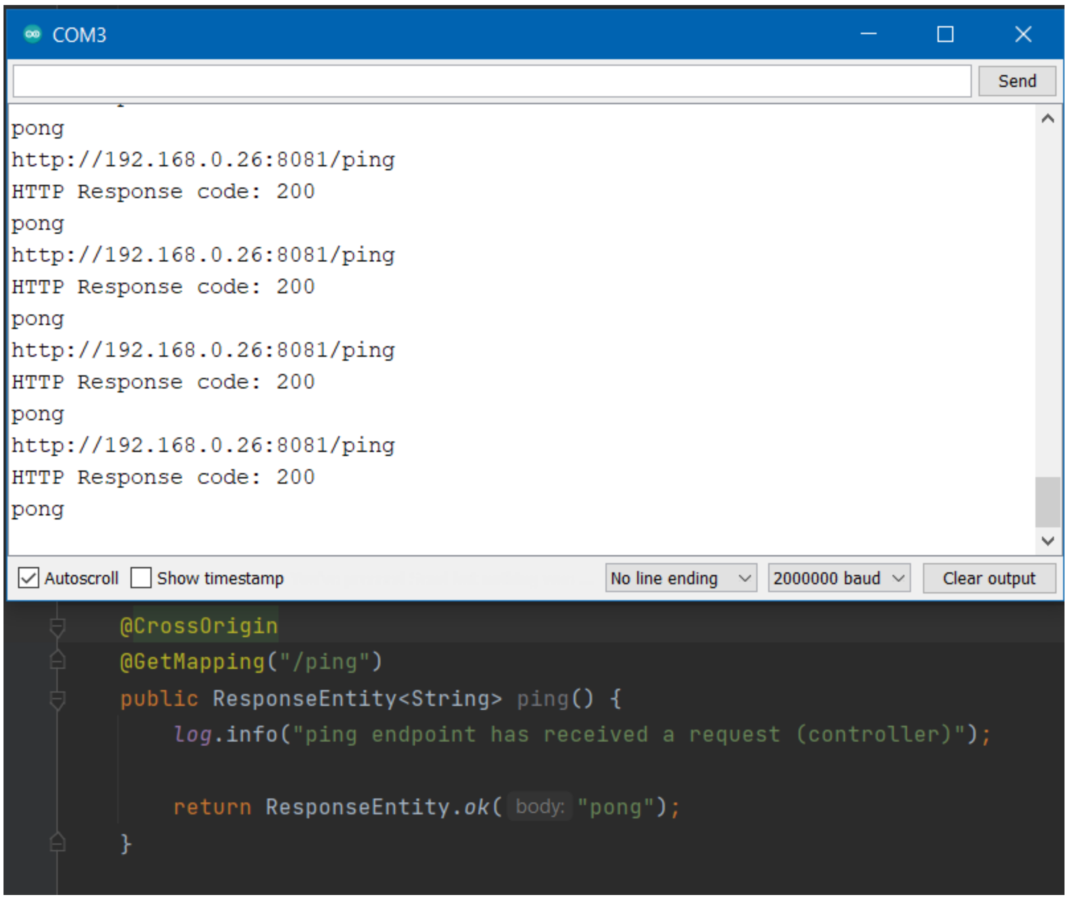
Figure 64, as shown above, showcases a screenshot of a laptop **connected to the same local Wi-Fi network that the ESP32 was connected to**. It then shows the HTTP request sent in the chrome browser over the local Wi-Fi network to the ESP32 to the “192.168.0.251/readings” address, which is a different IP address from the ESP32 access point as the ESP32 was hosting the web server as a station on a different IP address to that of the access point. Finally, it shows the HTTP body response in the chrome main page, which was returned directly from the ESP32 to the laptop. All 12 readings are returned in a clear, concise format for easy readability.

## External Communications Second Half – Cloud website and storage flow

To send the data from the slave ESP32 to a cloud-hosted website and to cloud bucket storage, there are a few different hoops that the data must jump through to reach its end destination. React.js/html/css was chosen as the most suited framework for the website functionality due to React's lightweight, yet asynchronous nature. This will facilitate asynchronous measurement updates to the website.

The facts were then laid on the table. The ESP32 can send and receive HTTP requests across the local network or through the access point mode. The React.js side of the website can also send HTTP requests, however, the React.js website cannot directly send or receive HTTP requests to or from the ESP32, and hence, cannot receive HTTP requests from the ESP32. This fact specified the need for a suitable back-end. A possible back-end choice for the React.js website could be Node.js, since React is native to Node, however, a Java/Spring Boot backend application was chosen due to its familiarity. Spring Boot is a backend Java-based framework used for creating microservices [25].

Before setting up the more-complex endpoints for transferring values between the ESP32 and the backend, a simple "ping" endpoint was set up in the Spring Boot application. This simple endpoint returns an "OK" (200) HTTP response with a body containing the string "pong". This is a very useful endpoint to test the APIs to ensure that they can communicate as expected. Figure 65, as shown below, showcases the ESP32 successfully hitting the ping endpoint in the backend Spring Boot application.



```
pong
http://192.168.0.26:8081/ping
HTTP Response code: 200
pong
http://192.168.0.26:8081/ping
HTTP Response code: 200
pong
http://192.168.0.26:8081/ping
HTTP Response code: 200
pong
http://192.168.0.26:8081/ping
HTTP Response code: 200
pong
```

```java
@CrossOrigin
@GetMapping("/ping")
public ResponseEntity<String> ping() {
    log.info("ping endpoint has received a request (controller)");

    return ResponseEntity.ok( body: "pong");
}
```

*Figure 65 - the ESP32 successfully hitting the ping endpoint in the backend Spring Boot application*

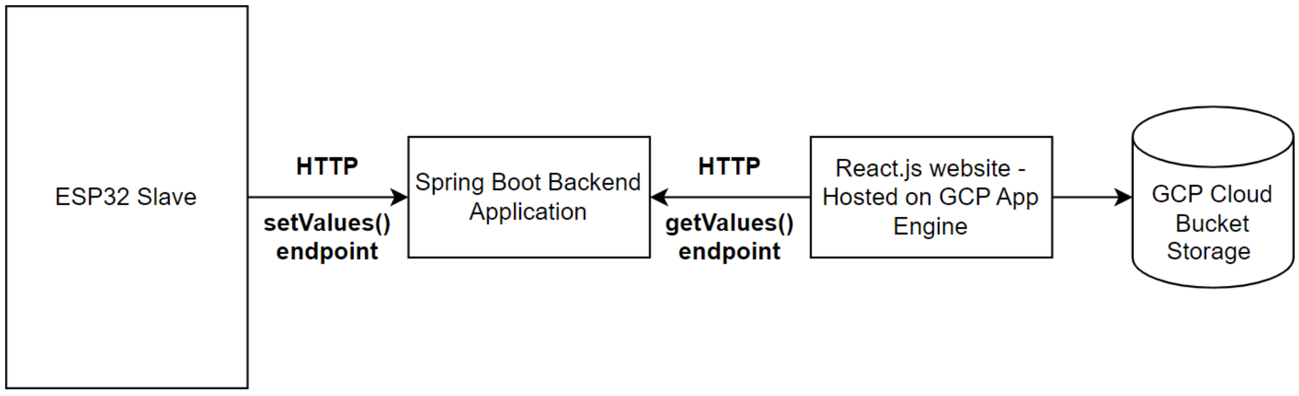


*Figure 66 - Architecture that was decided on for the cloud data flow from the ESP32 through to the cloud web-hosting and database storage*

**Spring Boot Backend**

Figure 66, as shown above, showcases the architecture that was decided on for the cloud data flow from the ESP32 through to the cloud web-hosting and database storage. The ESP32 sends an HTTP POST request to the **setValues()** endpoint of the Spring Boot application. This POST request contains the 12 useful live measurements and the Java app then parses those variables into local Java variables. Since the Spring Boot/Java app is continuously running in Google App Engine, whenever the React.js website sends an HTTP GET request to the **getValues()** endpoint Spring Boot backend application, the values of the variables in the Spring Boot app (that were last set by the ESP32 through the setValues() endpoint) will then be returned to the React website in JSON format as part of a "SensorData" object, which contains the 12 values.

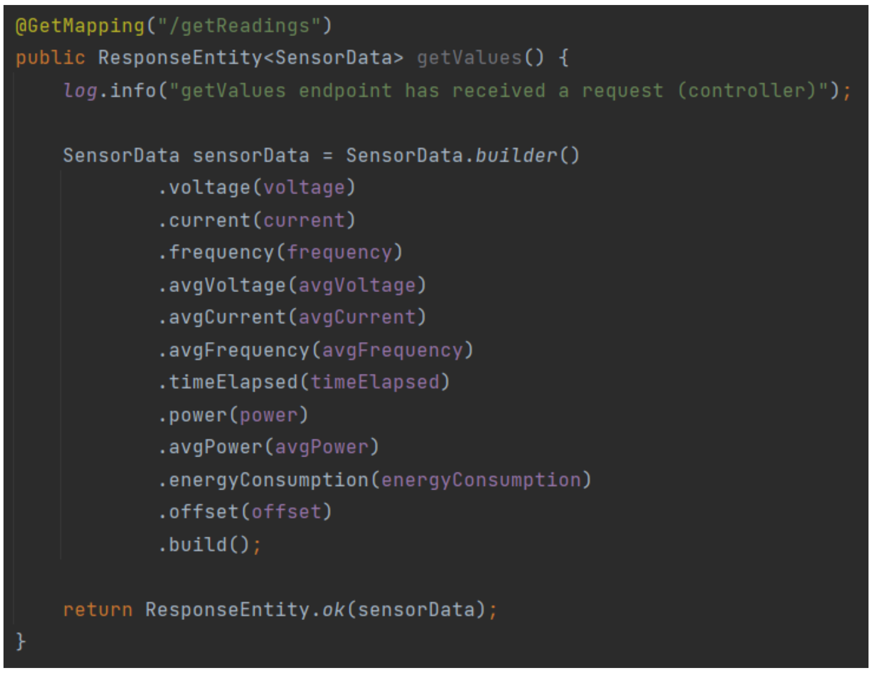

```java
@GetMapping("/getReadings")
public ResponseEntity<SensorData> getValues() {
    log.info("getValues endpoint has received a request (controller)");

    SensorData sensorData = SensorData.builder()
            .voltage(voltage)
            .current(current)
            .frequency(frequency)
            .avgVoltage(avgVoltage)
            .avgCurrent(avgCurrent)
            .avgFrequency(avgFrequency)
            .timeElapsed(timeElapsed)
            .power(power)
            .avgPower(avgPower)
            .energyConsumption(energyConsumption)
            .offset(offset)
            .build();

    return ResponseEntity.ok(sensorData);
}
```

*Figure 67 - The Java code for the getValues() endpoint in the Spring Boot application*

Figure 67, as shown above, showcases the Java code for the getValues() endpoint in the Spring Boot application. The endpoint functionality is very basic – whenever the endpoint is called by the React app, it builds a SensorData object (using the local variable values) then returns the SensorData object to the React app. When instantiating Plain Old Java Objects (POJOs) in Java, each field must be set manually after instantiating the object. The Lombok library [26] was utilised with the SensorData class to facilitate the use of builder() methods for instantiation and value assigning of SensorData objects, as shown in Figure 67, above. The full Java/Spring Boot code can be seen in Appendix 12.

The URL of the getValues() endpoint is set at “**{baseUrl}/getReadings”**. When running the application locally, either of the following URLs can be used to hit the getReadings() endpoint:

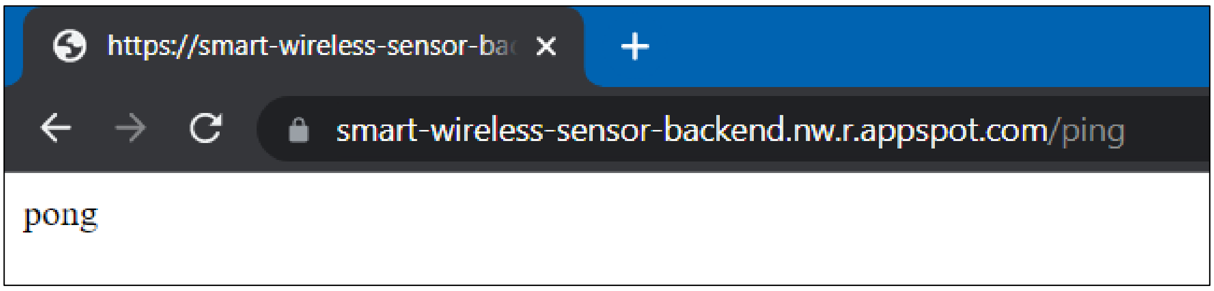
- **“localhost:8081/getReadings”**
- **“http://{ipAddress}:8081/getReadings”**

In the Google Cloud Platform (GCP) App Engine production instance however, the URL would be **“https://smart-wireless-sensor-backend.nw.r.appspot.com/getReadings”.** This URL can only be called by the React App as a cross origin specification was set up to only allow API calls from the React App to work. This was set up using the @CrossOrigin Spring Boot annotation and was done so to prevent anyone from maliciously spamming the website, hence, messing up the readings.

To test the sending of API calls to the production instance, the “ping” endpoint has the cross origin API calls set to any origin, therefore, the ping endpoint can be called directly by using the following link:

https://smart-wireless-sensor-backend.nw.r.appspot.com/ping

The ping endpoint should return an “OK” (200) HTTP response with body containing the string “pong”, as shown earlier. This can be validated be visiting the link and the response will be similar to Figure 68, shown below. This ping endpoint is useful to test if the Spring Boot instance is up and running in the GCP App Engine cloud.


https://smart-wireless-sensor-ba
smart-wireless-sensor-backend.nw.r.appspot.com/ping

pong

*Figure 68 – HTTP response returned from GCP Backend ping endpoint*

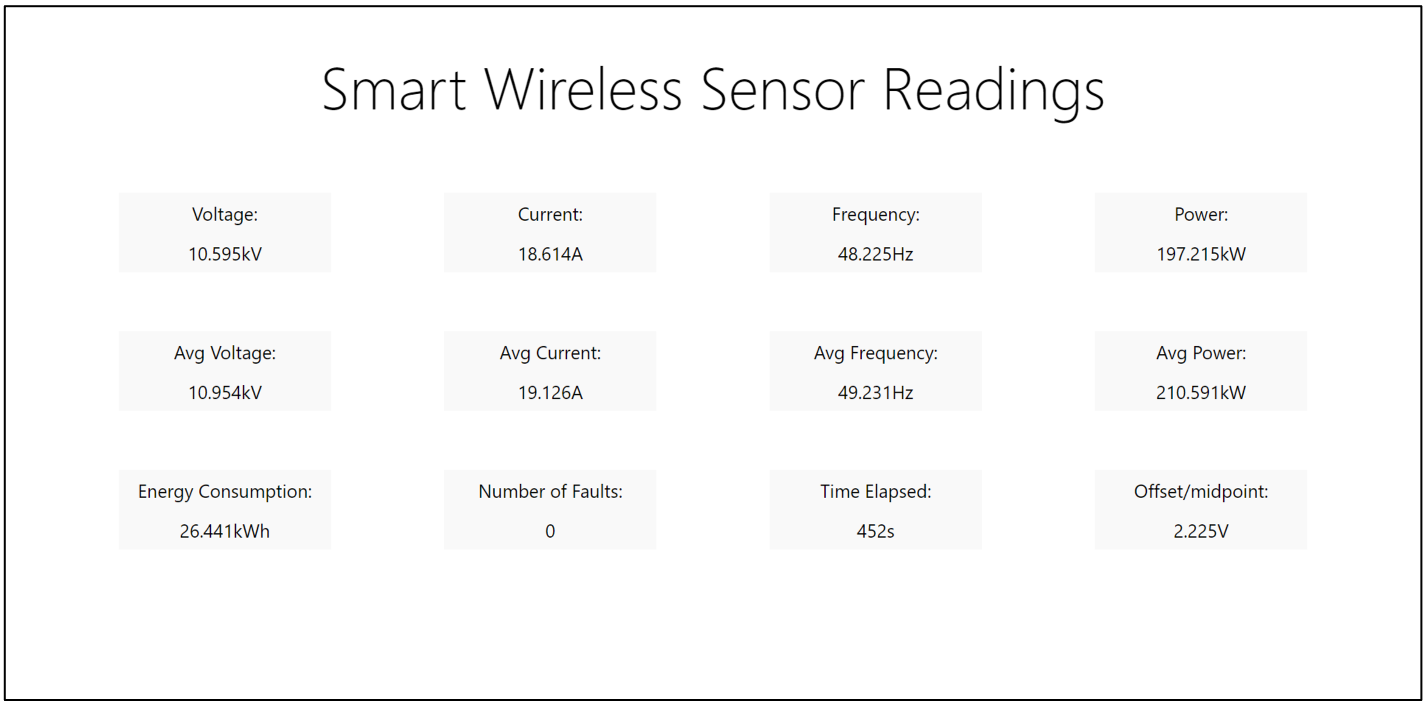
**React JS Website**

The React website sends a GET request to the getValues() endpoint of the Spring Boot application every two seconds. The GET request returns the SensorData object to the React app in JSON format. The React app then extracts the useful live measurements from the SensorData JSON format then updates its local variables, hence causing the values displayed on the website to update, asynchronously – which means that the web page doesn’t have to refresh to do so.

If for some reason the ESP32 is not sending data to the Spring Boot application, then the React App will detect that none of the values have updated for the past 5 seconds and hence it will enter a demo mode. This demo mode is used to demonstrate how the website normally functions when the ESP32 is turned on and is sending data.

Website link: https://smart-wireless-sensor.nw.r.appspot.com/

The full React website code can be seen in Appendix 13. Figure 69, as shown below, showcases a screenshot of the React website in live action.

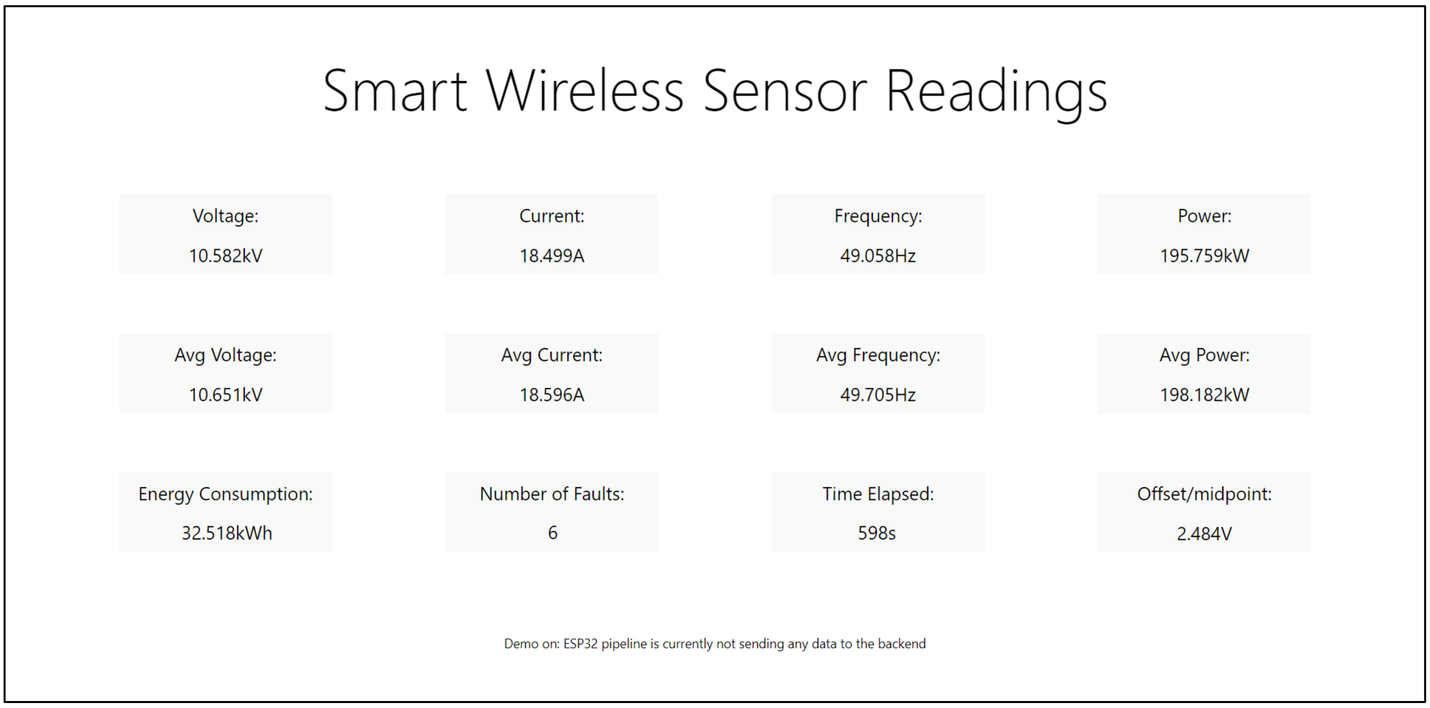


Smart Wireless Sensor Readings

| | | | |
|---|---|---|---|
| Voltage: 10.595kV | Current: 18.614A | Frequency: 48.225Hz | Power: 197.215kW |
| Avg Voltage: 10.954kV | Avg Current: 19.126A | Avg Frequency: 49.231Hz | Avg Power: 210.591kW |
| Energy Consumption: 26.441kWh | Number of Faults: 0 | Time Elapsed: 452s | Offset/midpoint: 2.225V |

*Figure 69 - React website in live action on Google App Engine hosting*

Figure 70, as shown below, showcases the website in demo mode. The following sentence of text can be seen displayed at the bottom of the page to inform the user that the website is currently in demo mode. “Demo on: ESP32 pipeline is currently not sending any data to the backend”



Smart Wireless Sensor Readings

| | | | |
|---|---|---|---|
| Voltage: 10.582kV | Current: 18.499A | Frequency: 49.058Hz | Power: 195.759kW |
| Avg Voltage: 10.651kV | Avg Current: 18.596A | Avg Frequency: 49.705Hz | Avg Power: 198.182kW |
| Energy Consumption: 32.518kWh | Number of Faults: 6 | Time Elapsed: 598s | Offset/midpoint: 2.484V |

Demo on: ESP32 pipeline is currently not sending any data to the backend

*Figure 70 – React website demo mode*

**Google Cloud Platform Bucket Storage**

The history of the 12 useful live measurements are being stored on the local MicroSD card every second. However, in an industry scenario, this MicroSD card may be unfeasible or impossible to access. Therefore, to solve this problem, the history of the useful live measurements should be stored in a suitable cloud storage system. The most suitable solution was found to be Google Cloud Platform Storage (similar to AWS simple storage solution (S3)). GCP Storage uses buckets to store the data – the buckets are basic containers used to store data in the cloud and can store most types of files.

A triggering service was setup so that when the React webpage calls the getValues() endpoint of the Spring Boot API, the 12 useful live measurements returned to the React web page are stored in a structured text file/folder format very similar to the structure that can be seen on the MicroSD card (this microSD card format can be seen in the next section, in Figure 71). Google's bucket storage system is simple, cheap and effective, making it the ideal solution for cloud data storage of the useful live measurements.

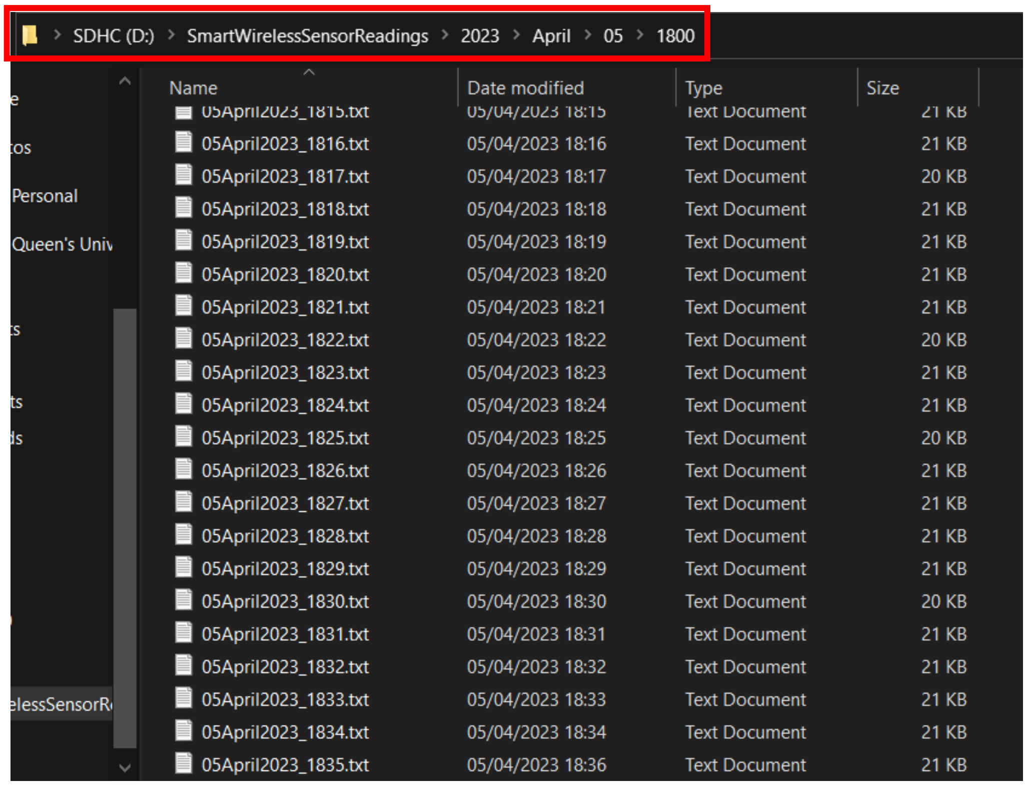
## MicroSD Card Smart Folder Structure and File Names

At this point in the project, the useful live measurements were being stored on the MicroSD card into the Smart_Wireless_Sensor_Readings.txt file. This file had no structure, the new values were just appended to the end of the file. There was no way for a user to see at what time the useful live measurements were stored. A possible solution could have been to print the time above the set of measurements, however, having all the measurements in one text file is very poor practice. This file would be hard and painful to try to navigate through, it would become too large in file size and hence eventually corrupt, it would be limited in terms of size due to the data partitions of the MicroSD card.

To solve these problems, a solution was formulated which involved finding the current date and time, then using this date and time as the file name of the text file. Once the time changes past a certain degree, then a new text file will be created. The next decision to be made was how often a new text file should be created, and the structure of file names and folders. The final decisions made were as follows:

- The useful live measurements should be appended to a separate text file for each new minute
- The text file name should be the full date and time
- The folder structure should be as follows: "Smart_Wireless_Sensor_Readings/year/month/date/time/datemonthyear_time.txt"
- Eg: "Smart_Wireless_Sensor_Readings/2023/April/12/0847/2023April2023_0847.txt"

The folder structure makes it very easy to navigate to the desired year, month, date, and hour. The file name includes the date and time too. Even though the text files are organised in folders based on the date, the text file names themselves include the date too to ensure that if these text files are copied to a location outside of the folder structure – eg. for analysis, or by accident – then it will be easy to determine the origin of those files.

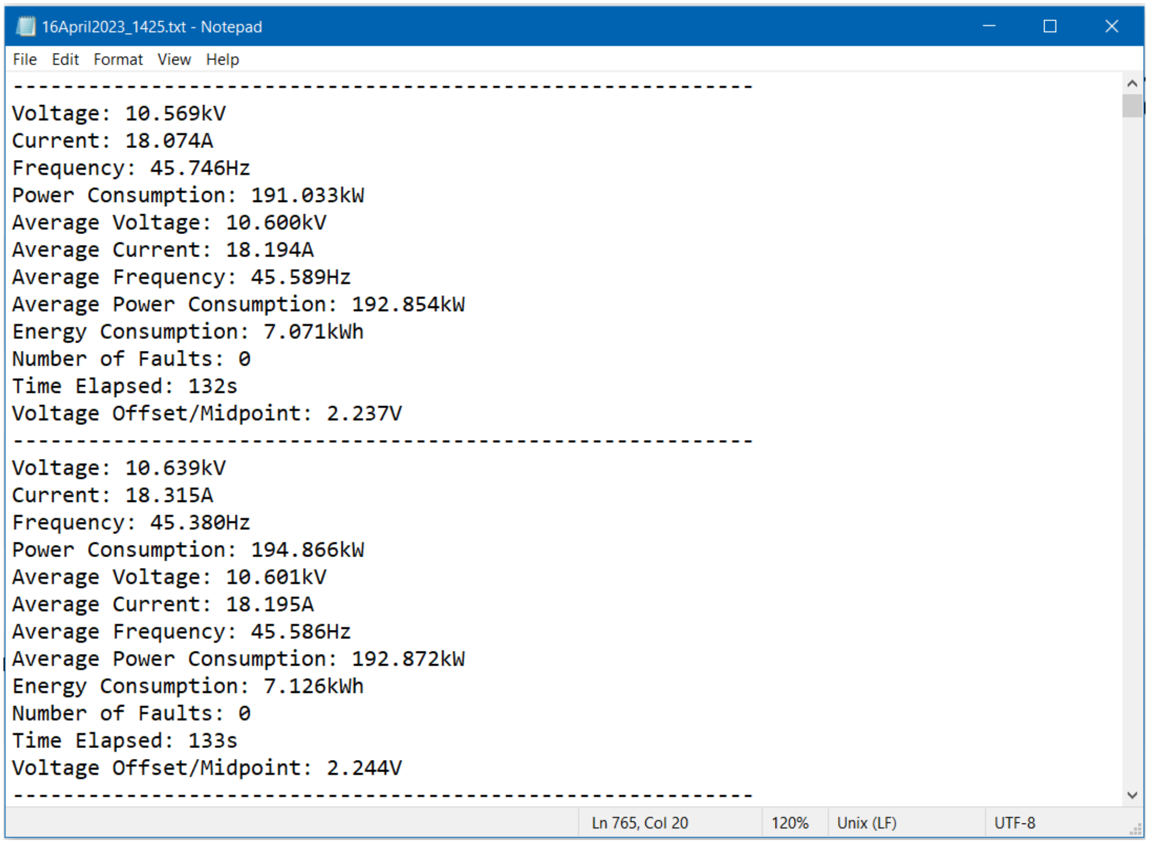


SDHC (D:) > SmartWirelessSensorReadings > 2023 > April > 05 > 1800

| Name | Date modified | Type | Size |
|---|---|---|---|
| 05April2023_1815.txt | 05/04/2023 18:15 | Text Document | 21 KB |
| 05April2023_1816.txt | 05/04/2023 18:16 | Text Document | 21 KB |
| 05April2023_1817.txt | 05/04/2023 18:17 | Text Document | 20 KB |
| 05April2023_1818.txt | 05/04/2023 18:18 | Text Document | 21 KB |
| 05April2023_1819.txt | 05/04/2023 18:19 | Text Document | 21 KB |
| 05April2023_1820.txt | 05/04/2023 18:20 | Text Document | 21 KB |
| 05April2023_1821.txt | 05/04/2023 18:21 | Text Document | 21 KB |
| 05April2023_1822.txt | 05/04/2023 18:22 | Text Document | 20 KB |
| 05April2023_1823.txt | 05/04/2023 18:23 | Text Document | 21 KB |
| 05April2023_1824.txt | 05/04/2023 18:24 | Text Document | 21 KB |
| 05April2023_1825.txt | 05/04/2023 18:25 | Text Document | 20 KB |
| 05April2023_1826.txt | 05/04/2023 18:26 | Text Document | 21 KB |
| 05April2023_1827.txt | 05/04/2023 18:27 | Text Document | 21 KB |
| 05April2023_1828.txt | 05/04/2023 18:28 | Text Document | 21 KB |
| 05April2023_1829.txt | 05/04/2023 18:29 | Text Document | 21 KB |
| 05April2023_1830.txt | 05/04/2023 18:30 | Text Document | 20 KB |
| 05April2023_1831.txt | 05/04/2023 18:31 | Text Document | 21 KB |
| 05April2023_1832.txt | 05/04/2023 18:32 | Text Document | 21 KB |
| 05April2023_1833.txt | 05/04/2023 18:33 | Text Document | 21 KB |
| 05April2023_1834.txt | 05/04/2023 18:34 | Text Document | 21 KB |
| 05April2023_1835.txt | 05/04/2023 18:36 | Text Document | 21 KB |

*Figure 71 - Full folder and file structure of the MicroSD card*

The full folder and file structure of the MicroSD card can be seen in Figure 72, as shown above. It demonstrates the comprehensive structure to both the text files and their overlying folders.



```
------------------------------------------------------------
Voltage: 10.569kV
Current: 18.074A
Frequency: 45.746Hz
Power Consumption: 191.033kW
Average Voltage: 10.600kV
Average Current: 18.194A
Average Frequency: 45.589Hz
Average Power Consumption: 192.854kW
Energy Consumption: 7.071kWh
Number of Faults: 0
Time Elapsed: 132s
Voltage Offset/Midpoint: 2.237V
------------------------------------------------------------
Voltage: 10.639kV
Current: 18.315A
Frequency: 45.380Hz
Power Consumption: 194.866kW
Average Voltage: 10.601kV
Average Current: 18.195A
Average Frequency: 45.586Hz
Average Power Consumption: 192.872kW
Energy Consumption: 7.126kWh
Number of Faults: 0
Time Elapsed: 133s
Voltage Offset/Midpoint: 2.244V
------------------------------------------------------------
```

*Figure 72 - Screenshot of the 12 useful live measurements being stored one of the MicroSD card text files*

Figure 71, as shown above showcases a screenshot of the 12 useful live measurements being stored one of the MicroSD card text files. The reason and choice behind each of these live measurements will be explained in a later section.

An accurate and updated date and time are needed to be able to create the correct microSD card text files and folders. When the ESP32 is reset or loses power, the current date and time are lost. Therefore, on each initialisation/reboot of the ESP32, the current date and time should be acquired and synced. To update the date and time, a valid internet connection must be secured. Therefore, the slave ESP32 will first connect to the internet then try to acquire the updated date and time via an ntp server via the internet. [27]

```
configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
```

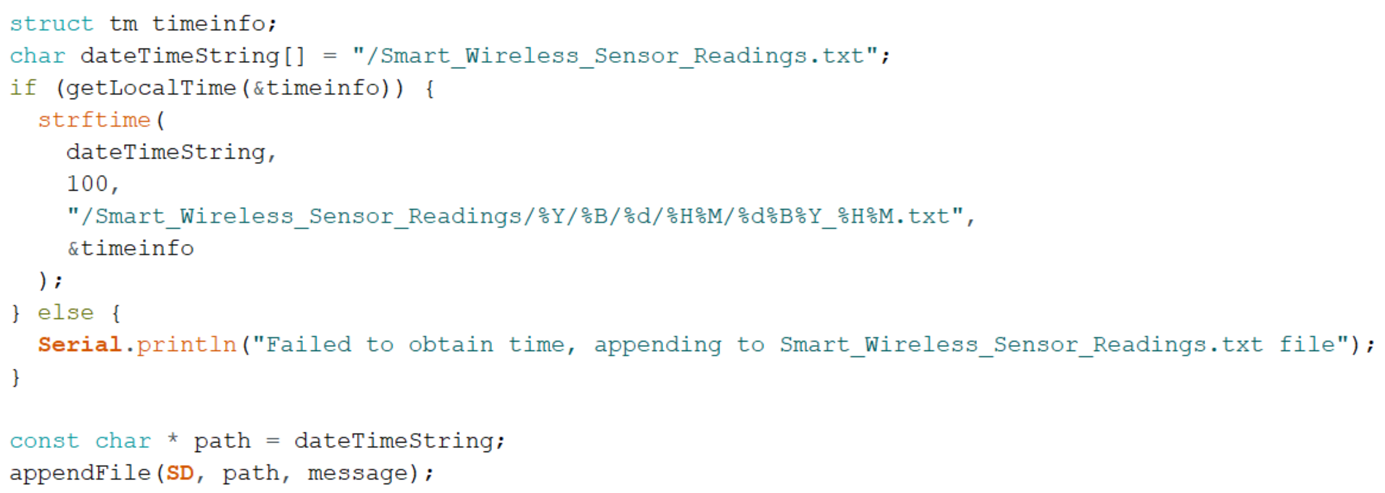
*Figure 73 – ESP32 ntp server setup code*

Figure73 as shown above, showcases the line of code used in the setup section of the ESP32 code to synchronise the ESP32 time with an ntpServer ("pool.ntp.org"). This line of code only needs to be ran in the setup section when initialising the ESP32. Once the ESP32 time has been synchronised here, the ESP32 time stay up to date for the remaining period that the ESP32 is running.

```
const char * format =
  "-----------------------------------------------------------\n"
  "Voltage: %.3fkV\n"
  "Current: %.3fA\n"
  "Frequency: %.3fHz\n"
  "Power Consumption: %.3fkW\n"
  "Average Voltage: %.3fkV\n"
  "Average Current: %.3fA\n"
  "Average Frequency: %.3fHz\n"
  "Average Power Consumption: %.3fkW\n"
  "Energy Consumption: %.3fkWh\n"
  "Number of Faults: %.0f\n"
  "Time Elapsed: %.0fs\n"
  "Voltage Offset/Midpoint: %.3fV\n";
```

*Figure 74 – ESP32 code used to create the skeleton pre-formatted string for the 12 useful live measurements*

Figure 74, as shown above, showcases the ESP32 code used to create the skeleton pre-formatted string for the 12 useful live measurements. The "format" variable is then fed into an snprintf() function, where the 12 measurements are passed in place of the skeleton placeholders defined by the percentage symbols. This skeleton code format can be seen in multiple other places in the code, including the HTTP response body return string of the station and AP web server.



```
struct tm timeinfo;
char dateTimeString[] = "/Smart_Wireless_Sensor_Readings.txt";
if (getLocalTime(&timeinfo)) {
  strftime(
    dateTimeString,
    100,
    "/Smart_Wireless_Sensor_Readings/%Y/%B/%d/%H%M/%d%B%Y_%H%M.txt",
    &timeinfo
  );
} else {
  Serial.println("Failed to obtain time, appending to Smart_Wireless_Sensor_Readings.txt file");
}

const char * path = dateTimeString;
appendFile(SD, path, message);
```

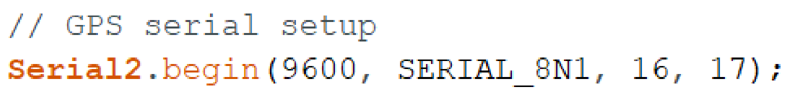
*Figure 75 - Code used for the time and microSD card functionality in the main loop of the slave ESP32*

Figure 75, as shown above, showcases the code used for the time and microSD card functionality in the main loop of the slave ESP32. The "getLocalTime(&timeinfo)" line calls the getLocalTime() function and passes a reference to the memory address of the timeinfo variable as a parameter. This means that the getLocalTime function will be able to directly access and make changes to the timeinfo variable without having to return a separate variable. The timeinfo variable is assigned a string value containing the updated date and time. This string is then formatted to the specific folder and file name format. The string is then used as the path when appending to the text file. The "appendFile()" microSD card function will even create a new text file if the specified file and folder path does not already exist.

## GPS breakout for ADC timing disciplining - Solution

As shown in the previous GPS section, the 1PPS signal coming from the GPS breakout was passed into the ESP32 interrupt pin and caused the ESP32 to take ADC readings accurately once every second. The next task was to use this 1PPS signal to discipline the interrupt timer of the ESP32, i.e. no matter the geo-location or the number of smart wireless sensors, the ADC readings in every sensor should be taken at exactly the same time – perfectly in sync. This is crucial for the smart wireless sensors, to ensure that the readings and useful live measurements of the sensors are taken at exactly the same time.

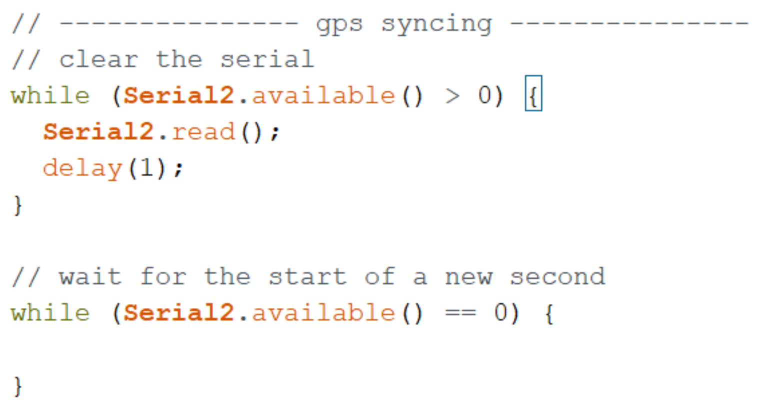
Since the ADC readings will be taken at a very high sampling rate of 1200sps, it would be unwise to try to sync the ESP32 every second. An optimal solution was formed of disciplining the timing interrupts initially then relying on the ESP32 clock, from then on, to maintain the 1200sps readings and hence, maintain the in-sync interrupt ADC reading timings. When the master ESP32 is initialised or rebooted, it will wait in a loop until it receives the 1PPS signal from the GPS breakout via UART. Once the signal is received, the ESP32 interrupt timer is kicked-off and the ESP32 begins taking readings from the ADC, following the timings of the ESP32 interrupt timer (at 1200sps).



```
// GPS serial setup
Serial2.begin(9600, SERIAL_8N1, 16, 17);
```

*Figure 76 - The line of code that initialises the GPS breakout*

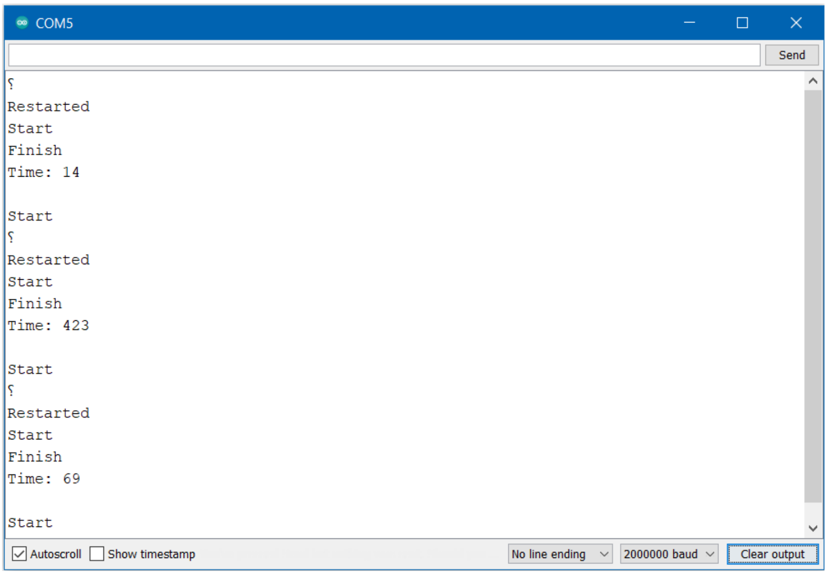
Figure 76, as shown above, showcases the line that initialises the GPS breakout. This line is ran in the setup section of ESP32 master program.



```
// --------------- gps syncing ---------------
// clear the serial
while (Serial2.available() > 0) {
  Serial2.read();
  delay(1);
}

// wait for the start of a new second
while (Serial2.available() == 0) {

}
```

*Figure 77 - GPS interrupt syncing code*

Figure 77, as shown above, showcases the GPS interrupt syncing code that runs at the start of the main ESP32 master code loop, after the ESP32 setup code has already ran. The first while loop clears the serial in preparation for the second while loop. Once the serial has been cleared by the first while loop, the second while loop waits for the first byte of the NMEA sequence sent from the GPS breakout to the ESP32. The first byte will be transferred at the exact start of a new second and hence, the while loop will break at this exact moment then the ADC readings will begin to be taken – following the 1200sps interrupt timings of the ESP32 clock. This interrupt disciplining ensures that the master ESP32 always starts off its timings at the exactly the start of a new second, and thanks to the ESP32 clock precision, the timing interrupts will stay in sync from this point forward.



```
ʕ
Restarted
Start
Finish
Time: 14

Start
ʕ
Restarted
Start
Finish
Time: 423

Start
ʕ
Restarted
Start
Finish
Time: 69

Start
```

*Figure 78 - Serial Monitor output of the GPS interrupt sync testing*

The GPS interrupt timing was thoroughly tested to ensure that the functionality worked as expected. Figure 78, as shown above, showcases the Serial Monitor output of the GPS interrupt sync testing. The millis() function was utilised to test how long the second while loop (shown in Figure 77) would have to wait for the exact start of a new second. This could be anywhere between 1 – 999ms. Figure 78 shows waiting times of 14ms, 423ms and 69ms, proving that the GPS interrupt timer disciplining functionality works as expected. The GPS breakout as part of the circuit can be seen later, in Figure 87.

## Signal Conditioning Circuit Problem Fix

In semester 1, there was an issue with the signal conditioning circuit. To shift the output voltage signal by 2.5V, a 2.5V signal was passed into the non-inverting pin of the op-amp.

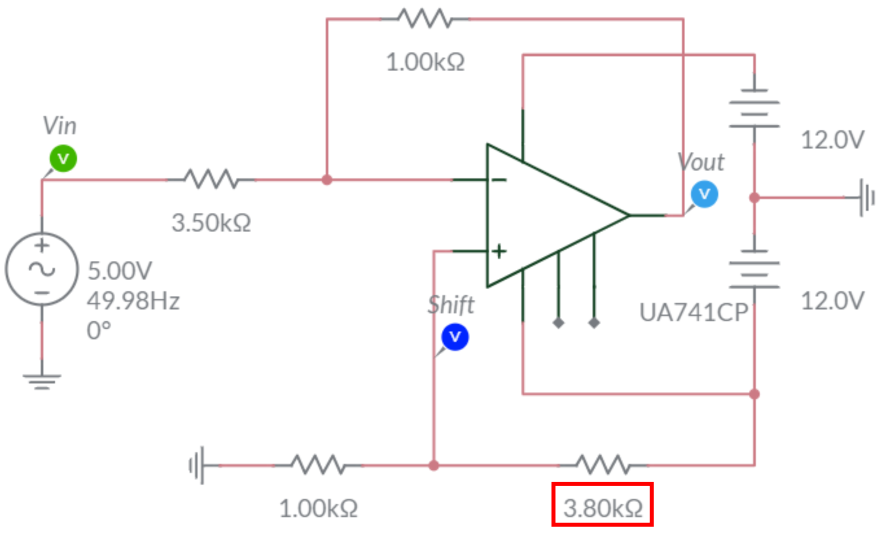
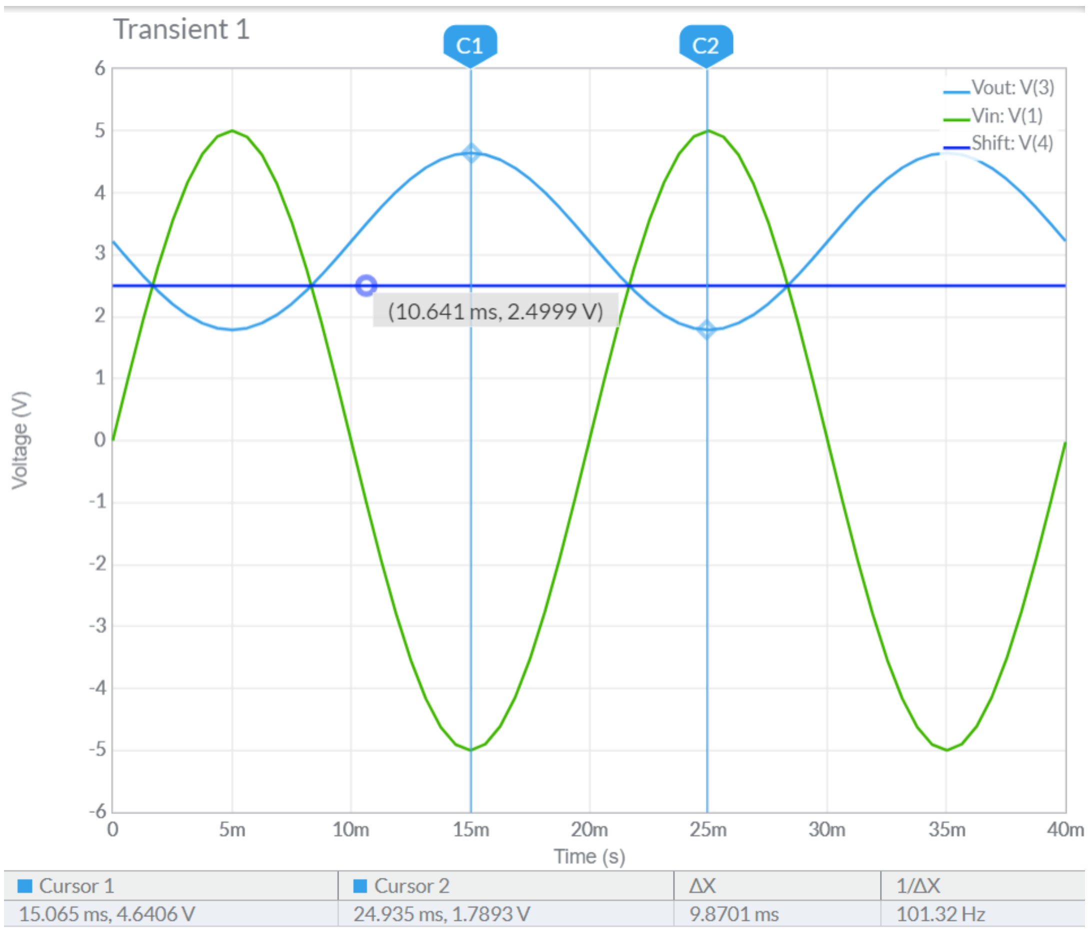


*Figure 79 - Semester 1 op-amp circuit*

Figure 79, as shown above, showcases the op-amp circuit in semester 1 that was having the circuit issues. The problem lay with the value of the voltage divider 3.80kΩ resistor value as highlighted in red.



| 🟦 Cursor 1 | 🟦 Cursor 2 | ΔX | 1/ΔX |
|---|---|---|---|
| 15.065 ms, 4.6406 V | 24.935 ms, 1.7893 V | 9.8701 ms | 101.32 Hz |

*Figure 80 - The transient output waveforms produced by the Figure 79 circuit*

Figure 80, as shown above, showcases the transient output produced by the Figure 79 circuit. The dark blue "Shift" line shows the 2.5V non-inverting input voltage being fed into the op-amp, however, the Vout signal has been shifted much further than 2.5V, as shown in the transient wave. Using the C1 and C2 wave cursors, the midpoint of the wave can clearly be found as (4.6406 + 1.7893) / 2 = 3.215V when it should actually be equal to 2.5V.

$$Vout = Vin \; \frac{R2}{R1 + R2}$$

$$Vout\;(R1 + R2) \;=\; Vin \times \; R2$$

$$R1 \;=\; \frac{Vin \times R2}{Vout} - R2$$

$$R1 \;=\; \frac{12 \times 1}{2.5} - 1$$

$$R1 \;=\; 3.80k\Omega$$

*Figure 81 - Voltage divider equation*

The original voltage divider calculation used to obtain this 3.80kΩ resistor value can be shown above, in Figure 81. The problem with this voltage divider calculation is that it does not account for the gain of op-amp itself.

$$Vout = Gain \times Vin \times \frac{R2}{R1 + R2}$$

$$Vout\ (R1 + R2) = Gain \times Vin \times R2$$

$$R1 = \frac{Gain \times Vin \times R2}{Vout} - R2$$

$$R1 = \frac{\left(1 + \frac{1}{3.5}\right) \times 12 \times 1}{2.5} - 1$$

$$R1 = 5.17k\Omega \approx 5.20k\Omega)$$

*Figure 82 - Fixed voltage divider equation*

Figure 82, as shown above, showcases the fixed voltage divider equation – with the gain included in the equation. As shown in the equation, the correct value of R1 in the voltage divider circuit was found to be 5.17kΩ, which can be rounded up to 5.20kΩ for the physical circuit later.

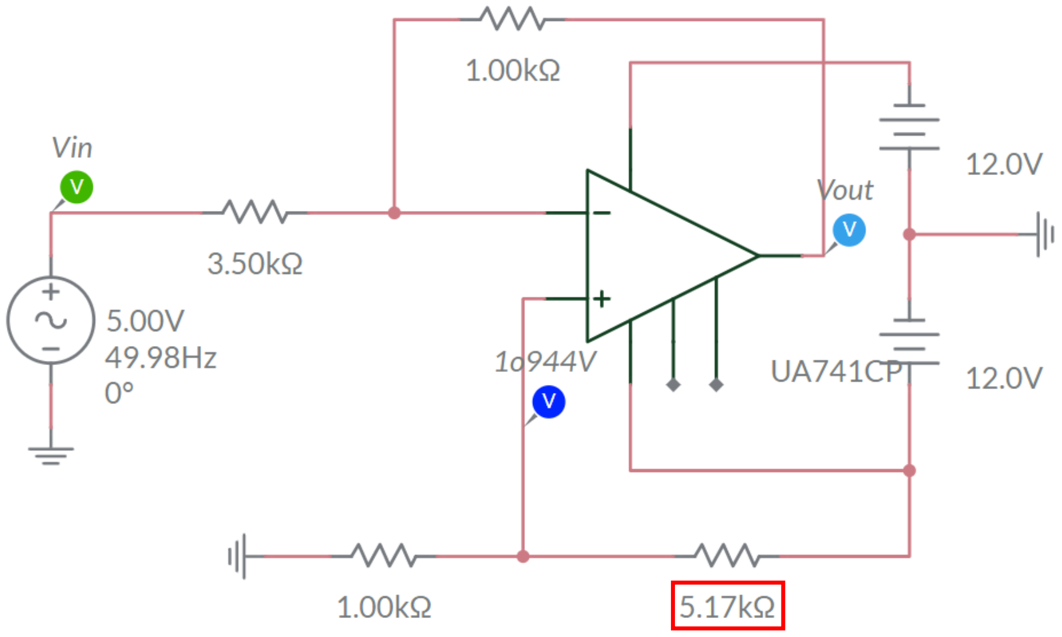
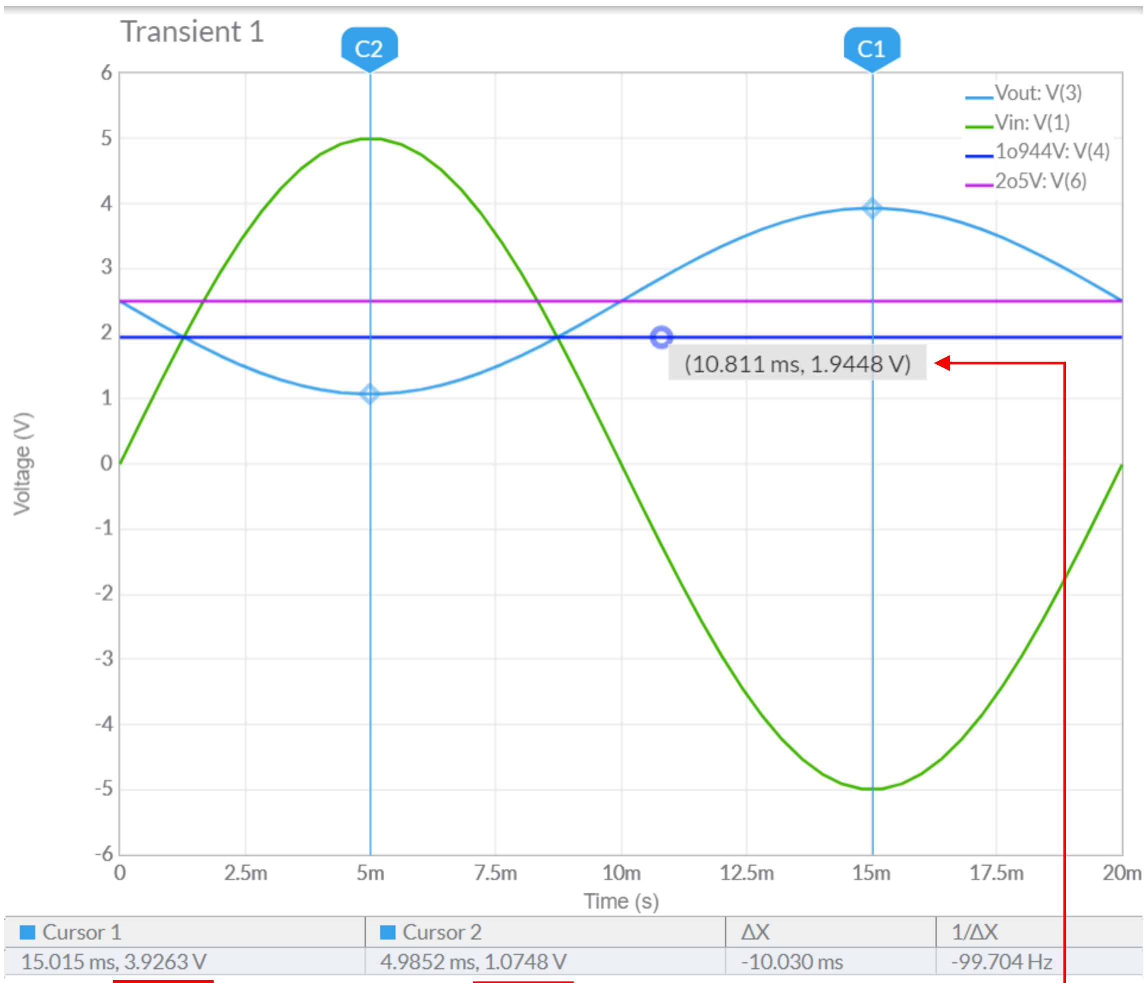


*Figure 83 - Newly-fixed inverting op-amp circuit with the correct voltage divider circuit containing the 5.17kΩ R1 resistor*

Figure 83, as shown above, showcases the newly-fixed inverting op-amp circuit with the correct voltage divider circuit containing the 5.17kΩ R1 resistor.

Figure 84, as shown below, showcases the transient waveform output of the fixed op-amp circuit shown in Figure 83. It's clear from using the C1 and C2 wave cursors in Figure 84, that the midpoint of the Vout waveform is equal to (3.9263 + 1.0748) / 2 = 2.501V which is almost exactly 2.5V, as expected.



| 🟦 Cursor 1 | 🟦 Cursor 2 | ΔX | 1/ΔX |
|---|---|---|---|
| 15.015 ms, 3.9263 V | 4.9852 ms, 1.0748 V | -10.030 ms | -99.704 Hz |

*Figure 84 - The transient waveform output of the fixed op-amp circuit shown in Figure 83*

$$Non\ inverting\ op\ amp\ input = \frac{Desired\ Offset}{Gain}$$

$$Non\ inverting\ op\ amp\ input = \frac{2.5}{1 + \frac{1}{3.5}}$$

$$Non\ inverting\ op\ amp\ input = \underline{1.944V}$$

*Figure 85 - Non inverting op-amp input equation*

Figure 85, as shown above, showcases a different equation that is used to validate the voltage divider voltage shift section of the op-amp circuit. The equation shows that the non-inverting op-amp input pin requires an input of roughly 1.944V to obtain the Vout voltage shift of 2.5V. The purple “1o944V” line on shown in Figure 84 (and its source in Figure 85) shows a DC value of 1.9448V being fed into the non-inverting pin of the op-amp, which is very close to the calculated value of 1.944V, hence validating that the voltage divider shift circuit is splitting the voltage by the correct amount and hence, is functioning as expected.

## Signal Conditioning Physical Circuit

After fixing the signal conditioning circuit with the correct resistor values, as shown in the previous section, it was time to build the physical circuit. The following parts were acquired for the circuit:

- 5.2kΩ resistor (5kΩ + 200Ω) (rounded up from 5.17kΩ)
- 3.5kΩ resistor (3.3kΩ + 200Ω)
- Two 1kΩ resistors
- Function Generator
- UA741 Op Amp
- 12V DC Power supply for the 741 op-amp
- Breadboard jumper wires
- Test leads

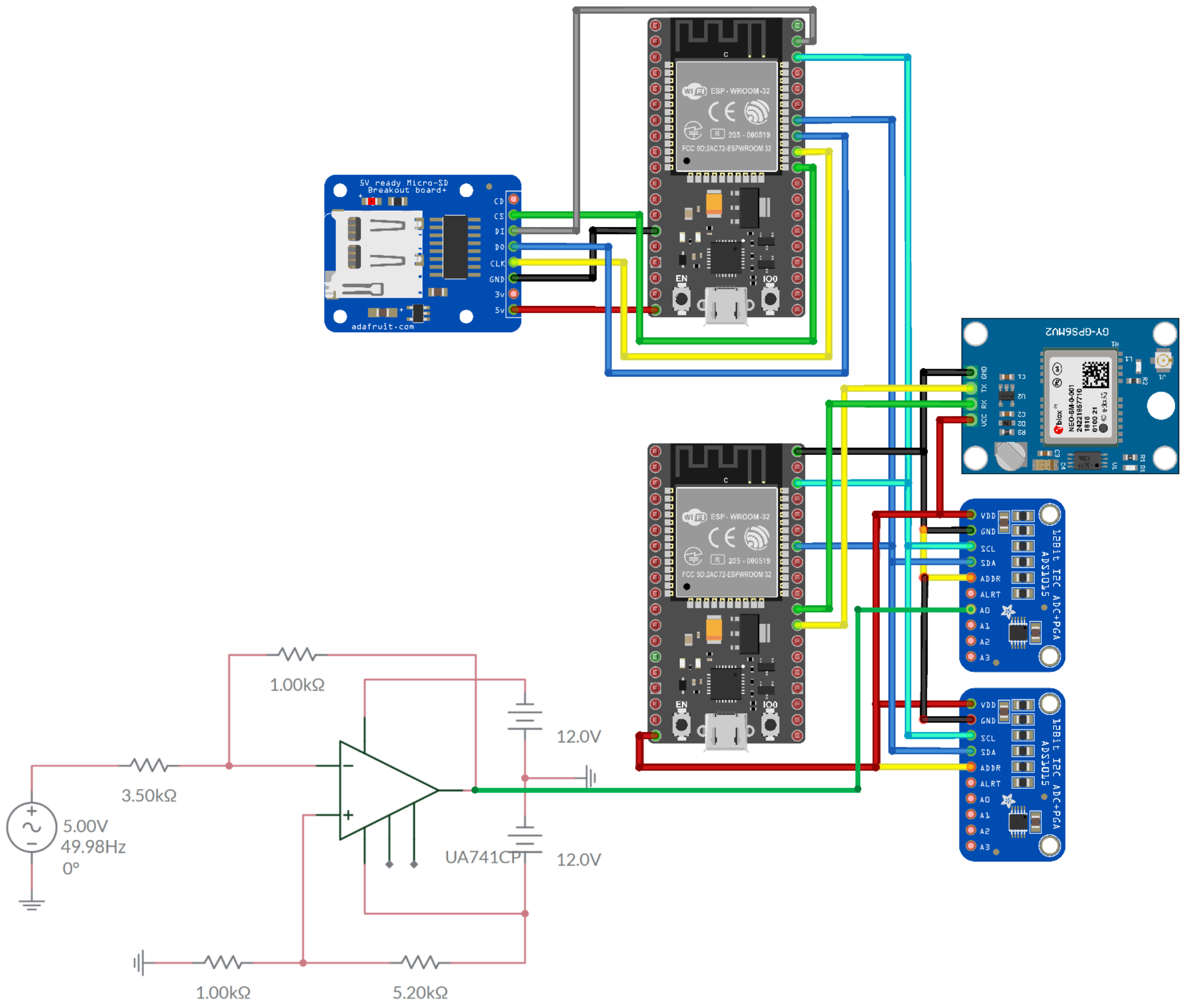


*Figure 86 - Circuit diagram of the op-amp circuit wired up to the ESP32 circuit*

Figure 86, as shown above, showcases the circuit diagram of the op-amp circuit wired up to the ESP32 circuit. The output of the op-amp circuit is connected to the A0 analogue input pin of the first ADS1015 ADC. The function generator amplitude was set at 5V and the frequency was set at 49.98Hz, to simulate a mains signal frequency – which will be extremely close to 50Hz. A photo of the physical circuit can be seen below, in Figure 87.

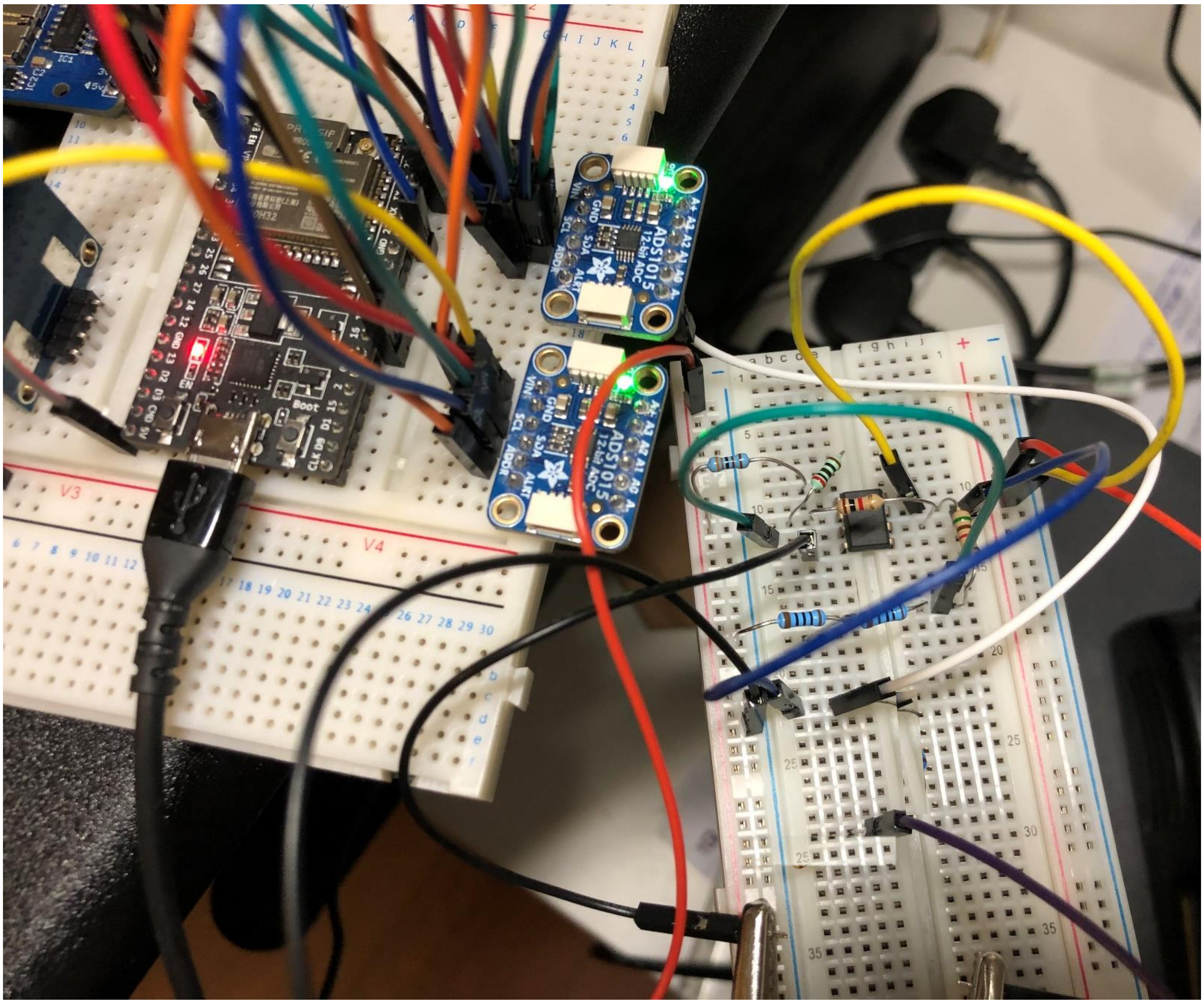


*Figure 87 – Physical op-amp circuit hooked up to ADC 1, as shown in circuit diagram in Figure 86*

*Figure 88 - Rapid Electronics 12V DC power supply that was used for this circuit*

Figure 88, as shown above, showcases the Rapid Electronics 12V DC power supply that was used for this circuit. This was later replaced by a 8 x 1.5V AA battery holder.

The function generator feeds a non-shifted 5V analogue input signal to the signal conditioning circuit, which inverts the signal and attenuates it down to the 1VRMS Vout. This 1VRMS output signal was then fed into an analogue input of the ADS1015.

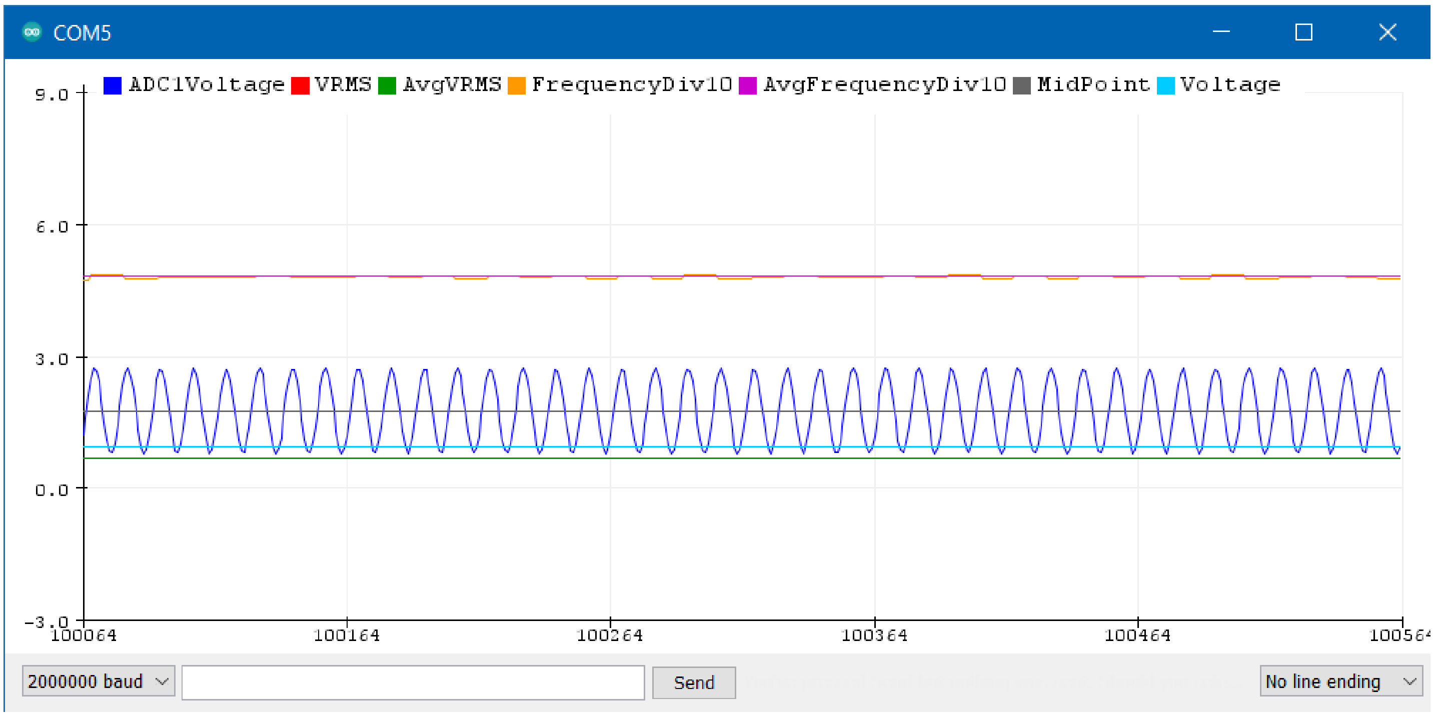

COM5
🟦 ADC1Voltage 🟥 VRMS 🟩 AvgVRMS 🟧 FrequencyDiv10 🟪 AvgFrequencyDiv10 ⬛ MidPoint 🟦 Voltage
9.0
6.0
3.0
0.0
-3.0
100064
100164
100264
100364
100464
100564
2000000 baud
Send
No line ending

*Figure 89 - Serial Plotter output waveform of the resultant signal being fed into the ADC from the inverting op-amp circuit*

Figure 89, as shown above, showcases the Serial Plotter output waveform of the resultant signal being fed into the ADC from the inverting op-amp circuit. The waveform was very clean after coming through the inverting op-amp circuit, this is thanks to the high impedance of the op-amp itself. This signal was used to represent the voltage coming from VT in the three phase system.

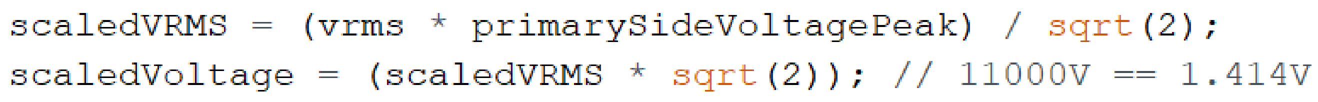
## Voltage Scaling

The smart wireless sensor measures the signal conditioning attenuation circuit Vout signal. The Vin value of the circuit was set to 5V, however, the secondary side voltage of the VT transformer is actually 110V. This 110V would be unsafe and unfeasible to work with in practice, and hence, 5V was used as Vin, for demonstration purposes.

Since this project is not working with the true VT values, this project is therefore used to simulate these values. The primary side voltage of the VT transformer is the voltage value that the smart wireless sensor is actually trying to measure, therefore, this primary side voltage must be calculated through scaling the simulated value. A suitable primary side voltage would be around 11kV.

The full V-peak value of the primary side is set at 11kV. Therefore, when the secondary side voltage is at the V-peak value of 110V, the voltage value in the program should be scaled to this full V-peak value. In the signal conditioning circuit, the 110V is scaled down to 1VRMS (1.414V) and hence, whenever the voltage being inputted into the ADC is of 1.414V, then the scaled voltage should be calculated to be 11kV for the primary side and 110V for the secondary side.

This is a simple ratio – if the value being fed into the ADC is was halved, i.e. 1.414 / 2 = 0.707V, then the scaled primary side voltage will be halved down to 5.5kV and the scaled secondary side voltage will then be halved down to 55V.



```
scaledVRMS = (vrms * primarySideVoltagePeak) / sqrt(2);
scaledVoltage = (scaledVRMS * sqrt(2)); // 11000V == 1.414V
```

*Figure 90 - Code used to scale the voltage value to the 11kV primary side simulated value*

Figure 90, as shown above, showcases the two lines of code used to scale the voltage value to the 11kV primary side simulated value. To find the voltage of the analogue input signal, the VRMS has to be worked out first, as explained in the VRMS section. The primarySideVoltagePeak float variable was set to 11000.0, earlier in the code. The VRMS value is scaled by calculating the ratio of the measured voltage value compared to the full scale ADC voltage value of sqrt(2) (= 1.414) which is then multiplied by 11000. In summary, an input signal of voltage 1.414V will equate to the scaledVoltage variable value (shown on the second line) being set to 11kV, as expected.

## Current

The current can't actually be measured, so it is instead being simulated. A simulation was made that represents a full fault current generator for the three-phase current transformer (CT). The peak fault current of the simulation is set to a value of 200A, however, most of the time the actual load current will be around 10% of that, i.e. at roughly 20A. The current will only reach 200A if there is short circuit fault in the circuit.

In this simulation circuit, the current is represented by the voltage output coming from the function generator. Since the load current is around 10% of the full 200A peak fault current, this must be accounted for when using the function generator to simulate the current signal (with voltage). There are two possible options to obtain this signal of 10% amplitude:

1. Use two function generators – one for the VT voltage and one for the CT voltage
2. Use one function generator in combination voltage divider circuit to split the signal between the voltage op-amp circuit and the "current" op-amp circuit. The current simulation signal will be dropped to a voltage of roughly 10% of the voltage simulation signal

The first option is wasteful as it requires two separate function generators and this set up will also produce two waveforms that are randomly out of phase – which is unrealistic as the VT and CT signals will be in phase in the real-world scenario. Therefore, the optimal choice is to use one function generator in combination with a simple voltage divider circuit.

Figure 91, as shown below, showcases the simple voltage divider equation used to determine what size of resistors were needed for the voltage divider circuit to split the function generator signal between the voltage and current simulation signal conditioning circuits.

$$Vout = Vin \frac{R2}{R1 + R2}$$

$$Vout\ (R1 + R2) = Vin \times R2$$

$$R1 = \frac{Vin \times R2}{Vout} - R2$$

$$R1 = \frac{5 \times 1}{0.5} - 1$$

$$R1 = 9k\Omega$$

*Figure 91 - Voltage divider equation to determine function generator resistor value for split*

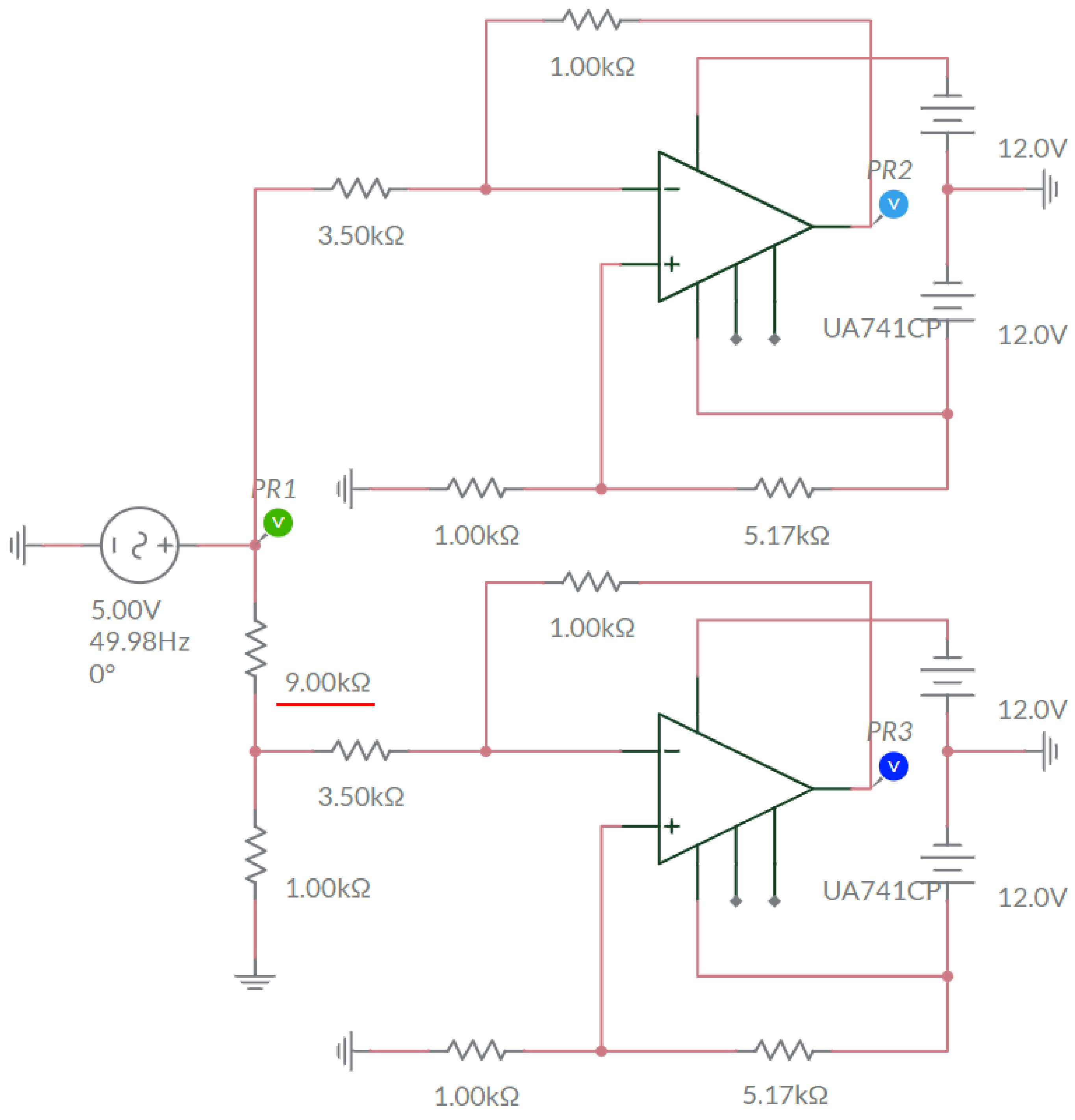
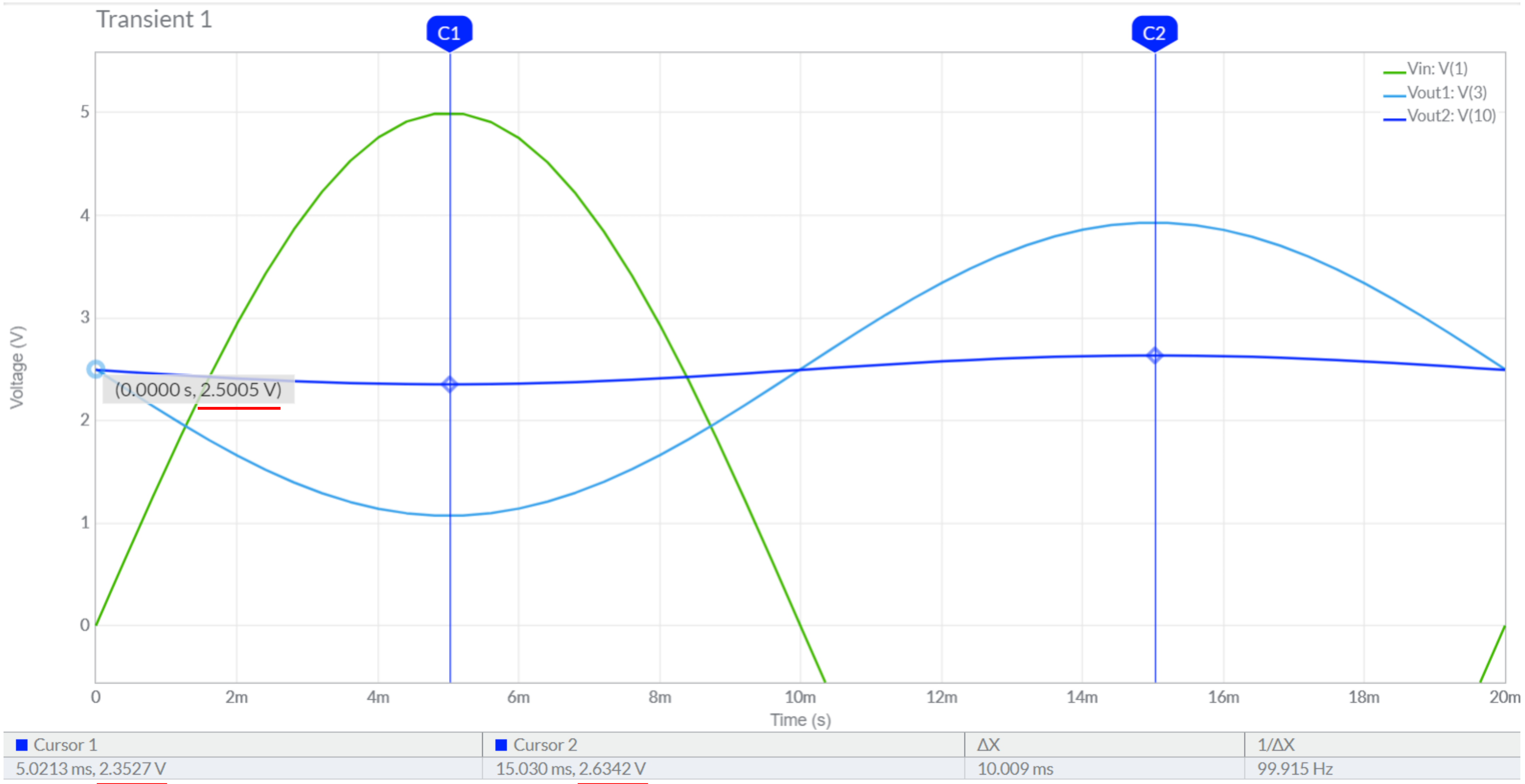


*Figure 92 - Multisim circuit schematic diagram for the voltage and current op-amp attenuation circuits being fed the 5V function generator signal*

Figure 92, as shown above, showcases the Multisim circuit schematic diagram for the voltage and current op-amp attenuation circuits being fed the 5V function generator signal split by the voltage divider.



| ■ Cursor 1 | ■ Cursor 2 | ΔX | 1/ΔX |
|---|---|---|---|
| 5.0213 ms, 2.3527 V | 15.030 ms, 2.6342 V | 10.009 ms | 99.915 Hz |

*Figure 93 - Transient waveform outputted by the circuit shown in Figure 92*

Figure 93, as shown above, showcases the transient waveform outputted by the circuit shown in Figure 92. The light blue waveform labelled “Vout” showcases the voltage output waveform and the dark blue waveform labelled “Vout2” showcases the simulated current output waveform. The dark blue current (“Vout2”) waveform can be seen to have a Vp-p value of (2.6342 - 2.3527) / 2 = 0.141Vp, which is exactly 10% of the PR2 voltage waveform (Vp = 1.414V), which is as expected. By using the current waveform cursor values again, the midpoint can also be proven to be (2.6342 + 2.3527) / 2 = 2.493V which is almost exactly 2.5V, as expected.

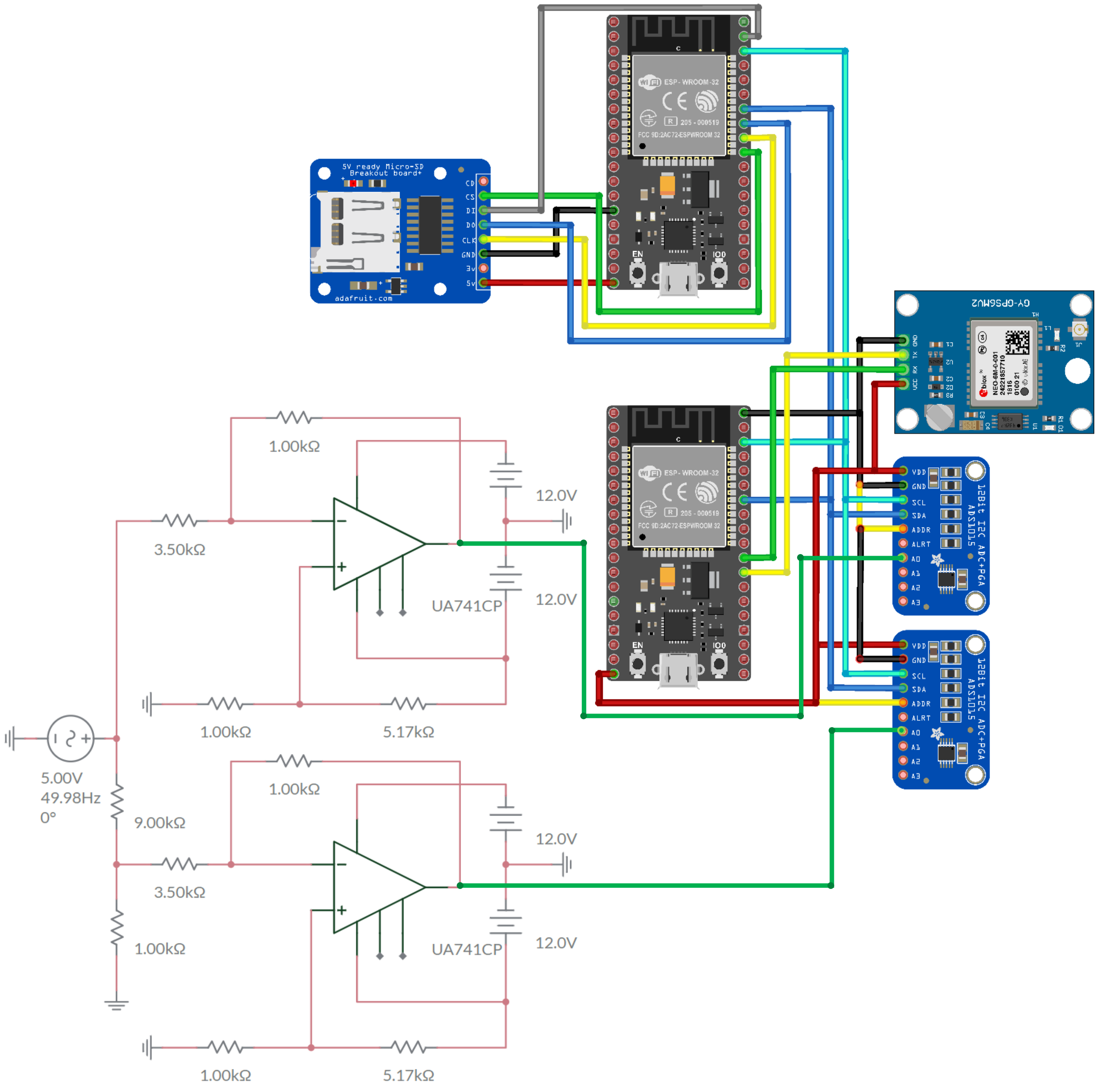


*Figure 94 - Fritzing circuit to represent the final smart wireless sensor circuit, which includes the full double op-amp attenuation circuit*

Figure 92,  as shown above, showcases the Fritzing circuit to represent the final smart wireless sensor circuit, which includes the full voltage and simulated current function generator double op-amp attenuation circuit connected to the ESP32 master slave circuit. The second physical inverting op-amp circuit was then put together on a breadboard, exactly the same as the voltage signal conditioning circuit. The physical voltage divider circuit was added to the function generator input to split the signal to between the voltage and simulated current op-amp circuits.

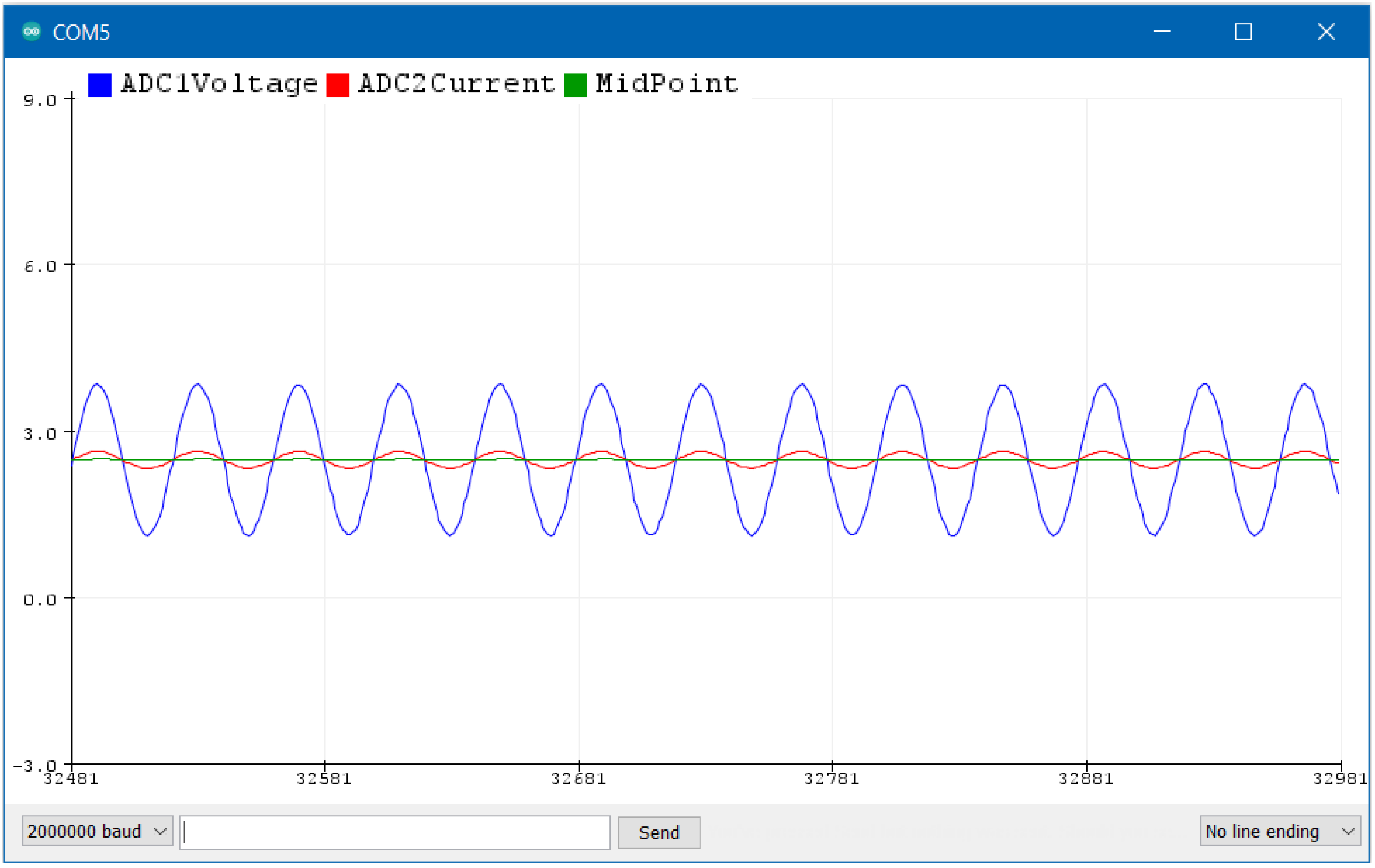
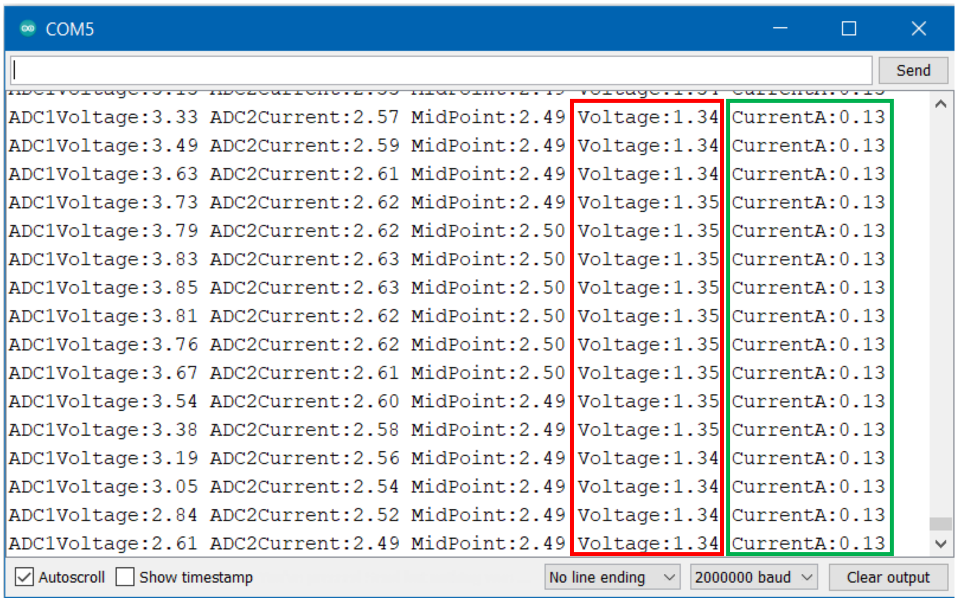


*Figure 95 - Serial Plotter output taken for the 1200sps ADC readings for the outputs from the voltage op-amp circuit and the simulated current op-amp circuit*

Figure 95, as shown above, showcases the simple Serial Plotter output taken for the 1200sps ADC readings for the outputs from the voltage op-amp circuit and the simulated current op-amp circuit. It is clear that both waveforms are very clean in nature and the simulated current voltage signal is small fraction of the first voltage signal. It is not clear from Figure 95 if the current value is 10% of the voltage value, however, this will be proven below.

Once the simulated current signal was being read from the ADC successfully, it was time to actually calculate the “current” value from this second voltage signal. The solution was exactly the same as the first signal – to work out the IRMS using the sliding window technique then the current value is just sqrt(2) multiplied by the IRMS value.

Now that the “simulated current” value was successfully being calculated, the value being roughly 10% of the voltage value is further proven in the Serial Monitor screenshot in Figure 96, as shown below. The calculated voltage and current signals were included in this screenshot to demonstrate the ratio, as highlighted by the red and green rectangles. The frequency of the current waveform did not need to be calculated in the code as the frequency of this waveform is exactly the same as the voltage waveform, due to them coming from the same function generator signal.



```
ADC1Voltage:3.33 ADC2Current:2.57 MidPoint:2.49 Voltage:1.34 CurrentA:0.13
ADC1Voltage:3.49 ADC2Current:2.59 MidPoint:2.49 Voltage:1.34 CurrentA:0.13
ADC1Voltage:3.63 ADC2Current:2.61 MidPoint:2.49 Voltage:1.34 CurrentA:0.13
ADC1Voltage:3.73 ADC2Current:2.62 MidPoint:2.49 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.79 ADC2Current:2.62 MidPoint:2.50 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.83 ADC2Current:2.63 MidPoint:2.50 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.85 ADC2Current:2.63 MidPoint:2.50 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.81 ADC2Current:2.62 MidPoint:2.50 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.76 ADC2Current:2.62 MidPoint:2.50 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.67 ADC2Current:2.61 MidPoint:2.50 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.54 ADC2Current:2.60 MidPoint:2.49 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.38 ADC2Current:2.58 MidPoint:2.49 Voltage:1.35 CurrentA:0.13
ADC1Voltage:3.19 ADC2Current:2.56 MidPoint:2.49 Voltage:1.34 CurrentA:0.13
ADC1Voltage:3.05 ADC2Current:2.54 MidPoint:2.49 Voltage:1.34 CurrentA:0.13
ADC1Voltage:2.84 ADC2Current:2.52 MidPoint:2.49 Voltage:1.34 CurrentA:0.13
ADC1Voltage:2.61 ADC2Current:2.49 MidPoint:2.49 Voltage:1.34 CurrentA:0.13
```

*Figure 96 – Serial monitor screenshot of voltage and current*

## Current Scaling

Now that the simulated current signal readings were coming through successfully and the current was being successfully calculated, it was time to scale the current. This was done in exactly the same manner as the voltage was scaled.

The full peak fault current value of the primary side of the CT transformer was set at 200A, previously. Therefore, a calculated simulated current voltage of 1.414V being calculated from the current op-amp circuit readings would equate to a primary side current of 200A. As explained previously, the load current of the circuit normally sits at around 10% of the full peak fault current value and hence the voltage divider circuit from the function generator signal was used to drop the voltage of the signal being fed into the simulated current op-amp circuit to 10%.

It was then proven, in Figure 96, that the “simulated current” voltage coming through was around (0.13V) 10% of the voltage value, as expected. Therefore, the primary side current shown in Figure 96 can be worked out using the following equation:

$$CT\ primary\ side\ current = Peak\ Fault\ Current\ \times \frac{Simulated\ Current\ Signal\ Voltage}{Full\ Scale\ Current\ Value}$$

$$CT\ primary\ side\ current = \ 200\ \times \frac{0.13}{\sqrt{2}}$$

$$CT\ primary\ side\ current = \ 18.385A$$

*Figure 97 – Scaling equation*

The scaling equation, shown in Figure 97, was then implemented into the code to scale the current values to their full CT primary side current values.

## Power and Energy Consumption

Now that the scaled voltage and scaled current values were being calculated from the two op-amp circuit outputs, the power and energy consumption values could finally be calculated by implementing two algorithms in the code. The power equation can be seen below, in Figure 98.

$$Power = Current \times Voltage$$

*Figure 98 – Power equation*

The energy consumption could then be calculated by using the energy equation in Figure 99, as shown below.

$$Energy(kWh) = Power(kW) \times Time(h)$$

*Figure 99 - Energy equation*

The C++ code implementation of the power and energy consumption equations can be shown below, in Figure 100 and Figure 101, respectively. The scaledPower and avgScaledPower variables are later converted to kW by the slave ESP32 whenever it parses the I2C message from the master.

```
scaledPower = scaledVoltage * scaledCurrent;   // W
avgScaledPower = ((avgScaledPower * i) + scaledPower) / (i + 1);   // W
```

*Figure 100 - Power C++ code*

```
timeElapsed = (millis() - startTime) / 1000;   // s
energyConsumption = avgScaledPower * timeElapsed / 3600000; // kWh
```

*Figure 101 - Energy C++ code*

The power and energy consumption were the final metrics that were calculated as part of the smart wireless sensor project. The full list of 18 metrics being measured/calculated can be shown in the list, in Figure 102, below. Both the scaled and non-scaled values can be found in the master ESP32 program, in Appendix 9.

- ADC1 voltage readings
- ADC2 "current" readings
- Scaled VRMS
- Scaled IRMS
- Average scaled VRMS
- Average scaled IRMS
- Scaled voltage
- Scaled current
- Average scaled voltage
- Average scaled current
- Frequency
- Average frequency
- Scaled power
- Average scaled power
- Energy Consumption
- Number of Faults
- Time elapsed
- Midpoint

*Figure 102 - 18 metrics calculated by the master ESP32*

Out of the 18 above metrics listed in Figure 102, the 12 underlined metrics are the metrics that are being sent to the slave ESP32, and hence, being sent to the smart wireless sensor cloud-hosted website and cloud data storage. Screenshots of the final website and website demo can be seen in the External Communications section, in Figure 69 and Figure 70, respectively.

## Battery-powered Solution

A portable power bank was utilised to provide power to both the ESP32s. This allows the double ESP32 circuit to be used in any scenario or location, regardless of whether there's a mains supply nearby. A photo of the power bank connected to the ESP32 circuit can be seen below, in Figure 103. The "Anker" portable power bank seen in this photo was chosen for its small cylindrical form factor and decent battery capacity. Multiple power banks could be used in future to improve the power bank longevity.

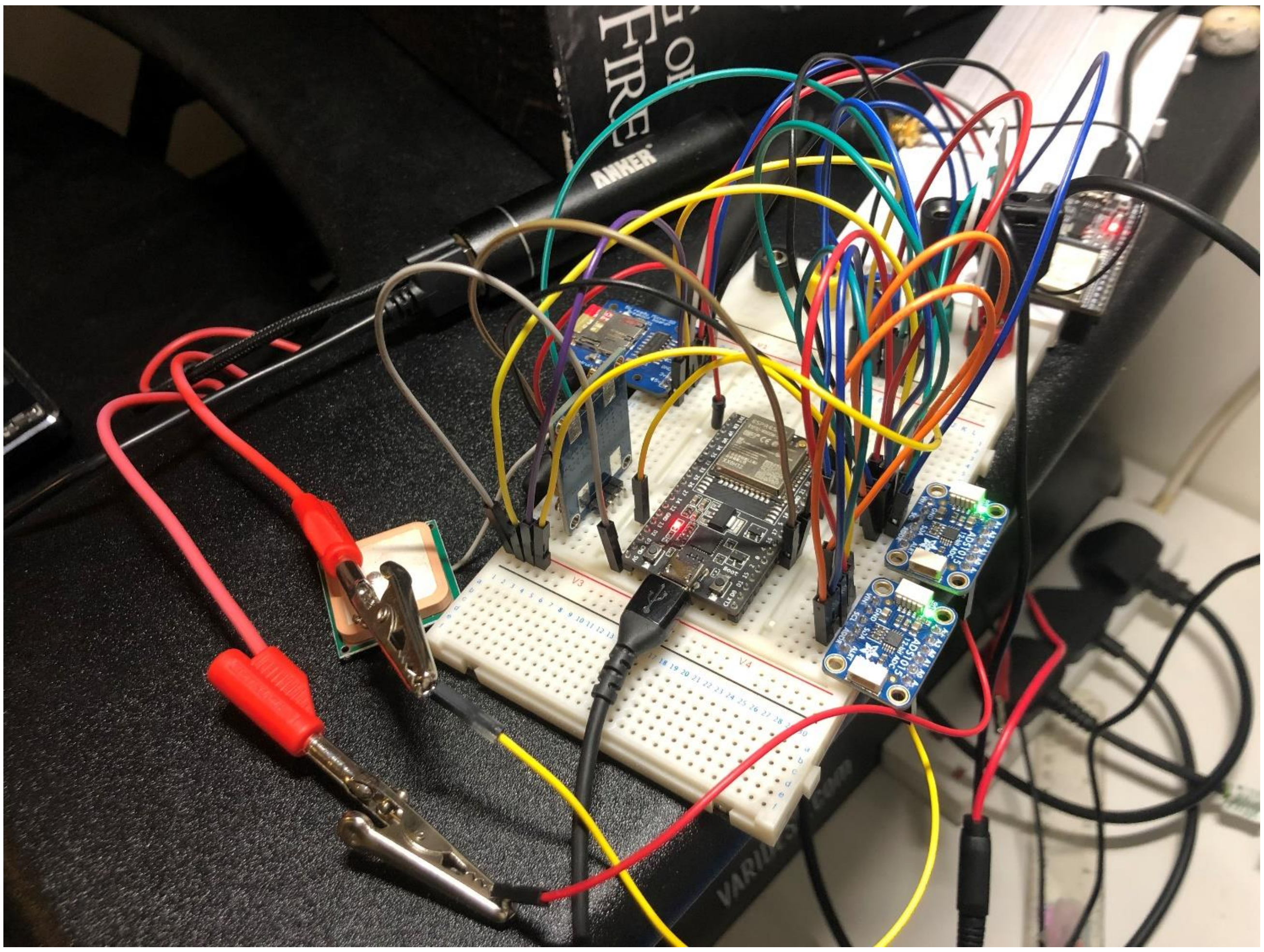


*Figure 103 - ESP32 portable battery bank circuit*

To power the 12V op-amps, a 12V car battery would obviously be an infeasible solution. Therefore, a small and lightweight 8 x AA battery holder was purchased to achieve the 12V solution. Each AA battery is 1.5V, and hence, having 8 of these batteries in series achieves the desired +-12V solution that is needed to power the op-amps. The battery holder was wired up to both of the two op-amps and was able to power them both successfully. A photo of the 8 x AA battery holder was taken after filling it up with fresh batteries and can be seen below, in Figure 104.

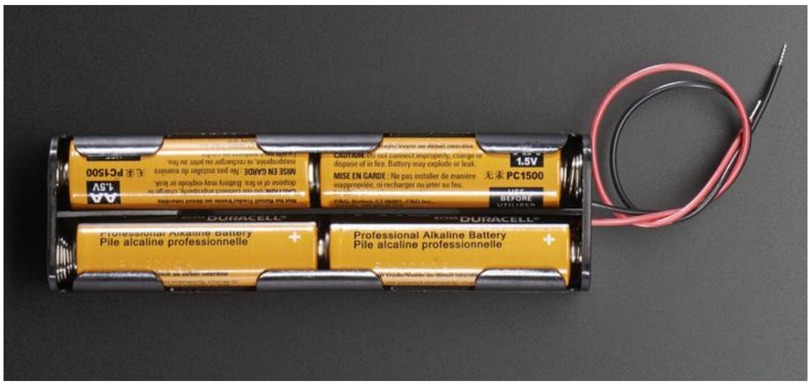


*Figure 104 - 8 x AA battery holder to achieve the 12V solution*

Since both of the ESP32s and both of the op-amps are now being fully powered by batteries, the smart wireless sensor can now be taken and used in any location or scenario without breaching the electrical integrity of the circuit.

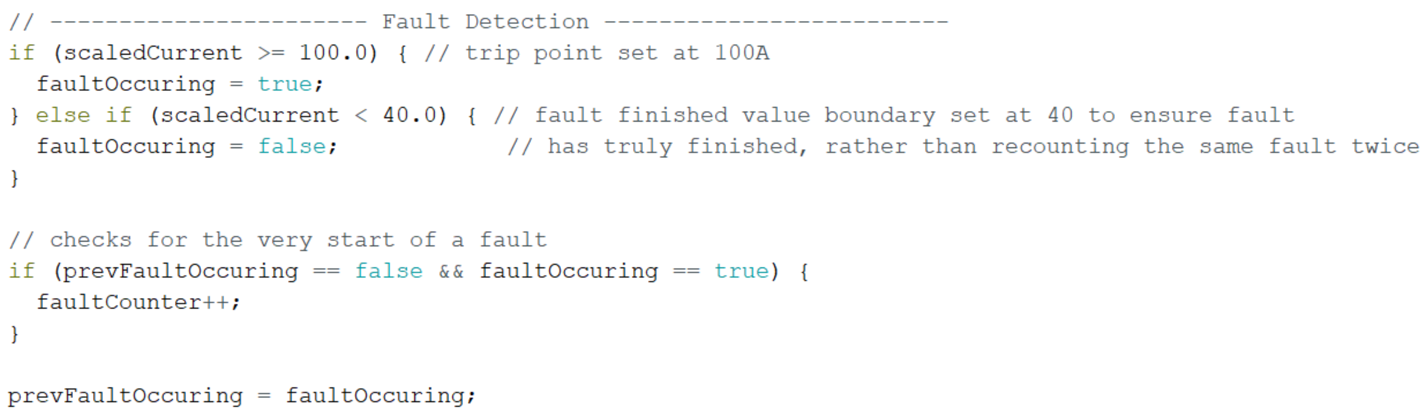
If this smart wireless sensor project was to continue, the next step would be to improve the battery functionality so that the two op-amps and the ESP32s could run off of the same supply. Through simulation, it was found that the op-amps functioned as expected, when being powered by a 9V supply. Using a single 9V lithium-ion battery would be a much smaller and neater solution than the current solution. 9V is too high for the ESP32s as they need a 5V supply. Therefore, a simple voltage divider circuit could be set up to provide 9V to the two op-amps, while also supplying 5V to the two ESP32s.

Research could be done into using a rechargeable 9V battery instead of the 9V alkaline battery, with the possibility of using solar power to recharging the battery – depending on the scenario of the sensor use-case. At the very minimum, it should be very easy and accessible for the user to replace a long-lasting the 9V battery in the circuit, every so often.

## Fault Detection Measurement

In the substation scenario, when there is a fault on the power line, this causes a short circuit and hence, the circuit rises quickly. A trip point is set up so that if the current exceeds this trigger point value, then the protection relay device will trip the circuit breaker, which knocks off the power line supply and switches to the voltage support. In this project, the primary side CT load current is around 20A. A trip point was set to 100A for this project so that whenever the current exceeds 100A for at least one reading, the number of faults will be incremented by one. This was implemented in the code, however, what if the fault stayed above 100A for an extended period of time?

The solution to this problem was solved in the code implementation – the fault is only declared as having finished once the current falls down below 40A again.



```
// ----------------------- Fault Detection -------------------------
if (scaledCurrent >= 100.0) { // trip point set at 100A
  faultOccuring = true;
} else if (scaledCurrent < 40.0) { // fault finished value boundary set at 40 to ensure fault
  faultOccuring = false;            // has truly finished, rather than recounting the same fault twice
}

// checks for the very start of a fault
if (prevFaultOccuring == false && faultOccuring == true) {
  faultCounter++;
}

prevFaultOccuring = faultOccuring;
```

*Figure 105 – Fault detection ESP32 code*

Figure 105, as shown above, showcases the fault detection code in the C++ master ESP32 program. The code checks if the scaledCurrent is detected at being over 100A. If the scaledCurrent was read as below 40A more recently than when it was read at above or equal to 100A, then the faultCounter will be incremented. This ensures that the faultCounter is only incremented once on each new fault, rather than being incremented multiple times during each fault. A value of 40A was chosen as the fault-finished value to ensure that the fault has truly finished occurring – when the scaledCurrent value lowers below 40A it can be safely concluded that the fault has finished occurring.

## Final Circuit

Figure 106, as shown below, showcases the final smart wireless sensor circuit – with the addition of the transformers. The VT and (simulated) CT transformers fed into the two op-amp attenuation circuit, with the rest of the circuit unchanged. The voltage outputs of these two signal conditioning circuits are then fed into the first analogue input of each of the two ADS1015 ADCs which take the 1200sps readings from the signals. The readings are run through the various algorithms and the 12 useful live measurements are then outputted via external communications means to the users.

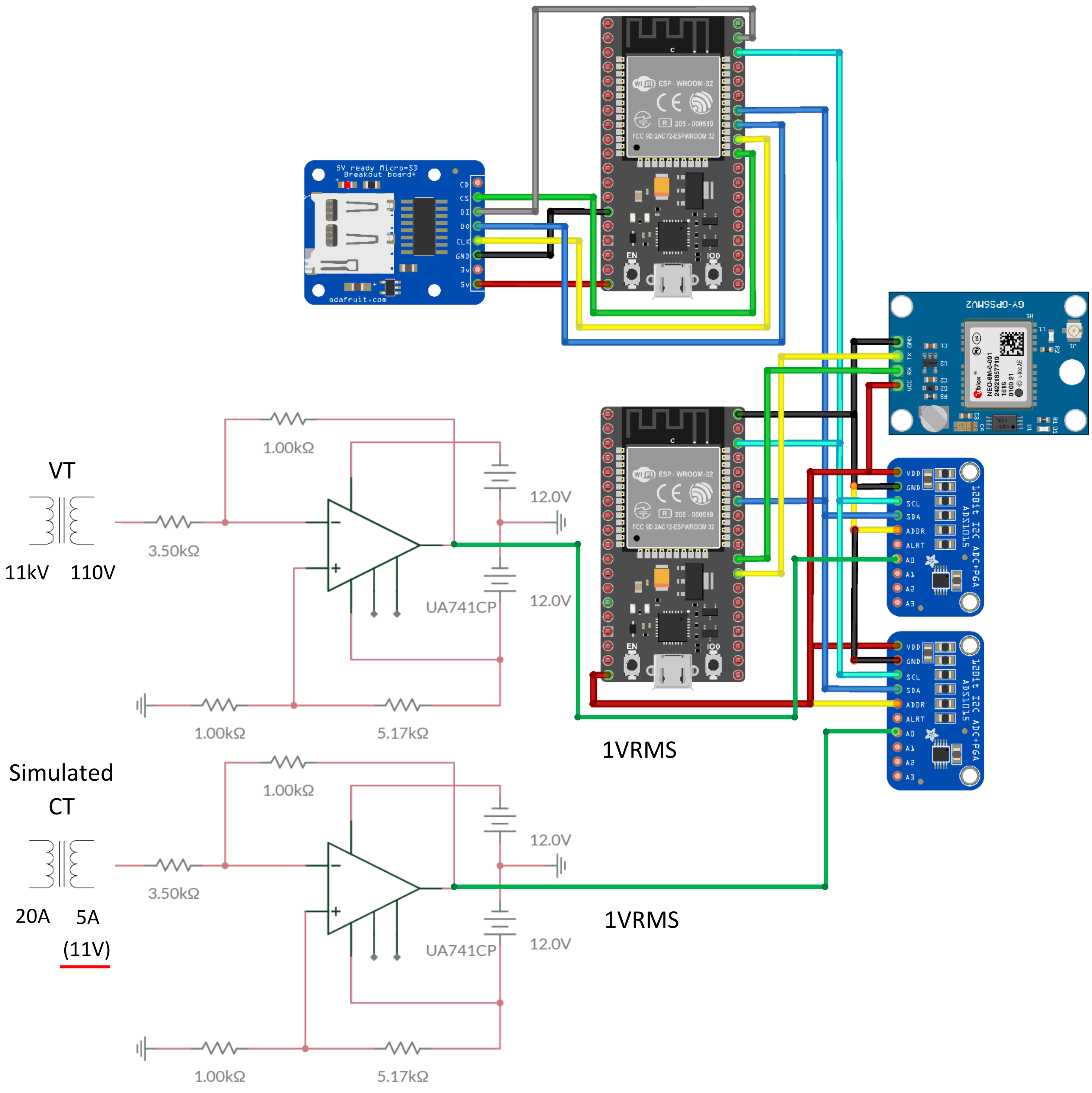


*Figure 106 - the final smart wireless sensor circuit – with the addition of the VT and CT transformers*

## Semester 1 Progress Gantt Chart

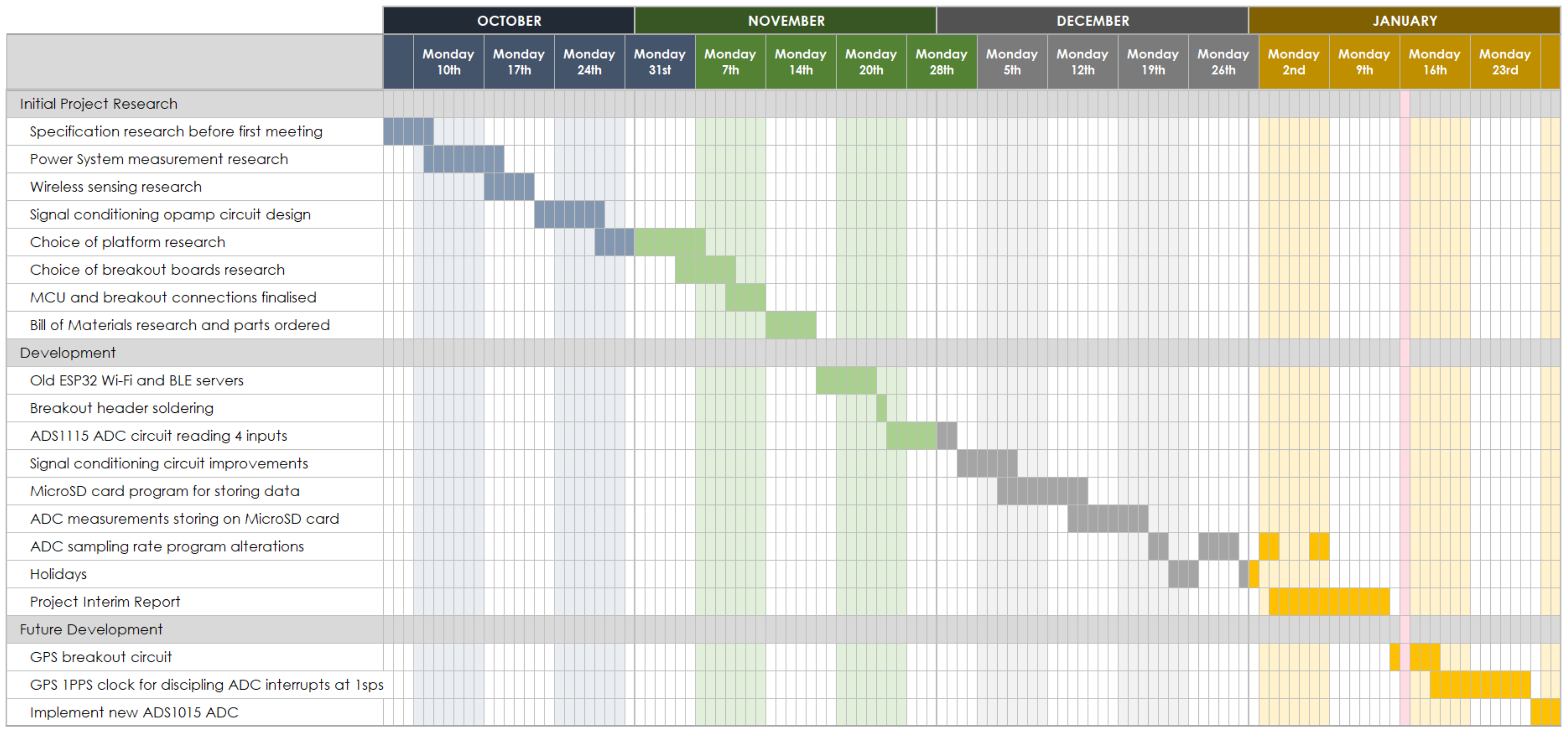


*Figure 107 - Gantt chart to show the project progress made in semester 1*

Interim Report Submission

## Semester 2 Predicted Progress Gantt Chart

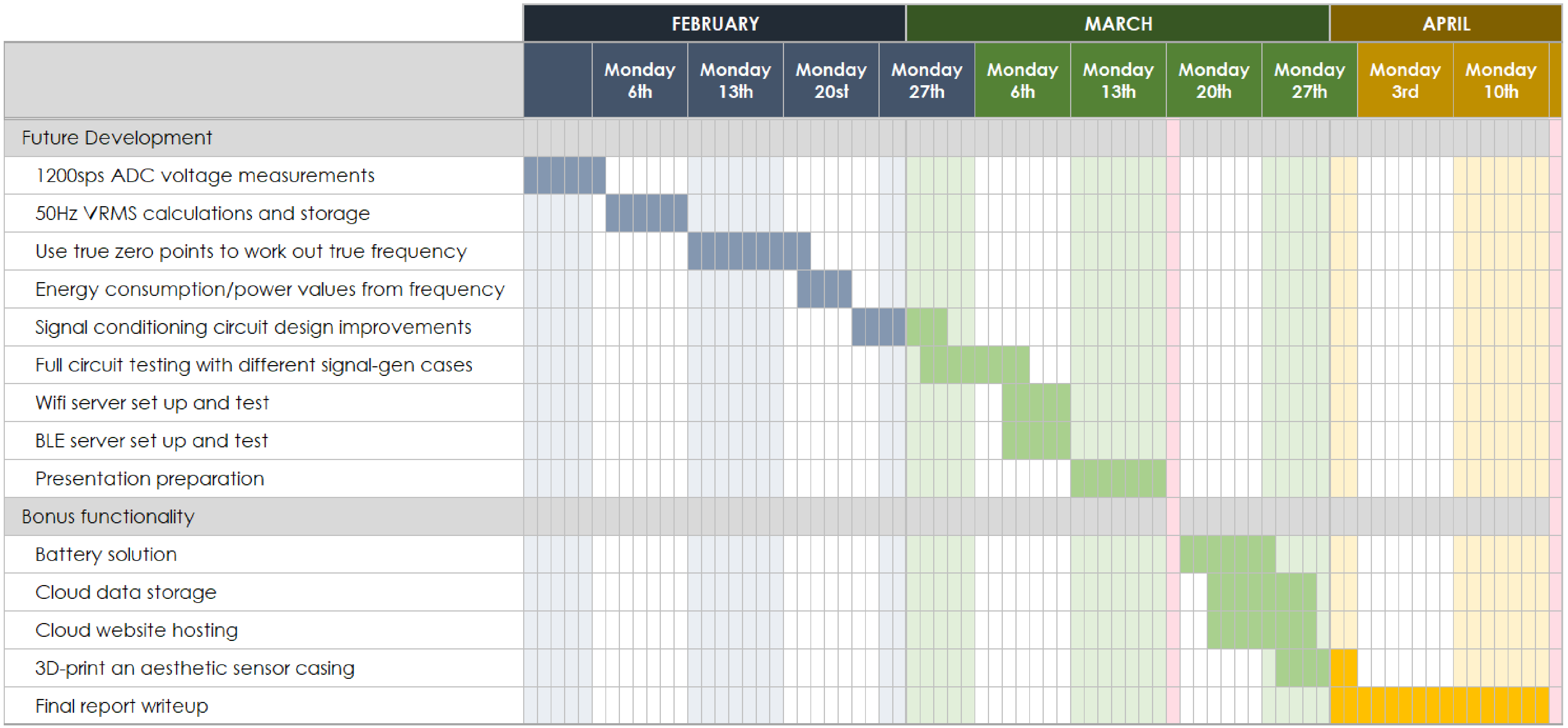


*Figure 108 - Gantt chart to show the project progress that is expected to be made in semester 2*

Oral Examination Week

Final Report Submission

## Semester 2 Recorded Progress Gantt Chart

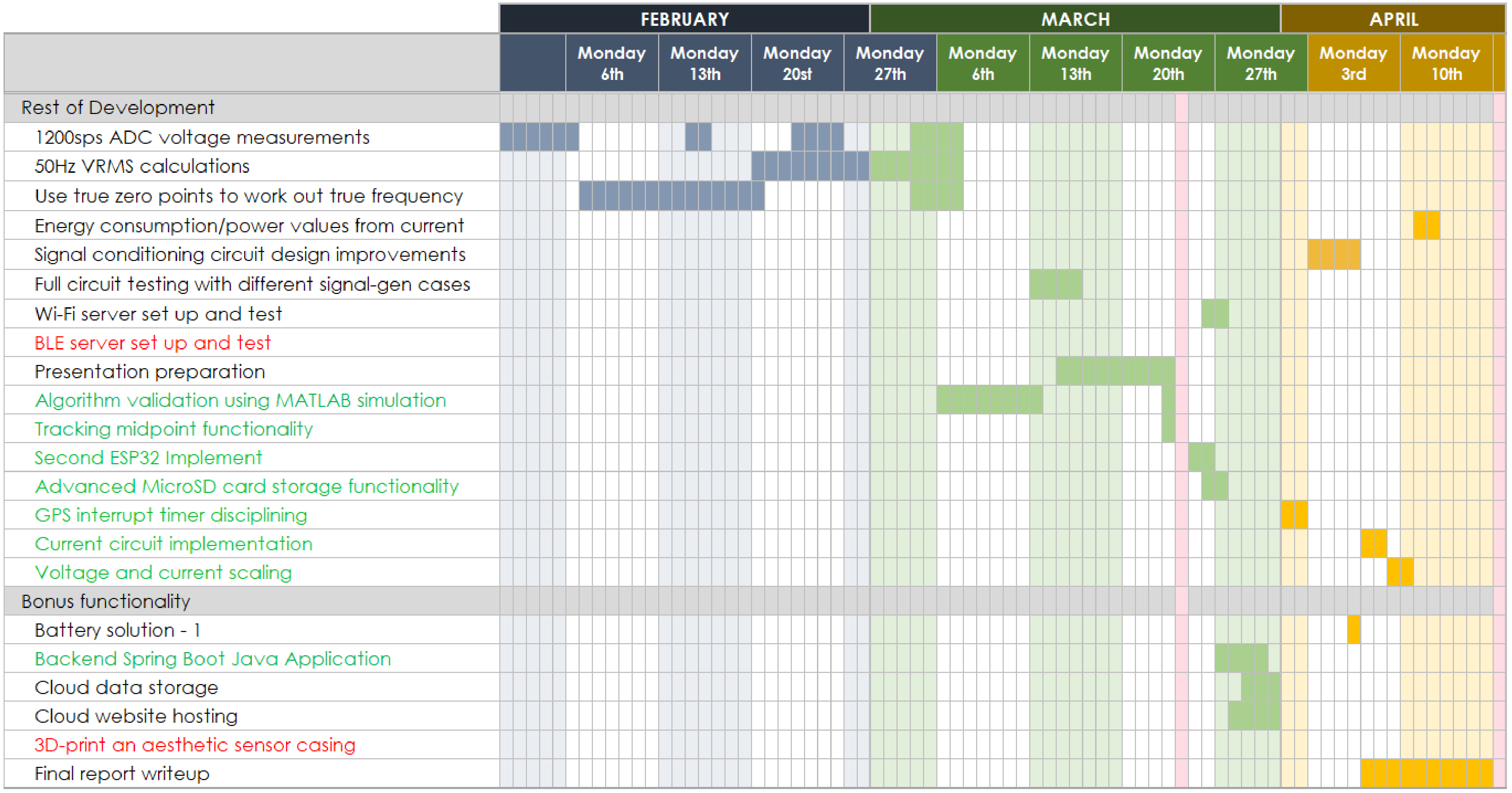


*Figure 109 - Gantt chart to show the project progress that was actually made in semester 2*

Oral Examination Day

Final Report Submission

## Management – Reflection and Analysis Against Plan

Figure 107 showcases the **recorded** progress Gantt Chart of Semester 1.

Figure 108 showcases the **predicted** progress Gantt chart going into the start of Semester 2.

Figure 109 showcases the **recorded** progress Gantt chart of Semester 2.

The progress of semester 1 was demonstrated in the Interim report. The following section will reflect and analyse the semester 2 predicted Gantt chart to the measured Gantt chart (Figure 108 and Figure 109, respectively).

As shown in the predicted semester 2 progress Gantt chart in Figure 108, the chapters with green text colour were tasks that were added in the recorded semester 2 Gantt chart. The chapters in with the red text colour were tasks that were either no longer seen as necessary or they are chapters that did not fulfill the predictions (weren't completed for various reasons).

When visually comparing the predicted and recorded semester 2 Gantt charts, it's very clear that the predicted Gannt chart had a much more linear, waterfall-like flow compared to the actual recorded Gantt chart. This is due to the fact that the order in which the semester 2 tasks were completed was actually very different to the predicted order. This gives the recorded Gantt chart a very messy and chaotic appearance, however, this is perfectly okay as the order of priority and dependency in projects cannot always be truly predicted.

The second noticeable visual difference is fact that the predicted Gantt chart finishes the current task before moving to multiple other tasks, whereas in the recorded Gantt chart, some tasks are reopened up to 4 times – as seen in the "1200sps ADC voltage measurements" task. Reopening and altering tasks is not ideal due to the inefficiency of context switching back to those old tasks, however, reopening old tasks is sometimes a necessary process. This is due to more experience gained further into project, meaning that in hindsight there are certain tasks that must be modified. This is also due to the fact that certain tasks have a cyclical dependency – they depend on the nature of the future tasks, which in turn depend on the past tasks. This leads to both tasks needing to be altered multiple times each, e.g. "1200sps ADC voltage measurements" and "Use true zero points to work out true frequency" tasks, as shown in the recorded Gannt chart.

### Individual tasks analysis

For the individual tasks, the "1200sps ADC voltage measurements" task was predicted to take 8 days. In the recorded chart, it take 8 days initially, however, it had to be reopened 3 more times to make appending changes after making progress with the frequency and VRMS. It had to be reopened due to the fact that the way that the 1200sps sampling rate was achieved in the first 8 days was inadequate for the nature of the VRMS and frequency.

The second task completed was the "Use true zero points to work out true frequency" task. It was predicted to take 8 days, but it actually took 14, due to the complex nature of the task and the fact that the ADC measurements task had to be amended as part of the process.

The next task completed was the "50Hz VRMS calculations" task. The VRMS and frequency tasks were completed in the opposite order to expected, which was fine as they didn't rely on each other. The VRMS task was expected to take 6 days, however, it actually took 16 days. This is due to the overlapping changes with the ADC readings and frequency, hence causing a delay in getting the VRMS functionality fully completed.

The next task was the “Algorithm validation in MATLAB simulation” task. This task was not predicted in the predicted Gantt chart, however, it proved to be a quintessential step for the project. The analogue signals themselves are not consistent enough for validating the algorithms so the analogue signals were simulated in MATLAB then the algorithms were ran on the signals, proving that the algorithms work as intended, and are valid to the required degree.

The next task completed was “Full circuit testing with different signal-gen cases”. This task was expected to be done after the energy consumption task, which actually ended up being the last task completed, which was poorly predicted. This task took the expected time of 4 days total, which is half the time that it was expected to take. This task paid off well as it ended up facilitating a smooth oral presentation demo.

The next task was the “Presentation preparation” task. The presentation date was marked as the Monday on the predicted chart, however, the actual date was on the Friday. This allowed for more time to prepare for the presentation, totaling at a respectable 9 days total.

The next task was the “Tracking midpoint functionality” task, which was completed on the day before the presentation. This was a last minute edit which thankfully worked perfectly at tracking the midpoint, even when varying the DC Offset. This functionality was not predicted and ended up being a great addition to the project, allowing the VRMS to be smoothly calculated, no matter how varying the DC offset is.

After the oral presentation, a lot more time was freed up for working on the project. The large amount of the semester 2 functionality was completed in the next 2 weeks. The next task was an unpredicted task, the “Second ESP32 Implement” task. This was a crucial task to be completed as the subsequent task all relied on the second ESP32 to be in place before they could be attempted.

The next task completed was the “Wi-Fi server set up and test” task. The basic Wi-Fi server code had already been set up in semester 1, however, to work in both station and AP mode there was a lot of additional intricate complexity that needed added to the code. It took 2 full days in total, which was great. This task was completed later than expected, as it was expected to be completed before the presentation.

The next task completed was the “Advanced MicroSD card storage functionality”. This task was not predicted in the predicted Gantt chart, however, it proved to be a vital addition to the project. It only took 2 days to get the full, complex format working, thanks to the work done in semester 1.

The next 3 tasks were heavily overlapping over a 5 day period. This was the full cloud functionality setup. The first task was unpredicted – the “Backend Spring Boot Java Application”. When creating the prediction Gantt chart, the methodology of how the cloud functionality would be implemented was completely unknown, so it was no surprise that this additional task had to be completed as part of setting up the cloud apps. This backend app was needed as a middleman between the ESP32 and the Google Cloud Platform (GCP) React JS website. Since the website calls the Spring Boot endpoints before storing the sensor readings in the relational could database, the website hosting task was started before the cloud data storage task – which was unpredicted.

The next task completed was the “GPS interrupt timer disciplining” task. This task was not predicted in the predicted Gantt chart, however, this task was essential to ensure that smart wireless sensor would only start taking readings on the exact start of a new second. This is crucial for working with multiple smart wireless sensors, to have the readings exactly in sync.

The next task completed was the “Signal conditioning circuit design improvements” task. This task took slightly less time than expected, at 4 days instead of 7. This task was predicted as the 5th task to be completed, however it was done towards the end, due to the fact that it was key to ensure that the main ESP32 was all set up correctly to ensure that the signals coming from the signal conditioning circuit would be as expected. The op-amp calculations were fixed and the voltage signal conditioning circuit was built.

The next task completed was the “Battery solution” task, which was completed a bit later than expected. It only took 1 day, whereas it was predicted to take 8 days. This is thanks to the simplicity of the solution itself – a portable power bank to power the ESP32s and a 8 x AA 12V battery holder to power the two op-amps.

The next task completed was the “Current circuit implementation” task. This task was somehow overlooked in the predicted chart and hence came in towards the end of the project as additional functionality. A second op-amp circuit was set up, exactly the same as the voltage op-amp circuit, and a voltage divider circuit was added to the function generator input signal to obtain a simulated current signal roughly 10% the amplitude of the main voltage signal. This signal was the passed into the current op-amp circuit.

The next task completed was the “Voltage and current scaling” task. This task was an additional task added into the recorded Gantt chart. This task essentially scaled the measured analogue input voltage values to the VT and CT primary side full peak voltage and current values, respectively.

The final actual project task completed was the “Energy consumption/power values from current” task. This task was meant to be completed as the fourth task in the predicted chart, however, the current acquisition process was clearly overlooked on creation of that predicted chart. This final task depended on a lot of tasks being completed first and hence ended being last. The scaled voltage and scaled current were utilised to calculate the power, then the energy consumption was the final value to be calculated and sent to the backend and cloud, tying off the project.

The “Final report writeup” task was predicted to take 16 days, however, it actually took 11 days. The final bits of project functionality took slightly longer than expected, which delayed the starting of the report. 11 days ended up being very sufficient for writing the report, thanks to the structure of the Winter Interim report saving a lot of time.

## Redundant tasks and uncompleted tasks

The “BLE server set up and test” task – shown in red – was uncompleted due to the fact that (during semester 2 it was found that) the ESP32 cannot use both of its Wi-Fi and Bluetooth modules at once, therefore, a choice had to be made between hosting a Wi-Fi server or BLE server. Since the ESP32 needed internet access for obtaining the GMT time for SD card file naming and also for cloud data uploading, it was an obvious choice to choose the Wi-Fi server. The Wi-Fi server was set up both Station mode and access point (AP) format, where the latter ended up acting very similar to how the BLE server would have worked anyway. This task was therefore coloured red due to the fact that it was made redundant by this technical limitation.

The “3D-print and aesthetic sensor casing” bonus task – shown in red – was uncompleted due to other task prioritisations taking its place. As bonus functionality for the future, the smart wireless sensor would be hosted on a neat stripboard/Veroboard, hence bringing down the total sensor size and paving the way for a sensor casing to be a great addition.

## Summary/Overview of the Smart Wireless Sensor Management

The project handbook suggested leaving 20% of the time reserved for unforeseen delays (extra tasks) but this was not account for in the predicted semester 2 Gantt chart. In hindsight, this is a good idea and shall be implemented in any future Gantt charts.

Overall, the predicted progress Gantt chart ended up being relatively accurate in terms of the task completed, however, the ordering of the tasks was slightly off – which is okay. Due to other university module assignments and responsibilities, the number of tasks completed from February up until the oral presentation (on the 25th March) was lower than the number of tasks completed in the few weeks following the presentation. Those tasks completed before the 25th March also took a lot longer per task when viewed on the chart. This is simply due to the fact that the number of hours per day being allocated to the tasks before the 25th March was a lot lower than the few weeks after. From the 25th March until the 17th April (report submission date), this project was the sole focus for those three and a half weeks, facilitating a lot of project work being able to be completed, thanks to large time blocks and minimised context switching between various forms of university work.

## Possible Future Work

If this smart wireless sensor project was to continue, the next step would be to improve the battery functionality so that the two op-amps and the ESP32s could run off of the same supply. Currently the ESP32s are being supplied by a portable power bank and the op-amps are being power by a 12V 8 x AA battery holder. Through simulation, it was found that the op-amps functioned as expected, when being powered by a 9V supply. Using a single 9V lithium-ion battery would be a much smaller and neater solution than the current solution. 9V is too high for the ESP32s as they need a 5V supply. Therefore, a simple voltage divider circuit could be set up to provide 9V to the two op-amps, while also supplying 5V to the two ESP32s.

A 4G sim card breakout could be integrated into the circuit to ensure that the sensor will almost always have internet access. This would maximise the scenarios where the sensor would have internet access as the sensor will not always have a local network to connect to nearby. This would be extremely beneficial in ensuring that the cloud-based website and cloud data storage is utilised as much as possible, rather than the local Wi-Fi AP server having to be used every time.

The power consumption of the project should then be optimised to use the least amount of power possible. Research could be done into using a rechargeable 9V battery instead of the 9V alkaline battery, with the possibility of using solar power to recharging the battery – depending on the scenario of the sensor use-case. At the very minimum, it should be very easy and accessible to replace the 9V battery in the circuit.

The next step would be to migrate the project from the breadboard to a stripboard/Veroboard. This is a more semi-permanent, smaller, and neater solution. After that, a the next step would be to 3D print a suitable sensor casing, and attach suitable voltage and wire clamps.

Once the prototype has been sufficiently tested for possible improvements, these improvements should be made then a PCB should be designed and produced. This would result in a much smaller and more accessible solution, hence reaching the full project potential. An aesthetic casing to hold the PCB and components should be designed and produced, while allowing easy access to the 9V battery, should it need to be replaced.

## Contribution – Comparison against specification

### Specification Objectives

**Objectives**

1. Investigate power system measurement and wireless sensing
2. Design a near-real-time solution in hardware and software to measure low voltage/current signals
3. Develop appropriate hardware/host software for wireless PC/laptop/GUI control and interfacing
4. Write C/C++ software to implement the solution for near-real-time measurement
5. Write C/C++ software to implement near-real-time determination of other quantities (e.g. power/energy)
6. Evaluate and test the smart sensors with different operating scenarios

**Bonus MEng Extension Objectives**

1. Develop appropriate hardware and software for bi-directional sensor measurement interfacing
2. Develop a broad range of operating scenarios with different voltage/current attributes for smart metering
3. Implement and evaluate final solution with simulated/measured/laboratory signals

The full specification page can be seen at the start of this project report. The main 6 project objectives are listed above, along with the 3 bonus MEng extension objectives. This project is a BEng project, however, so these MEng extension objectives were not expected to be completed.

### Required Specification Objectives Table

| BEng Objectives | Completed |
|---|---|
| 1. Investigate power system measurement and wireless sensing | ✓ |
| 2. Design a near-real-time solution in hardware and software to measure low voltage/current signals | ✓ |
| 3. Develop appropriate hardware/host software for wireless PC/laptop/GUI control and interfacing | ✓ |
| 4. Write C/C++ software to implement the solution for near-real-time measurement | ✓ |
| 5. Write C/C++ software to implement near-real-time determination of other quantities (e.g. power/energy) | ✓ |
| 6. Evaluate and test the smart sensors with different operating scenarios | ✓ |

*Figure 110 - Bonus Objectives Tables (Not required by BEng specification)*

## Bonus Objectives Tables (Not required by BEng specification)

| MEng Bonus Objectives | Completed |
|---|---|
| 1. Develop appropriate hardware and software for bi-directional sensor measurement interfacing | ✓ |
| 2. Develop a broad range of operating scenarios with different voltage/current attributes for smart metering | ✓ |
| 3. Implement and evaluate final solution with simulated/measured/laboratory signals | ✓ |

*Figure 111 - MEng Bonus Objectives*

| Further Bonus Objectives (only some listed) | Completed |
|---|---|
| 1. Backend Spring Boot Java Application system for relaying HTTP requests between slave ESP32 and the cloud-side | ✓ |
| 2. Cloud website hosting to display up-to-date sensor readings that updates asynchronously periodically | ✓ |
| 3. Cloud data storage to store the sensor readings in a clear and structured file and folder relational format, for intuitive navigation of sensor readings | ✓ |
| 4. Validate the frequency and VRMS algorithms using simulated analogue signals (MATLAB) to conclude if their precision is better than +-10mHz and +- 10mV | ✓ |
| 5. Advanced MicroSD card storage system with file and folder funneling based on updated current GMT time. | ✓ |
| 6. Develop a suitable algorithm for varying DC offset tracking, without affecting VRMS measurements | ✓ |
| 7. Develop consistent interrupt timer disciplining through the use of a 1PPS GPS signal, facilitating multiple sensors to always take readings in sync | ✓ |
| 8. Maximise code sustainability (green code) – Use industry-leading data structures and algorithms techniques to improve microcontroller algorithm efficiency, hence reducing power consumption and improving code sustainability – a worldwide recognised issue | ✓ |

*Figure 112 - Further Bonus Objectives (only some listed)*

This project is a BEng project, therefore, the only required specification objectives of this project can be seen as the 6 objectives listed in Figure 110. These objectives were all completed and hence the project specification was fulfilled.

Even though this project is only a BEng project, all 3 of the MEng Bonus Objectives were completed, hence showing more added value, beyond the specification. This is shown in Figure 111

Figure 112 showcases the Further Bonus Objectives table. These 8 bonus objectives were self-proposed and completed, bringing the project to an even higher standard.

**In summary, all 6 required objectives were fulfilled and an additional 11 bonus objectives were completed, hence adding very strong value beyond that resulting from just fulfilling the specification.**

## Discussion

This project sought to develop an appropriate solution to compensate for the downsides of Digital Fault Recorders. Individual large and heavy DFRs in Distributed DFR systems must be hardwired to a Data Concentrator by means of serial or digital communication. This in turn leads to inaccessible live measurements and a restriction as to where the DFRs can be placed without breaching the electrical integrity of the system. A smart sensor was developed to augment DFRs to solve these problems.

The overall functionality required of the sensor was determined and was demonstrated in semester 1 in Figure 1. For all the researched platforms, it was found that the built-in platform ADC inputs would fail to provide asynchronous 8-channel measurements so this functionality had to be acquired externally. Through detailed research, the most suitable platform was found to be the ESP32-DevKitC-32UE due to its strong processing power, small form factor, reliability, pin inputs/outputs, etc. To make up for missing functionality, suitable breakout boards were researched and narrowed down to a final bill of materials order shown in Table/Figure 7 – the communication forms of these breakouts are shown in Figure 8. The transmission side of the transformer is around 110V and the ADC can only take in a range of roughly -0.3 – 5.3V (with a 5V supply) so a signal conditioning inverting op-amp attenuation circuit was designed, with a voltage shift of 2.5V to ensure that the negative half of the signal wave could be measured. It was found that the ADS1115 could successfully store measurements on the microSD card but its maximum sampling rate was found to be too low – at only 860sps, whereas 1200sps is needed for a 50Hz signal at 24 samples/cycle – so the 3300Hz sampling rate model was ordered (ADS1015). 3300sps voltage readings were achieved then a true zero point algorithm was implemented, to facilitate the true frequency to be measured. Midpoint, VRMS and current algorithms were designed and implemented too. These algorithms were then validated through simulation in MATLAB and found to be 33 times more accurate than originally required. A GPS breakout was utilised for interrupt timing disciplining so that multiple sensors could be used in sync. The sensor was tested thoroughly with a function generator before designing the final op-amp circuits to facilitate the voltage and simulated current to be measured and scaled to represent the primary side VT and CT voltages and currents, respectively. A second ESP32 was added to the circuit to handle external communications and a battery solution was implemented. An advanced local MicroSD card storage file/folder system was further refined for storing the 12 measurements. A full backend cloud data pipeline was implemented – to facilitate the 12 useful live measurements to be asynchronously displayed on a cloud-hosted website and stored in cloud data storage for later access. A local Access Point & Station Wi-Fi server was added on the slave ESP32, so that readings could be read, even in absence of Wi-Fi.

The wireless sensor fault analysis functionality is inferior to that of the standalone DFRs – as DFRs have higher sampling rate/resolution, higher memory capacity, improved datasets, etc. – as they are specifically designed for fault analysis (among other purposes) and cost over 20 times the price [28]. The purpose of the smart wireless sensor is not to replace DFRs but to augment DFRs to make up for their limitations. The sensor solution can be used in combination with DFRs to provide live readings to any users, through the cloud-hosted website and cloud storage, while retaining the superior fault analysis functionality of the DFRs. The smart wireless sensors can be placed in locations that were previously electrically impossible for the DFRs due to the sensors' wireless communications and their lightweight, small form factor. In these locations where the DFR cannot be placed, the smart wireless sensor provides an adequate solution for fault analysis, when this was previously impossible.

Many previous studies investigate the optimisation of DFR systems. This project takes already optimised DFR systems and augments them with further functionality.

## Conclusion

The project sought to develop a smart wireless sensor MCU solution to solve the problems that industry-standard 3-phase Distributed Digital Fault Recorders currently face – inaccessible live measurements and electrical-based restrictions for DFR placement. Research was conducted to investigate the possibility of a smart wireless sensor platform solution to augment DFRs to compensate for currently lacking functionality.

The end-goal specification was used to determine the requirements of the platform which included Wi-Fi, Bluetooth, 1200sps ADC reading functionality, a 1PPS GPS signal for interrupt syncing, MicroSD card storage, and a possible battery supply solution. The suitable platform had to be chosen. An MCU was concluded as more suited to this project than an SBC. The ESP32 was concluded as the optimal MCU for this project and it came with built-in Wi-Fi and Bluetooth but lacked the rest of the additional functionality. The ADC of the ESP32 is incapable of the required 8-channel asynchronous measurements so an external ADC, MicroSD card, and GPS breakout were researched and ordered. The secondary side of the transformer is around 110V and the ADC can only take in a range of roughly -0.3 – 5.3V, so a signal conditioning inverting op-amp attenuation circuit was designed, with a voltage shift of 2.5V to ensure that the negative half of the signal could be measured.

A circuit was created consisting of the ESP32, ADC, and MicroSD card breakouts and a program was developed to take 4 singled-ended input voltage values and store them on the MicroSD card. A higher refresh rate ADC was purchased then its library was optimised to facilitate the full 3300sps readings to be taken. Consistent 1200sps readings were then set up, as required for a 50Hz mains signal of 24 samples/cycle, by utilising the ESP32's built-in interrupt clock. The GPS breakout was implemented for interrupt timer disciplining. True frequency, midpoint, VRMS and current algorithms were designed and implemented into the ESP32 code to facilitate all the required measurements to be calculated for later use. These algorithms were then validated through a simulation in MATLAB and found to be 33 times more accurate than originally required (+-0.3mHz for true frequency and +-0.3mV for VRMS). The circuit was thoroughly tested with a function generator before designing the final op-amp circuits to facilitate the voltage and simulated current to be measured and scaled to represent the primary side VT and CT voltages and currents, respectively. A second ESP32 was added to the circuit to handle external communications and a battery solution was implemented, for portability benefits. An advanced local MicroSD card storage file/folder system was refined, along with a full backend cloud data flow – for the sensor readings to be asynchronously displayed on a cloud-hosted website and cloud data storage for later access. A local Access Point & Station Wi-Fi server was added, so that readings could be read, even in absence of Wi-Fi (via the access point).

In reflection, this project demonstrates the value provided and the potential that the smart wireless sensor offers. It will be able to solve the two biggest problems of DFRs by augmenting them with additional functionality. Users can read the useful live/near-real-time measurements – such as number of faults, voltage, current, energy consumption, power consumption, etc. – via the cloud-hosted website, cloud storage, and local station web server. However, if the sensor is in the absence of a network connection, then the readings can still be accessed through the platform Wi-Fi AP server – if the user is within the 2.4GHz Wi-FI range of the sensor. It's a non-invasive solution, the voltage and current connections just need to be connected with a clamp. The sensor is battery-powered, wireless, portable, and low maintenance, hence making it the perfect solution to be clamped on almost anywhere that it would be electrically impossible to do so with a DFR.

The DFR costs over 20 times the price of the smart wireless sensor [28] – as a result, the DFR has superior fault analysis functionality. This is a limitation of the smart wireless sensor but, when used in combination with the DFR, the combined fault analysis of the DFR and live measurements provide an unmatched solution.

This augmented DFR solution has proven to be an extremely effective solution to solve the main pitfalls of standalone DFRs in distributed DFR systems. Useful and accessible live readings being transmitted to a cloud-hosted website, cloud data storage, and an ESP32 Access Point & Station Web server mean that the readings can seamlessly be read worldwide, as well as by anyone connected to the local network, or even without a local Wi-Fi network – by anyone nearby. This, combined with being able to place the sensor in previously inaccessible DFR placement locations, has successfully fulfilled the proof of concept and pushed the boundaries of smart wireless communications in substations.

Further research and development could be invested into the smart wireless sensor to facilitate an improved battery system with the use of a single 9V battery, rather than the current portable power bank and 12V solution. A rechargeable solution could be engineered, with the possibility of using more sustainable energy sources, such as using solar panels, etc. 4G internet capabilities could be implemented to ensure that readings are uploaded to the cloud website and storage, regardless of whether there's a local Wi-Fi network nearby. The smart wireless sensor could be moved from the breadboard to a more semi-permanent solution, such as a stripboard/Veroboard. After thorough prototype testing, a final PCB solution could be designed and engineered, along with an aesthetic casing – facilitating the smart wireless sensor to reach its full project potential.

# Appendices

## Appendix 1: 4-channel ADC measurements storing to MicroSD Card C++ program written in semester 1

```
#include <Adafruit_ADS1X15.h>
#include "FS.h"
#include "SD.h"
#include "SPI.h"
#include <iostream>
#include <string>

using namespace std;

// set up ADS1115
Adafruit_ADS1115 ads;

void writeFile(
  fs::FS &fs,
  const char * path,
  const char * message
) {
  Serial.printf("Writing file: %s\n", path);

  File file = fs.open(path, FILE_WRITE);
  if (!file) {
    Serial.println("Failed to open file for writing");
    return;
  }
  if (file.print(message)) {
    Serial.println("File written");
  } else {
    Serial.println("Write failed");
  }
  file.close();
}

void appendFile(
  fs::FS &fs,
  const char * path,
  const float volts0,
  const float volts1,
  const float volts2,
  const float volts3
) {
  Serial.printf("Appending to file: %s\n", path);

  String message = "-------------------------------------------------------
  ----\nAIN0: "
                   + String(volts0, 6) + "V\n"
                   + "AIN1: " + String(volts1, 6) + "V\n"
                   + "AIN2: " + String(volts2, 6) + "V\n"
                   + "AIN3: " + String(volts3, 6) + "V\n";

  File file = fs.open(path, FILE_APPEND);
  if (!file) {
    Serial.println("Failed to open file for appending");
    return;
  }
  if (file.print(message)) {
    Serial.println("Message appended");
  } else {
    Serial.println("Append failed");
  }
  file.close();
}

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

  // set up SD card
  if (!SD.begin()) {
    Serial.println("Card Mount Failed");
    return;
  }

  uint8_t cardType = SD.cardType();
  if (cardType == CARD_NONE) {
    Serial.println("No SD card attached");
    return;
  }

  // create the voltages text file
  writeFile(
    SD,
    "/ADS1115_voltage_measurements.txt",
    "ADS1115 Voltage Values\n"
  );

  if (!ads.begin()) {
    Serial.println("Failed to initialise ADS.");
    while (1);
  }
}

void loop(void)
{
  int16_t adc0, adc1, adc2, adc3;
  float volts0, volts1, volts2, volts3;

  adc0 = ads.readADC_SingleEnded(0);
  adc1 = ads.readADC_SingleEnded(1);
  adc2 = ads.readADC_SingleEnded(2);
  adc3 = ads.readADC_SingleEnded(3);

  volts0 = ads.computeVolts(adc0);
  volts1 = ads.computeVolts(adc1);
  volts2 = ads.computeVolts(adc2);
  volts3 = ads.computeVolts(adc3);

  Serial.println("-----------------------------------------------------------
--");
  Serial.print("A0: "); Serial.print(volts0); Serial.println("V");
  Serial.print("A1: "); Serial.print(volts1); Serial.println("V");
  Serial.print("A2: "); Serial.print(volts2); Serial.println("V");
  Serial.print("A3: "); Serial.print(volts3); Serial.println("V");
  appendFile(
    SD,
    "/ADS1115_voltage_measurements.txt",
    volts0,
    volts1,
    volts2,
    volts3
  );

  delay(5);
}
```

## Appendix 2: Arduino MCU Comparison Table

| Platform | Chip | Operating Voltage | Flash Memory | SRAM | EEPROM | Clock Speed | Analogue Input Pins | Digital I/O Pins | PCB Size | Weight | Wi-Fi | Bluetooth | Price |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arduino Micro | ATmega32U4 | 5V | 32KB | 2.5KB | 1KB | 16MHz | 12 | 20 (7 PWM) | 13 x 48 mm | 13g | n | n | €21.60 |
| Arduino Nano | ATmega328 | 5V | 32KB | 2KB | 1KB | 16MHz | 8 | 22 (6 PWM) | 18 x 45 mm | 7g | n | n | €21.60 |
| Arduino Nano 33 IoT | SAMD21 Cortex®-M0+ 32bit | 3.3V | 256KB | 32KB | None | 48MHz | 8 | 14 (11 PWM) | 18 x 45 mm | 5g | y | y | €20.80 |
| Arduino Uno Rev3 | ATmega328P | 5V | 32KB | 2KB | 1KB | 16MHz | 6 | 14 (6 PWM) | 53 x 69 mm | 25g | n | n | €24.00 |
| Arduino Uno WiFi Rev2 | ATmega4809 | 5V | 48KB | 6,144B | 256B | 16MHz | 6 | 15 (5 PWM) | 53 x 69 mm | 25g | y | y | €46.70 |
| Arduino Mega 2560 Rev3 | ATmega2560 | 5V | 256KB | 8KB | 4KB | 16MHz | 16 | 54 (15 PWM) | 53 x 102 mm | 37g | n | n | €42.00 |
| Arduino MKR WiFi 1010 | SAMD21 Cortex®-M0+ 32bit | 3.3V | 256KB | 32KB | None | 48MHz | 7 | 8 (13 PWM) | 25 x 52 mm | 32g | y | y | €33.50 |
| Arduino Portenta H7 | STM32H747XI dual Cortex®-M7+M4 32bit | 3.3V | 2MB | 1MB | None | 480MHz | 8 | 84 (10 PWM) | 25 x 66 mm | 30g | y | y | €99.00 |
| Arduino Nano RP2040 Connect | Dual-core 32-bit Arm Cortex-M0+ processor | 3.3V | 2MB | 264kB on-chip SRAM | None | 133 MHz (up to) | 8 | 14 (10 PWM) | 18 x 45 mm | 6g | y | y | €20.16 |

*Figure 113 - Table used for comparing the Arduino MCU models. The Arduino Nano RP2040 Connect was found to be the most suited Arduino MCU for this project*

## Appendix 3: Raspberry Pi SBC Comparison Table

| Platform | Chip | Operating Voltage | RAM | Clock Speed | Digital I/O Pins | PCB Size | Weight | Wi-Fi (2.4GHz) | Bluetooth | Price |
|---|---|---|---|---|---|---|---|---|---|---|
| Raspberry Pi 4 | Quad core Cortex-A72 (ARM v8) 64-bit | 5V | 1/2/4/8GB SDRAM | 1.5GHz | 40 | 49 x 85 mm | 46g | y (5GHz too) | y | £40.00 (1GB RAM model) |
| Raspberry Pi Zero | Single core | 3.3V | 512MB | 1GHz | 40 (unpopulated) | 30 x 65 mm | 16g | n | n | £5 |
| Raspberry Pi Zero W | Single core | 3.3V | 512MB | 1GHz | 40 (unpopulated) | 30 x 65 mm | 16g | y | y (4.1 and BLE) | £10 |
| Raspberry Pi Zero 2 W | Quad-core 64-bit Arm Cortex-A53 | 3.3V | 512MB (SDRAM) | 1GHz | 40 (unpopulated) | 30 x 65 mm | 16g | y | y (4.2 and BLE) | £14 |
| Raspberry Pi Pico WH RP2040 Microcontroller | Dual-core 32-bit Arm Cortex-M0+ processor | 3.3V | 264kB on-chip SRAM | 133 MHz (up to) | 26 (3 analogue inputs) | 21 x 51 mm | 10g | y | n | £5 |

*Figure 114 - Table used for comparing the Raspberry Pi SBC models. The Raspberry Pi Zero 2 W was found to be the most suited Raspberry Pi SBC for this project*

## Appendix 4: Other Platforms Specifications

### Teensy 4.1 Development Board [29]

- ARM Cortex-M7 at 600 MHz
- 7936K Flash, 1024K RAM (512K tightly coupled), 4K EEPROM (emulated)
- 55 digital input/output pins, 35 PWM output pins
- 18 analogue input pins
- 8 serial, 3 SPI, 3 I2C ports
- 1 SDIO (4-bit) native SD Card port

### ESP8266 (ESP32) [30]

- Processor: L106 32-bit RISC microprocessor core based on the Tensilica Diamond Standard 106Micro running at 80 or 160 MHz
- 32 KiB instruction, 80 KiB user data
- IEEE 802.11 b/g/n Wi-Fi
- 17 GPIO pins
- 10-bit ADC (successive approximation ADC)

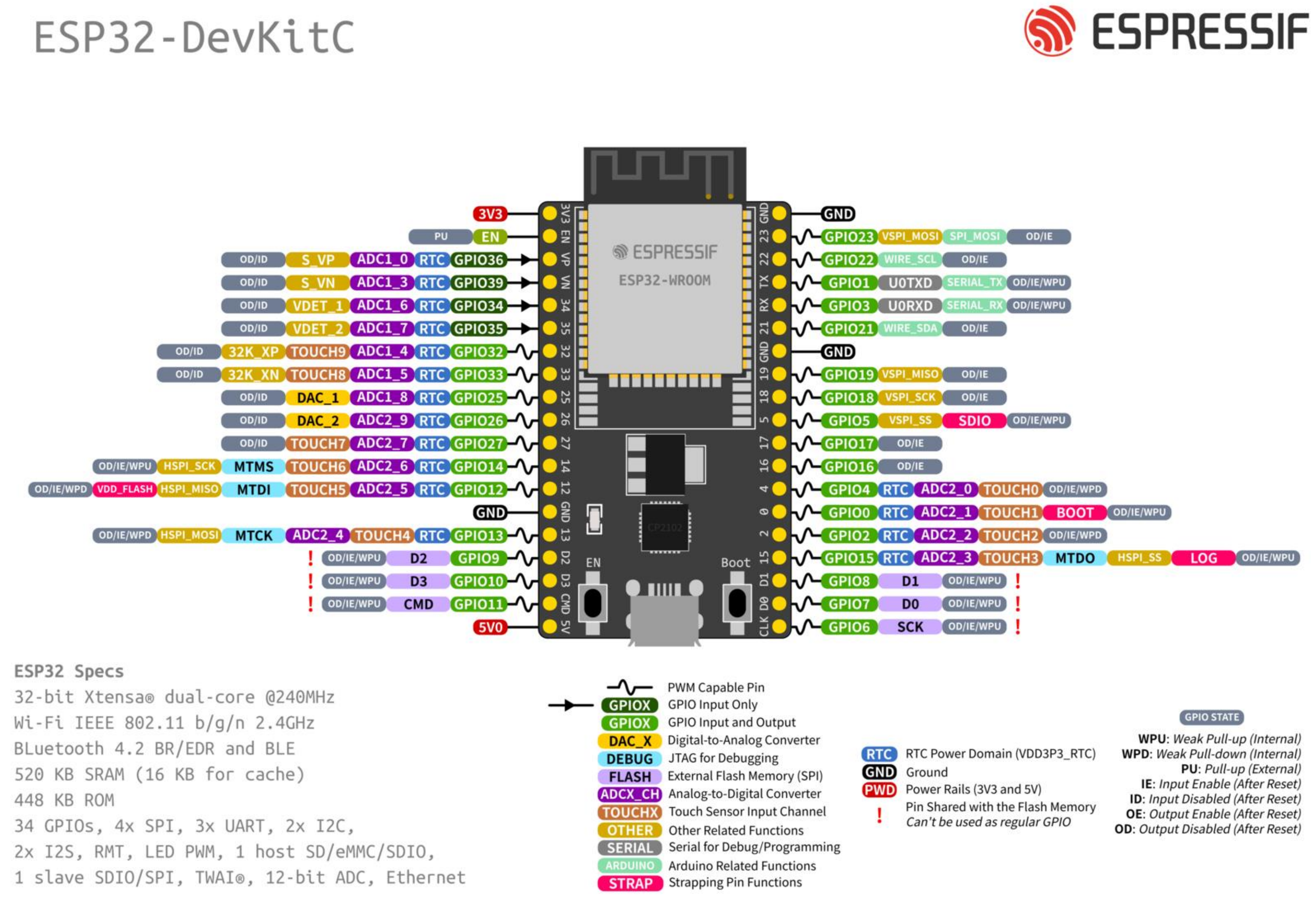
### ESP32 [31]

- CPU: Xtensa dual-core (or single-core) 32-bit LX6 microprocessor, operating at 160 or 240 MHz and performing at up to 600 DMIPS
- Ultra-low power (ULP) co-processor
- Memory: 320 KiB RAM, 448 KiB ROM
- Wi-Fi: 802.11 b/g/n
- Bluetooth: v4.2 BR/EDR and BLE (shares the radio with Wi-Fi)
- 34 × programmable GPIOs
- 12-bit SAR ADC up to 18 channels
- 2 × 8-bit DACs
- 10 × touch sensors (capacitive sensing GPIOs)
- 4 × SPI

### BeagleBone AI [32]

- Dual Arm® Cortex®-A15 microprocessor subsystem
- 1GB RAM and 16GB on-board eMMC flash with high-speed interface
- Gigabit Ethernet, 2.4/5GHz WiFi, and Bluetooth
- microHDMI
- Zero-download out-of-box software experience with Debian GNU/Linux
- £130

## Appendix 5: Espressif ESP32-DevKitC Specifications [33]


ESP32-DevKitC
ESPRESSIF
ESPRESSIF
ESP32-WROOM
CP2102
EN
Boot
3V3 — 3V3
PU EN — EN
OD/ID S_VP ADC1_0 RTC GPIO36 — VP
OD/ID S_VN ADC1_3 RTC GPIO39 — VN
OD/ID VDET_1 ADC1_6 RTC GPIO34 — 34
OD/ID VDET_2 ADC1_7 RTC GPIO35 — 35
OD/ID 32K_XP TOUCH9 ADC1_4 RTC GPIO32 — 32
OD/ID 32K_XN TOUCH8 ADC1_5 RTC GPIO33 — 33
OD/ID DAC_1 ADC1_8 RTC GPIO25 — 25
OD/ID DAC_2 ADC2_9 RTC GPIO26 — 26
OD/ID TOUCH7 ADC2_7 RTC GPIO27 — 27
OD/IE/WPU HSPI_SCK MTMS TOUCH6 ADC2_6 RTC GPIO14 — 14
OD/IE/WPD VDD_FLASH HSPI_MISO MTDI TOUCH5 ADC2_5 RTC GPIO12 — 12
GND — GND
OD/IE/WPD HSPI_MOSI MTCK ADC2_4 TOUCH4 RTC GPIO13 — 13
! OD/IE/WPU D2 GPIO9 — D2
! OD/IE/WPU D3 GPIO10 — D3
! OD/IE/WPU CMD GPIO11 — CMD
5V0 — 5V
GND — GND
23 — GPIO23 VSPI_MOSI SPI_MOSI OD/IE
22 — GPIO22 WIRE_SCL OD/IE
TX — GPIO1 U0TXD SERIAL_TX OD/IE/WPU
RX — GPIO3 U0RXD SERIAL_RX OD/IE/WPU
21 — GPIO21 WIRE_SDA OD/IE
GND — GND
19 — GPIO19 VSPI_MISO OD/IE
18 — GPIO18 VSPI_SCK OD/IE
5 — GPIO5 VSPI_SS SDIO OD/IE/WPU
17 — GPIO17 OD/IE
16 — GPIO16 OD/IE
4 — GPIO4 RTC ADC2_0 TOUCH0 OD/IE/WPD
0 — GPIO0 RTC ADC2_1 TOUCH1 BOOT OD/IE/WPU
2 — GPIO2 RTC ADC2_2 TOUCH2 OD/IE/WPD
15 — GPIO15 RTC ADC2_3 TOUCH3 MTDO HSPI_SS LOG OD/IE/WPU
D1 — GPIO8 D1 OD/IE/WPU !
D0 — GPIO7 D0 OD/IE/WPU !
CLK — GPIO6 SCK OD/IE/WPU !

**ESP32 Specs**

32-bit Xtensa® dual-core @240MHz
Wi-Fi IEEE 802.11 b/g/n 2.4GHz
BLuetooth 4.2 BR/EDR and BLE
520 KB SRAM (16 KB for cache)
448 KB ROM
34 GPIOs, 4x SPI, 3x UART, 2x I2C,
2x I2S, RMT, LED PWM, 1 host SD/eMMC/SDIO,
1 slave SDIO/SPI, TWAI®, 12-bit ADC, Ethernet

| | |
|---|---|
| ∿ | PWM Capable Pin |
| → GPIOX | GPIO Input Only |
| GPIOX | GPIO Input and Output |
| DAC_X | Digital-to-Analog Converter |
| DEBUG | JTAG for Debugging |
| FLASH | External Flash Memory (SPI) |
| ADCX_CH | Analog-to-Digital Converter |
| TOUCHX | Touch Sensor Input Channel |
| OTHER | Other Related Functions |
| SERIAL | Serial for Debug/Programming |
| ARDUINO | Arduino Related Functions |
| STRAP | Strapping Pin Functions |

| | |
|---|---|
| RTC | RTC Power Domain (VDD3P3_RTC) |
| GND | Ground |
| PWD | Power Rails (3V3 and 5V) |
| ! | Pin Shared with the Flash Memory *Can't be used as regular GPIO* |

GPIO STATE

**WPU**: *Weak Pull-up (Internal)*
**WPD**: *Weak Pull-down (Internal)*
**PU**: *Pull-up (External)*
**IE**: *Input Enable (After Reset)*
**ID**: *Input Disabled (After Reset)*
**OE**: *Output Enable (After Reset)*
**OD**: *Output Disabled (After Reset)*

*Figure 115 - Photo to show the specifications and pinouts of the ESP32 devkit used in this project [33]*

## Appendix 6: Adafruit ADS1115 ADC Breakout Specifications [34]

- Supply voltage range: 2 – 5.5V
- Analogue input range: -0.3 – 5.3V (when using 5V Vdd supply)
- 16-bit ADC
- 4-channel inputs
- Continuous current consumption: 150uA
- I2C interface
- Internal Oscillator
- Programmable data rate: 8SPS to 860SPS
- 860 samples/second over I2C

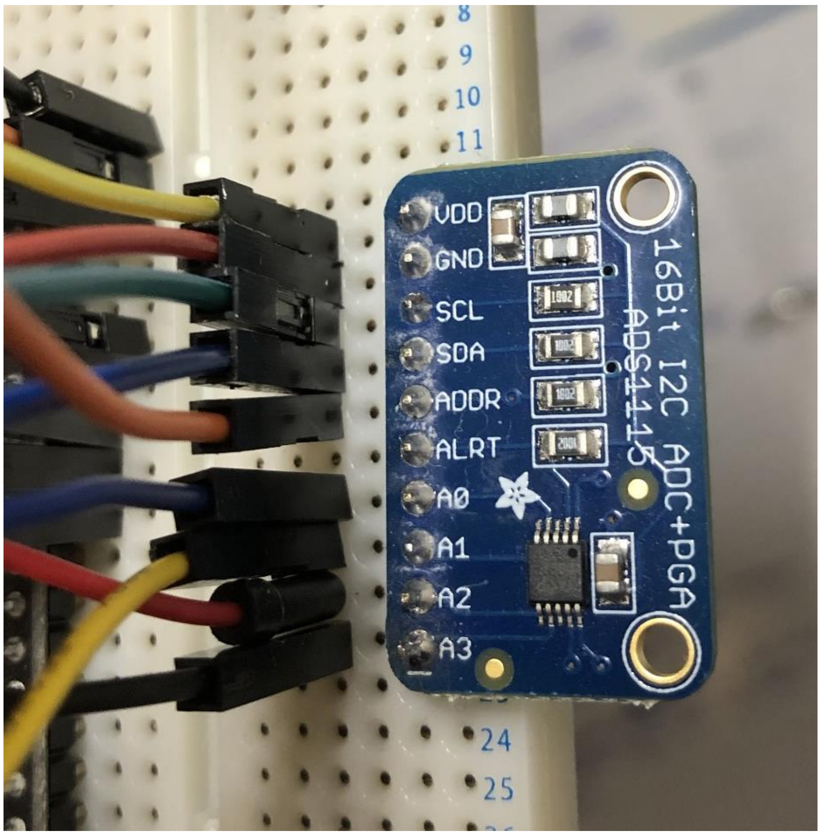


*Figure 118 - Photo taken of the ADA ADS1115 ADC being used in the project prototype breadboard circuit*

## Appendix 7: Adafruit MicroSD Card Breakout Board+ Specifications [35]

- Onboard 5v -> 3v regulator provides 150mA for power-hungry cards
- 3v level shifting means you can use this with ease on either 3v or 5v systems
- Uses a proper level shifting chip, not resistors: fewer problems, and faster read/write access
- Use 3 or 4 digital pins to read and write 2Gb+ of storage
- Activity LED lights up when the SD card is being read or written
- 8 mounting holes and separate header
- Push-push socket with card slightly over the edge of the PCB

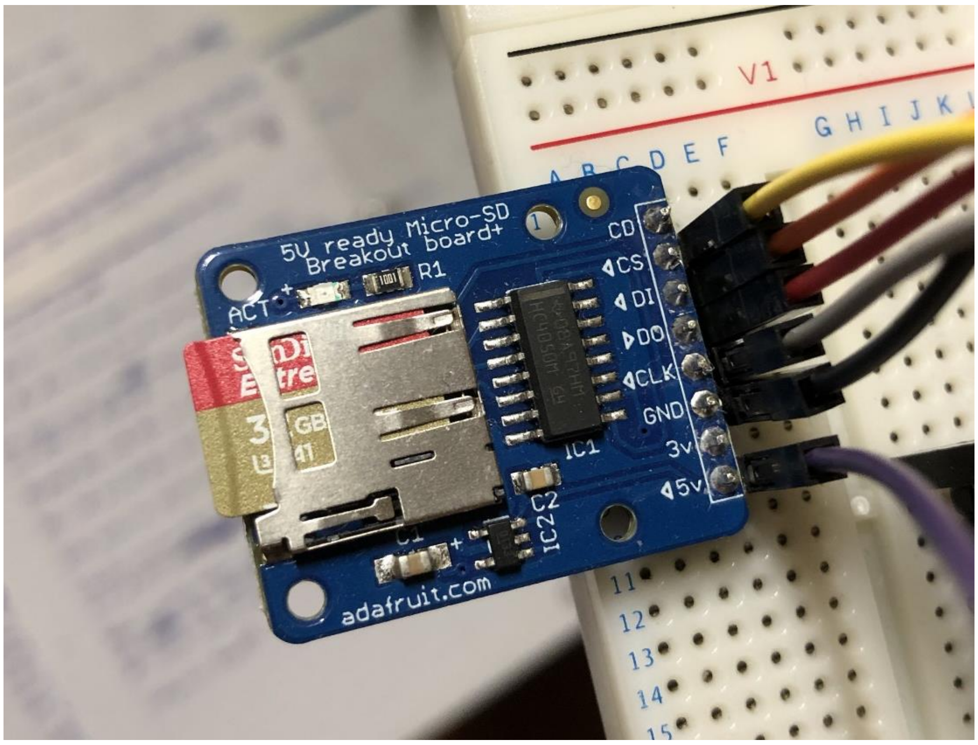


*Figure 119 – Photo taken of the ADA MicroSD breakout being used in the project prototype breadboard circuit*

## Appendix 8: U-Blox Neo-6m GPS Breakout Board Specifications [36]

- GPS Module GY-NEO6M NEO-6M + onboard Battery + EEPROM + Antenna + U.FL connector
- Highly configurable U-Blox Neo-6M module
- Onboard LDO 3.3v regulator
- EEPROM & Battery for faster GPS Lock
- Can be configured to 5Hz update rate for better responsiveness
- The default baud rate is 9600 but can be changed
- Provided with an external antenna
- Can be used with the Arduino TinyGPS Library
- Standard NMEA and/or U-Blox UBX outputs
- 1PPS (1 Pulse per Second) output signal on the connector

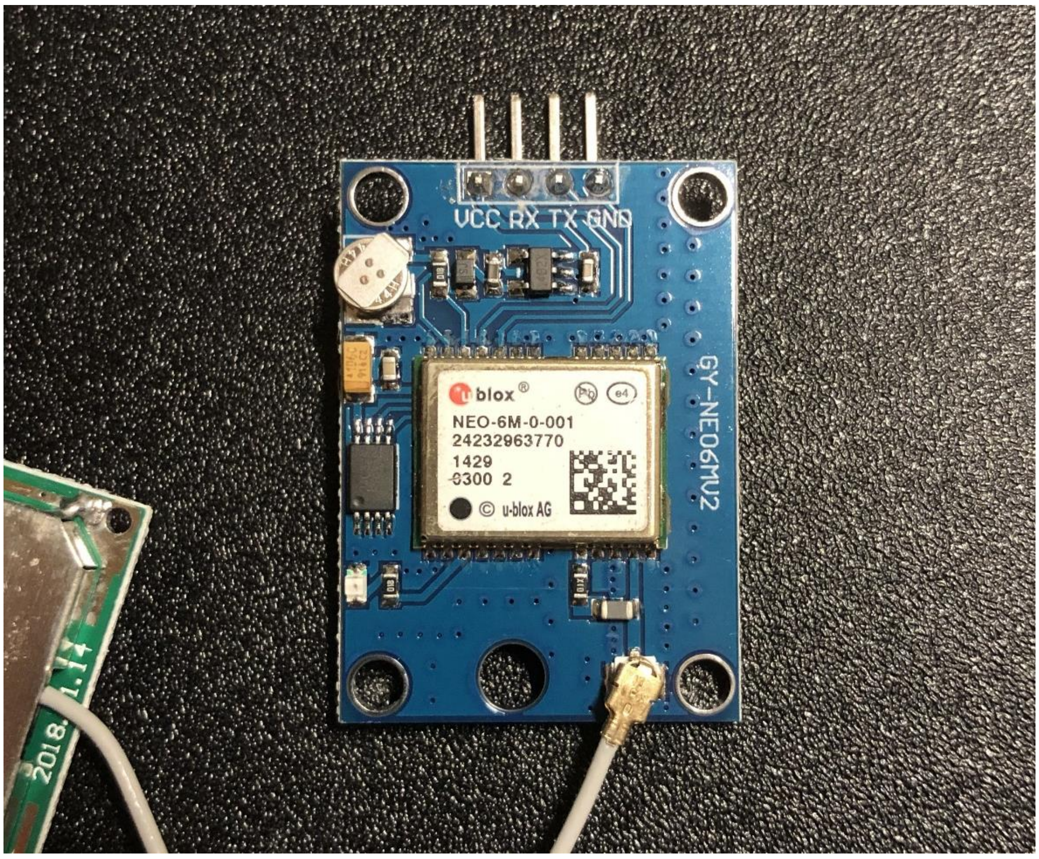


*Figure 120 – Photo taken of the ADA MicroSD breakout that will be used in the project prototype breadboard circuit*

## Appendix 9: Final ESP32 Master C++ Code for ADC Readings, GPS, etc.

```
#include <Adafruit_ADS1X15.h>

#include <Arduino.h>
#include <Wire.h>
#include <WireSlave.h>

#define I2C_SLAVE_ADDR 0x04

Adafruit_ADS1015 ads1;
Adafruit_ADS1015 ads2;

// Required on ESP32 to put the ISR in IRAM.
#ifndef IRAM_ATTR
#define IRAM_ATTR
#endif

// Pointer to the timer
hw_timer_t *My_timer = NULL;

volatile bool new_data = false;
void IRAM_ATTR onTimer() {
  new_data = true;
}

void setup(void)
{
  // Set-up the timer interrupt and start it
  My_timer = timerBegin(0, 80, true);
  timerAttachInterrupt(My_timer, &onTimer, true);
  timerAlarmWrite(My_timer, 833, true);
  timerAlarmEnable(My_timer);

  Serial.begin(2000000);

  // ADS1015 ADCs
  ads1.setDataRate(RATE_ADS1015_3300SPS);
  ads2.setDataRate(RATE_ADS1015_3300SPS);

  if (!ads1.begin(0x48)) {
    Serial.println("Failed to initialize ADS1015 1.");
    ESP.restart();
  }

  if (!ads2.begin(0x49)) {
    Serial.println("Failed to initialize ADS1015 2.");
    ESP.restart();
  }

  // Start differential conversions.
  ads1.startADCReading(ADS1X15_REG_CONFIG_MUX_SINGLE_0,
/*continuous=*/true);
  ads2.startADCReading(ADS1X15_REG_CONFIG_MUX_SINGLE_0,
/*continuous=*/true);

  // GPS serial setup
  Serial2.begin(9600, SERIAL_8N1, 16, 17);  // set up GPS module serial
communication on 9600 baud
}
```

```
void loop(void)
{
  unsigned long startTime;
  unsigned long timeTaken;
  unsigned long prevPositiveZeroCrossing = micros();
  unsigned long prevTime = micros();

  static unsigned long lastWireTransmit = 0;
  static byte x = 0;

  int bufferSize = 400;
  float sumSquareVoltages = 0.0;
  float sumSquareCurrents = 0.0;
  float vrms = 0.0;
  float irms = 0.0;
  float avgScaledVRMS = 0.0;
  float avgScaledIRMS = 0.0;

  float scaledVoltage = 0.0;
  float scaledCurrent = 0.0;
  float scaledVRMS = 0.0;
  float scaledIRMS = 0.0;

  float scaledPower = 0.0;
  float avgScaledPower = 0.0;
  float timeElapsed = 0.0;
  float energyConsumption = 0.0;
  float faultCounter = 0.0;

  float bufferV[bufferSize];
  float bufferC[bufferSize];
  float bufferM[bufferSize];

  float prevVoltage = 0.0;
  float currentFrequency = 0.0;
  float avgFrequency = 0.0;

  int cycleCounter = 1; // used to count the number of full cycles that
have occurred for the running average frequency calculations

  float freqMidPoint = 2.5;
  float midPoint = 2.5;
  float midPointAvg = 0.0;

  float primarySideVoltagePeak = 11000.0;
  float primarySideFaultCurrentPeak = 200.0;

  boolean faultOccuring = false;
  boolean prevFaultOccuring = false;



  // --------------- gps syncing ---------------
  // clear the serial
  while (Serial2.available() > 0) {
    Serial2.read();
    delay(1);
  }

  // wait for the start of a new second
  while (Serial2.available() == 0) {

  }

  // warmup - starting at the exact start of a new second
  for (int i = 0; i < 100 ; i++) {
    while (!new_data) {

    }
    ads1.getLastConversionResults();
    ads2.getLastConversionResults();
  }


  startTime = millis();
  // 1000 seconds
  for (int i = 0; i < 1200000 ; i++) {

    // Tries to send new data to the slave once per second
    if (!new_data && millis() - lastWireTransmit > 1000 && i > 100) {
      // first create a WirePacker that will assemble a packet
      WirePacker packer;
      char values[250];

      // First row
      dtostrf(scaledVoltage, -10, 3, values);
      packer.write(values);
      packer.write(",");

      dtostrf(scaledCurrent, -10, 3, values);
      packer.write(values);
      packer.write(",");

      dtostrf(currentFrequency, -10, 3, values);
      packer.write(values);
      packer.write(",");

      dtostrf(scaledPower, -10, 3, values);
      packer.write(values);
      packer.write(",");


      // Second row
      dtostrf(avgScaledVRMS * sqrt(2), -10, 3, values); // avgScaledVoltage
      packer.write(values);
      packer.write(",");

      dtostrf(avgScaledIRMS  * sqrt(2), -10, 3, values);  //
avgScaledCurrent
      packer.write(values);
      packer.write(",");

      dtostrf(avgFrequency, -10, 3, values);
      packer.write(values);
      packer.write(",");

      dtostrf(avgScaledPower, -10, 3, values);
      packer.write(values);
      packer.write(",");
      // Third row
      dtostrf(energyConsumption, -10, 3, values);
      packer.write(values);
      packer.write(",");

      dtostrf(faultCounter, -10, 1, values);
      packer.write(values);
      packer.write(",");

      dtostrf(timeElapsed, -10, 3, values);
      packer.write(values);
      packer.write(",");

      dtostrf(midPoint, -10, 3, values);
      packer.write(values);
      packer.write(",");


      // close the packet
      packer.end();

      // transmit the packed data
      Wire.beginTransmission(I2C_SLAVE_ADDR);
      while (packer.available()) {    // write every packet byte
        int content = packer.read();
        Wire.write(content);
      }
      Wire.endTransmission();         // stop transmitting
      lastWireTransmit = millis();
    }

    // waits until new data is taken from the ADC at each timer interrupt
    while (!new_data) {
    }

    int16_t resultsADS1 = ads1.getLastConversionResults();
    float voltage = ads1.computeVolts(resultsADS1);

    int16_t resultsADS2 = ads2.getLastConversionResults();
    float current = ads2.computeVolts(resultsADS2);


    // --------------------------- VRMS -----------------------------

    if (i < bufferSize) {
      // midPoint
      midPointAvg = ((midPointAvg * i) + voltage) / (i + 1);
      bufferM[i] = midPoint;

      // vrms
      bufferV[i] = voltage - midPoint;
      sumSquareVoltages += pow(voltage - midPoint, 2);
      vrms = sqrt(sumSquareVoltages / (i + 1));

      // irms
      bufferC[i] = current - midPoint;
      sumSquareCurrents += pow(current - midPoint, 2);
      irms = sqrt(sumSquareCurrents / (i + 1));
    } else {
      int pos = i % bufferSize;
      float oldM = bufferM[pos];
```

```
      float oldV = bufferV[pos];
      float oldC = bufferC[pos];

      // midPoint
      midPointAvg = ((midPointAvg * bufferSize) + voltage - (oldV + oldM))
/ bufferSize;
      midPoint = midPointAvg;
      bufferM[pos] = midPoint;

      // vrms
      sumSquareVoltages = (pow(vrms, 2) * bufferSize) - pow(oldV, 2) +
pow(voltage - midPoint, 2);
      vrms = sqrt(sumSquareVoltages / bufferSize);
      bufferV[pos] = voltage - midPoint;

      // irms
      sumSquareCurrents = (pow(irms, 2) * bufferSize) - pow(oldC, 2) +
pow(current - midPoint, 2);
      irms = sqrt(sumSquareCurrents / bufferSize);
      bufferC[pos] = current - midPoint;
    }

    // scale the voltage and current values to the primary side values
    scaledVRMS = (vrms * primarySideVoltagePeak) / sqrt(2);
    scaledVoltage = (scaledVRMS * sqrt(2)); // 11000V == 1.414V

    scaledIRMS = (irms * primarySideFaultCurrentPeak) / sqrt(2);
    scaledCurrent = (scaledIRMS * sqrt(2)); // 200A == 1.414V

    scaledPower = scaledVoltage * scaledCurrent;  // Watts
    avgScaledPower = ((avgScaledPower * i) + scaledPower) / (i + 1);  //
Watts

    avgScaledVRMS = ((avgScaledVRMS * i) + scaledVRMS) / (i + 1);
    avgScaledIRMS = ((avgScaledIRMS * i) + scaledIRMS) / (i + 1);


    // ------------------------- Frequency ---------------------------

    long currentTime = micros();
    // if at zero crossing (zero crossing from negative to positive
specifically)
    if (prevVoltage <= freqMidPoint && voltage > freqMidPoint) {
      long trueZero = currentTime - ((currentTime - prevTime) * ((voltage -
freqMidPoint) / (voltage - prevVoltage)));

      currentFrequency = 1000000.0 / (trueZero - prevPositiveZeroCrossing);

      if (i > 100) {
        avgFrequency = ((avgFrequency * cycleCounter) + currentFrequency) /
(cycleCounter + 1);
        cycleCounter++;
      } else {
        avgFrequency = currentFrequency;
      }

      prevPositiveZeroCrossing = trueZero;
    }


    // needs 100 samples warmup
    if ( i > 100 ) {
      // energy consumption
      timeElapsed = (millis() - startTime) / 1000;  // s
      energyConsumption = avgScaledPower * timeElapsed / 3600000; // kWh


      // ------------------------ Fault Detection --------------------------
      if (scaledCurrent >= 100.0) { // trip point set at 100A
        faultOccuring = true;
      } else if (scaledCurrent < 40.0) { // fault finished value boundary
set at 40 to ensure fault
        faultOccuring = false;            // has truly finished, rather
than recounting the same fault twice
      }

      // checks for the very start of a fault
      if (prevFaultOccuring == false && faultOccuring == true) {
        faultCounter++;
      }

      prevFaultOccuring = faultOccuring;


      // --------------------------- Plotting ----------------------------

      //        Serial.print("ScaledVoltage:");
      //        Serial.print(scaledVoltage);
      //        Serial.print(" ");
      //
      //        Serial.print("ScaledCurrent:");
      //        Serial.print(scaledCurrent);
      //        Serial.print(" ");
      //
      //        Serial.print("Frequency:");
      //        Serial.print(currentFrequency);
      //        Serial.print(" ");
      //
      //        Serial.print("ScaledPower:");
      //        Serial.print(scaledPower, 3);
      //        Serial.print(" ");
      //
      //        Serial.print("AvgScaledVoltage:");
      //        Serial.print(avgScaledVRMS * sqrt(2));
      //        Serial.print(" ");
      //
      //        Serial.print("AvgScaledCurrent:");
      //        Serial.print(avgScaledIRMS * sqrt(2));
      //        Serial.print(" ");
      //
      //        Serial.print("AvgFrequency:");
      //        Serial.print(avgFrequency);
      //        Serial.print(" ");
      //
      //        Serial.print("AvgPower:");
      //        Serial.print(avgScaledPower, 3);
      //        Serial.print(" ");
      //
      //        Serial.print("EnergyConsumptionkWh:");
      //        Serial.print(energyConsumption);
      //        Serial.print(" ");
      //
      //      Serial.print("TimeElapsed:");
      //      Serial.print(timeElapsed);
      //      Serial.print(" ");
      //
      //      Serial.print("ScaledVRMS:");
      //      Serial.print(scaledVRMS);
      //      Serial.print(" ");
      //
      //      Serial.print("ScaledIRMS:");
      //      Serial.print(scaledIRMS);
      //      Serial.print(" ");
      //      Serial.print("AvgScaledVRMS:");
      //      Serial.print(avgScaledVRMS);
      //      Serial.print(" ");
      //
      //      Serial.print("AvgScaledIRMS:");
      //      Serial.println(avgScaledIRMS);

      Serial.print("ADC1Voltage:");
      Serial.print(voltage);
      Serial.print(" ");

      Serial.print("ADC2Current:");
      Serial.print(current);
      Serial.print(" ");

      Serial.print("FaultCounter:");
      Serial.print(faultCounter);
      Serial.print(" ");

      Serial.print("MidPoint:");
      Serial.println(midPoint);

    }

    prevVoltage = voltage;
    prevTime = currentTime;

    new_data = false;
  }

  timeTaken = millis() - startTime;


}
```

## Appendix 10: Final ESP32 Slave C++ Code for storing to MicroSD card, hosting station and AP web server, sending data to Spring Boot application pipeline for cloud website and storage, etc.

```
#include <Arduino.h>
#include <Wire.h>
#include <WireSlave.h>
#include <string>
#include "FS.h"
#include "SD.h"
#include "SPI.h"
#include <WiFi.h>
#include <HTTPClient.h>
#include "time.h"

#include <ESPAsyncWebServer.h>

#define SDA_PIN 21
#define SCL_PIN 22
#define I2C_SLAVE_ADDR 0x04

// define prototypes for functions
void receiveEvent(int howMany);
void appendFile(fs::FS &fs, const char * path, String message);
void sendReadingsToBackend();

const char *ssid = "";  // your local network ssid
const char *password = "";  // your local network password

const char *soft_ap_ssid = "SmartWirelessSensorAP";
const char *soft_ap_password = "testpassword";

AsyncWebServer server(80);

const char *ntpServer = "pool.ntp.org";
const long  gmtOffset_sec = 0;
const int   daylightOffset_sec = 3600;

float voltage = 0.0;
float current = 0.0;
float frequency = 0.0;
float power = 0.0;

float avgVoltage = 0.0;
float avgCurrent = 0.0;
float avgFrequency = 0.0;
float avgPower = 0.0;

float energyConsumption = 0.0;
float faultCounter = 0.0;
float timeElapsed = 0.0;
float offset = 0.0;

void setup()
{
  Serial.begin(2000000);

  // I2C
  bool success = WireSlave.begin(SDA_PIN, SCL_PIN, I2C_SLAVE_ADDR);
  if (!success) {
    Serial.println("I2C slave init failed");
    ESP.restart();
  }

  // microSD card
  while (!SD.begin()) {
    delay(100);
    Serial.println("Failed to initialise microSD card - check that it's
inserted");
    ESP.restart();
  }

  uint8_t cardType = SD.cardType();
  if (cardType == CARD_NONE) {
    Serial.println("Invalid card type: none");
    ESP.restart();
  }

  // Wi-Fi server mode
  WiFi.mode(WIFI_MODE_APSTA);
  WiFi.softAP(soft_ap_ssid, soft_ap_password);
  WiFi.begin(ssid, password);

  // Connect to Wi-Fi network


  Serial.println("Connecting");
  unsigned long startTime = millis();
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
    if (millis() - startTime > 3000) {
      Serial.println("***WiFi Disconnected - Please try restarting the
ESP32***");
      Serial.println("The sensor readings are still available via the Smart
Wireless Sensor Access Point");
      Serial.print("Connect to the \"SmartWirelessSensorAP\" Access Point
on any device, then go to http://");
      Serial.print(WiFi.softAPIP());
      Serial.println("/readings");
      break;
    }
  }
  Serial.println("");

  WireSlave.onReceive(receiveEvent);

  // Time - Use Wi-Fi to syncronise the ESP32 time with this server (only
needs done on initialisation)
  configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);


  // Wi-Fi Server and AP Server Setup
  Serial.print("ESP32 IP as soft AP: ");
  Serial.println(WiFi.softAPIP());

  Serial.print("ESP32 IP on the WiFi network: ");
  Serial.println(WiFi.localIP());

  server.on("/readings", HTTP_GET, [](AsyncWebServerRequest * request) {
    const char * format =
      "Smart Wireless Sensor Measurements\n"
      "----------------------------------\n"
      "Voltage: %.3fkV\n"
      "Current: %.3fA\n"
      "Frequency: %.3fHz\n"
      "Power Consumption: %.3fkW\n"
      "Average Voltage: %.3fkV\n"
      "Average Current: %.3fA\n"
      "Average Frequency: %.3fHz\n"
      "Average Power Consumption: %.3fkW\n"
      "Energy Consumption: %.3fkWh\n"
      "Number of Faults: %.0f\n"
      "Time Elapsed: %.0fs\n"
      "Voltage Offset/Midpoint: %.3fV\n";


    char message[450];
    snprintf(
      message,
      450,
      format,

      voltage,
      current,
      frequency,
      power,

      avgVoltage,
      avgCurrent,
      avgFrequency,
      avgPower,

      energyConsumption,
      faultCounter,
      timeElapsed,
      offset
    );

    if (ON_STA_FILTER(request)) {
      request->send(200, "text/plain", message);
      return;

    } else if (ON_AP_FILTER(request)) {
      request->send(200, "text/plain", message);
      return;
    }
  });

  server.begin();
}

void loop()
{
  WireSlave.update();
  delay(1);
}

void sendReadingsToBackend() {
  //Check WiFi connection status
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    char requestURL[250];
    snprintf(
      requestURL,
      250,
      "https://smart-wireless-sensor-
backend.nw.r.appspot.com/setReadings/%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3
f,%.3f,%.3f,%.3f,%.3f", // cloud server URL
      //      "http://192.168.0.26:8081/setReadings/%.3f,%.3f,%.3f,%.3f,%.3
f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f", // local NPM server URL
      voltage,
      current,
      frequency,
      power,

      avgVoltage,
      avgCurrent,
      avgFrequency,
      avgPower,

      energyConsumption,
      faultCounter,
      timeElapsed,
      offset
    );

    String serverPath = requestURL;

    // Your Domain name with URL path or IP address with path
    http.begin(serverPath.c_str());
    Serial.print("Sending GET request to: ");
    Serial.println(serverPath.c_str());

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

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

    Serial.println("***WiFi Disconnected - Please try restarting the
ESP32***");
    Serial.println("The sensor readings are still available via the Smart
Wireless Sensor Access Point");
    Serial.print("Connect to the \"SmartWirelessSensorAP\" Access Point on
any device, then go to http://");
    Serial.print(WiFi.softAPIP());
    Serial.println("/readings");
  }
```

```
}


void receiveEvent(int howMany)
{
  String s = "";
  while (WireSlave.available()) // loop through every byte in the I2C
message
  {
    char ch = WireSlave.read();
    s += ch;
  }

  //  Serial.println(s);

  int commaCount = 0;
  int start = 0;
  for (int i = 0; i < s.length(); i++) {
    char ch = s.charAt(i);
    if (ch == ',') {
      float floatValue = s.substring(start, i).toFloat();
      switch (commaCount) {
        case 0:
          voltage = floatValue / 1000.0;
          break;
        case 1:
          current = floatValue;
          break;
        case 2:
          frequency = floatValue;
          break;
        case 3:
          power = floatValue / 1000.0;
          break;

        case 4:
          avgVoltage = floatValue / 1000.0;
          break;
        case 5:
          avgCurrent = floatValue;
          break;
        case 6:
          avgFrequency = floatValue;
          break;
        case 7:
          avgPower = floatValue / 1000.0;
          break;

        case 8:
          energyConsumption = floatValue;
          break;
        case 9:
          faultCounter = floatValue;
          break;
        case 10:
          timeElapsed = floatValue;
          break;
        case 11:
          offset = floatValue;
          break;
      }
      start = i + 1;
      commaCount++;
    }
  }

  const char * format =
    "------------------------------------------------------------\n"
    "Voltage: %.3fkV\n"
    "Current: %.3fA\n"
    "Frequency: %.3fHz\n"
    "Power Consumption: %.3fkW\n"
    "Average Voltage: %.3fkV\n"
    "Average Current: %.3fA\n"
    "Average Frequency: %.3fHz\n"
    "Average Power Consumption: %.3fkW\n"
    "Energy Consumption: %.3fkWh\n"
    "Number of Faults: %.0f\n"
    "Time Elapsed: %.0fs\n"
    "Voltage Offset/Midpoint: %.3fV\n";


  char message[450];
  snprintf(
    message, 450, format,

    voltage,
    current,
    frequency,
    power,

    avgVoltage,
    avgCurrent,
    avgFrequency,
    avgPower,

    energyConsumption,
    faultCounter,
    timeElapsed,
    offset
  );

  struct tm timeinfo;
  char dateTimeString[] = "/Smart_Wireless_Sensor_Readings.txt";
  if (getLocalTime(&timeinfo)) {
    strftime(
      dateTimeString,
      100,
      "/%d%B%Y_%H%M.txt",
      &timeinfo
    );
  } else {
    Serial.println("Failed to obtain time, appending to
Smart_Wireless_Sensor_Readings.txt file");
  }

  const char * path = dateTimeString;
  appendFile(SD, path, message);

  sendReadingsToBackend();
}
```

```
void appendFile(
  fs::FS &fs,
  const char * path,
  String message
) {

  File file = fs.open(path, FILE_APPEND);
  if (!file) {
    Serial.println("Text file could not be appended to or created");
    return;
  }

  Serial.print(message);
  if (!file.print(message)) {
    Serial.println("***File append failed: Check that microSD card has been
inserted, then reboot***");
  }

  file.close();
}
```

## Appendix 11: MATLAB simulation to validate the true frequency and vrms algorithms that are run on the master ESP32

```
clear all;
clc;

fs = 1200; % sampling frequency
dt = 1/fs; % seconds per sample

f = [49.97, 49.98, 49.97];
cycles = [50, 150, 70];

stopTimeTotal = 0;
totalSamples = 1;
for i = 1:length(f)
    stopTimes(i) = cycles(i) / f(i);
    samples(i) = stopTimes(i) * fs;
    stopTimeTotal = stopTimeTotal + stopTimes(i);
    totalSamples = totalSamples + samples(i);
end

t = (0:dt:stopTimeTotal);

for i = 1:totalSamples
    if (i <= samples(1) + 1)
        y(i) = sqrt(2) * sin(2 * pi * f(1) * t(i));
        yTrue(i) = f(1);
    elseif (i <= samples(1) + samples(2) + 1)
        y(i) = sqrt(2) * sin(2 * pi * f(2) * t(i));
        yTrue(i) = f(2);
    elseif (i <= samples(1) + samples(2) + samples(3) + 1)
        y(i) = sqrt(2) * sin(2 * pi * f(3) * t(i));
        yTrue(i) = f(3);
    end
end

plot(t, y);

hold on

prevVoltage = 0.0;
prevTime = 0.0;
currentFrequency = 0.0;
disp(currentFrequency)

midPoint = 0;
prevPositiveZeroCrossing = -1;

buffer = y(1:24);
disp(buffer);
sumSquareVoltages = 0;
for i = 1:length(buffer)
    sumSquareVoltages = sumSquareVoltages + buffer(i)^2;
    disp(sumSquareVoltages);
    vrmsPlot(i) = 0;
end
vrms = sqrt(sumSquareVoltages / length(buffer));
```

```
for i = 1:length(y)
    voltage = y(i);
    currentTime = t(i);

    % VRMS
    if (i > length(buffer))

        pos = mod(i-1, length(buffer)) + 1;

        old = buffer(pos);
        new = y(i);

        sumSquareVoltages = (vrms^2 * length(buffer)) - old^2 + new^2;
        vrms = sqrt(sumSquareVoltages/length(buffer));
        buffer(pos) = y(i);

        disp(round(vrms,3));  % 3dp
        vrmsPlot(i) = vrms;
    end

    % Frequency
    if (prevVoltage <= midPoint && voltage > midPoint)
        trueZero = currentTime - ((currentTime - prevTime) * ((voltage - midPoint)
/ (voltage - prevVoltage)));

        currentFrequency = 1 / (trueZero - prevPositiveZeroCrossing);
        if (i > 100)
            disp(round(currentFrequency, 3));   % 3dp
        end
        prevPositiveZeroCrossing = trueZero;
    end

    frequenciesPlot(i) = currentFrequency;

    prevVoltage = voltage;
    prevTime = currentTime;
end


plot(t,frequenciesPlot)

hold on

plot(t, yTrue)

hold on

plot(t, vrmsPlot)

disp("length(t)");
disp(length(t));
disp("length(yTrue)");
disp(length(yTrue));
```

## Appendix 12: Backend Spring Boot Java Application

### SensorController.java file

```java
package com.sam.ross.sensor.contoller;

import com.sam.ross.sensor.objects.SensorData;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
// specifies that should only accept HTTP requests received from the React
JS web URL
@CrossOrigin(origins = {"https://smart-wireless-sensor-
backend.nw.r.appspot.com/"})
//@CrossOrigin  // uncomment to enable HTTP requests from any origin
@Slf4j
public class SensorController {
    double voltage = 0.0;
    double current = 0.0;
    double frequency = 0.0;
    double avgVoltage = 0.0;
    double avgCurrent = 0.0;
    double avgFrequency = 0.0;
    double timeElapsed = 0.0;
    double power = 0.0;
    double avgPower = 0.0;
    double energyConsumption = 0.0;
    double offset = 0.0;
    double faultCounter = 0.0;

    @GetMapping("/setReadings/{data}")
    public ResponseEntity<String> setValues(@PathVariable String data) {
        log.info("setValues endpoint has received a request (controller)");

        String[] values = data.split(",");

        for (String value: values) {
            if (value.equals("nan")) {
                return ResponseEntity.ok("\"nan value received\"");
            }
        }

        voltage = Double.parseDouble(values[0]);
        current = Double.parseDouble(values[1]);
        frequency = Double.parseDouble(values[2]);
        power = Double.parseDouble(values[3]);
        avgVoltage = Double.parseDouble(values[4]);
        avgCurrent = Double.parseDouble(values[5]);
        avgFrequency = Double.parseDouble(values[6]);
        avgPower = Double.parseDouble(values[7]);
        energyConsumption = Double.parseDouble(values[8]);
        faultCounter = Double.parseDouble(values[9]);
        timeElapsed = Double.parseDouble(values[10]);
        offset = Double.parseDouble(values[11]);

        return ResponseEntity.ok("Success: " + data);
    }

    @GetMapping("/getReadings")
    public ResponseEntity<SensorData> getValues() {
        log.info("getValues endpoint has received a request (controller)");

        SensorData sensorData = SensorData.builder()
                .voltage(voltage)
                .current(current)
                .frequency(frequency)
                .avgVoltage(avgVoltage)
                .avgCurrent(avgCurrent)
                .avgFrequency(avgFrequency)
                .timeElapsed(timeElapsed)
                .power(power)
                .avgPower(avgPower)
                .energyConsumption(energyConsumption)
                .offset(offset)
                .faultCounter(faultCounter)
                .build();

        return ResponseEntity.ok(sensorData);
    }

    @CrossOrigin    // ping requests accepted from any origin
    @GetMapping("/ping")
    public ResponseEntity<String> ping() {
        log.info("ping endpoint has received a request (controller)");

        return ResponseEntity.ok("pong");
    }

}
```

SensorData.java file

```java
package com.sam.ross.sensor.objects;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
@Getter
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class SensorData {
    private double voltage;
    private double current;
    private double frequency;
    private double power;
    private double avgVoltage;
    private double avgCurrent;
    private double avgFrequency;
    private double avgPower;
    private double energyConsumption;
    private double faultCounter;
    private double timeElapsed;
    private double offset;
}
```

## Appendix 13: React JS Website Application

index.js – The main ReactJS JavaScript file

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';

class MainWrapper extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      baseUrl: "http://192.168.0.26:8081/getReadings",
      // baseUrl: "https://smart-wireless-sensor-
backend.nw.r.appspot.com/getReadings",

      voltage: 0.0,
      current: 0.0,
      frequency: 0.0,
      power: 0.0,

      avgVoltage: 0.0,
      avgCurrent: 0.0,
      avgFrequency: 0.0,
      avgPower: 0.0,

      energyConsumption: 0.0,
      numberOfFaults: 0,
      timeElapsed: 0.0,
      offset: 0.0,

      demo: "demo-on",
      counter: 0,

      // alphabetical here for handy alignment with chrome dev tools for
obtaining demo values
      demoAvgCurrent: [18.601, 18.599, 18.598, 18.596, 18.596],
      demoAvgFrequency: [49.707, 49.714, 49.701, 49.705, 49.717],
      demoAvgPower: [198.121, 198.119, 198.079, 198.032, 198.066],
      demoAvgVoltage: [10.651, 10.650, 10.648, 10.651, 10.654],
      demoCurrent: [18.598, 18.596, 18.596, 18.601, 18.599],
      demoFrequency: [49.701, 49.705, 49.717, 49.707, 49.714],
      demoOffset: [2.497, 2.483, 2.5, 2.484, 2.497],
      demoPower: [198.032, 198.066, 198.121, 198.119, 198.079],
      demoTimeElapsed: 1,
      demoVoltage: [10.648, 10.651, 10.654, 10.651, 10.650],
      prevResultTimeElapsed: 0.0,
      sameReadingsCounter: 0,
    };
  }

  componentDidMount = () => {
    this.myTimer = setInterval(() => {
      this.getReadings();
    }, 1000);
  }

  componentWillUnmount = () => {
    clearInterval(this.myTimer);
  }

  getReadings() {
    fetch(this.state.baseUrl)
      .then((res) => {
        if (res.status >= 400 || res.status === 204) {
          this.setDemoValues();
          throw new Error(res.status);
        }
        return res.json();
      })
      .then(
        (result) => {
          console.log(result);
          let demoResult = "demo-off";
          let updatedSameReadingsCounter = this.state.sameReadingsCounter + 1;
          if (result.timeElapsed !== this.state.prevResultTimeElapsed) {
            updatedSameReadingsCounter = 0;
          }

          this.setState({
            sameReadingsCounter: updatedSameReadingsCounter
          });

          if (this.state.sameReadingsCounter > 7) {
            this.setDemoValues();
            demoResult = "demo-on";
          } else {
            if (result.timeElapsed === this.state.prevResultTimeElapsed) {

            }
            this.setState({
              voltage: result.voltage,
              current: result.current,
              frequency: result.frequency,
              power: result.power,

              avgVoltage: result.avgVoltage,
              avgCurrent: result.avgCurrent,
              avgFrequency: result.avgFrequency,
              avgPower: result.avgPower,

              energyConsumption: result.energyConsumption,
              numberOfFaults: result.faultCounter,
              timeElapsed: result.timeElapsed,
              offset: result.offset,

              demoTimeElapsed: result.timeElapsed,

              demo: demoResult,

              prevResultTimeElapsed: result.timeElapsed,
            })
          }
        },
        (error) => {
          console.log("Unexpected error returned: " + error.message);
          this.setDemoValues();
        }
      )
  }

  setDemoValues() {
    let count = this.state.counter;
    this.setState({
      demoTimeElapsed: this.state.demoTimeElapsed + 1,
      demoEnergyConsumption: this.state.demoEnergyConsumption + 0.055023167,
    })

    // reset demo state every 24 hours
    if (this.state.demoTimeElapsed === 86401) {
      this.setState({
        demoTimeElapsed: 0,
        demoEnergyConsumption: 0,
      })
    }

    this.setState({
      voltage: this.state.demoVoltage[count % 5],
      current: this.state.demoCurrent[count % 5],
      frequency: this.state.demoFrequency[count % 5],
      power: this.state.demoPower[count % 5],
      avgVoltage: this.state.demoAvgVoltage[count % 5],
      avgCurrent: this.state.demoAvgCurrent[count % 5],
      avgFrequency: this.state.demoAvgFrequency[count % 5],
      avgPower: this.state.demoAvgPower[count % 5],

      energyConsumption: this.state.demoAvgPower[count % 5] *
this.state.demoTimeElapsed / 3600.0,
      timeElapsed: this.state.demoTimeElapsed,
      offset: this.state.demoOffset[count % 5],
      counter: count + 1,

      demo: "demo-on"
    });
  }


  render() {
    return (
      <div className="container">
        <header className="header">
          <h1 className="header1">Smart Wireless Sensor Readings</h1>
        </header>

        <div className="section-middle">
          <div className='outer-row'>
            <div className="row">
              <div className="readings">
                <p>Voltage:</p>
              </div>
              <div className="readings">
                <p>Current:</p>
              </div>
              <div className="readings">
                <p>Frequency:</p>
              </div>
              <div className="readings">
                <p>Power:</p>
              </div>
            </div>

            <div className="row">
              <div className="readings">
                <p>{this.state.voltage.toFixed(3)}kV</p>
              </div>
              <div className="readings">
                <p>{this.state.current.toFixed(3)}A</p>
              </div>
              <div className="readings">
                <p>{this.state.frequency.toFixed(3)}Hz</p>
              </div>
              <div className="readings">
                <p>{this.state.power.toFixed(3)}kW</p>
              </div>
            </div>
          </div>

          <div className='outer-row'>
            <div className="row">
              <div className="readings">
                <p>Avg Voltage:</p>
              </div>
              <div className="readings">
                <p>Avg Current:</p>
              </div>
              <div className="readings">
                <p>Avg Frequency:</p>
              </div>
              <div className="readings">
                <p>Avg Power:</p>
              </div>
            </div>

            <div className="row">
              <div className="readings">
                <p>{this.state.avgVoltage.toFixed(3)}kV</p>
              </div>
              <div className="readings">
                <p>{this.state.avgCurrent.toFixed(3)}A</p>
              </div>
              <div className="readings">
                <p>{this.state.avgFrequency.toFixed(3)}Hz</p>
              </div>
              <div className="readings">
                <p>{this.state.avgPower.toFixed(3)}kW</p>
              </div>
            </div>
          </div>

          <div className='outer-row'>
            <div className="row">
              <div className="readings" id='readings'>
                <p>Energy Consumption:</p>
              </div>
              <div className="readings" id='readings'>
                <p>Number of Faults:</p>
              </div>
              <div className="readings" id='readings'>
```

```
              <p>Time Elapsed:</p>
            </div>
            <div className="readings" id='readings'>
              <p>Offset/midpoint:</p>
            </div>
          </div>

          <div className="row">
            <div className="readings" id='readings'>
              <p>{this.state.energyConsumption.toFixed(3)}kWh</p>
            </div>
            <div className="readings" id='readings'>
              <p>{this.state.numberOfFaults}</p>
            </div>
            <div className="readings" id='readings'>
              <p>{this.state.timeElapsed}s</p>
            </div>
            <div className="readings" id='readings'>
              <p>{this.state.offset.toFixed(3)}V</p>
            </div>
          </div>
        </div>
      </div>

      <header className="header">
        <div className={this.state.demo}>
          Demo on: ESP32 pipeline is currently not sending any data to the
backend
        </div>
      </header>
    </div>
  );
 }
}

// ========================================

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<MainWrapper />);
```

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
  <meta name="description" content="Web site created using create-react-app"
/>
  <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />

  <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />

  <title>ESP32 Sensor Readings</title>
</head>

<body>
  <div class="container">
    <div id="root"></div>
  </div>
</body>

</html>
```

index.css

```css
html,
body {
  margin: 0;
  min-height: 100%;
  height: 100%;
}

body {
  font: 14px "Century Gothic", Futura, sans-serif;

  font-family: system-ui, -apple-system, system-ui, "Helvetica Neue",
Helvetica,
  Arial, sans-serif;
  max-width: 100%;
  overflow-x: hidden;
}

.header {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 27%;
}

h1 {
  font-weight: 200;
  font-size: 4rem;
}

#root {
  height: 100%;
  margin: 0;
  display:flex;
  justify-content: center;
  flex-direction: column;
  flex-grow: 1;
}

.container {
  height: 100%;
  margin: 0;
  display: flex;
  flex-direction: column;
}

/* index.js CSS */
```

```
.section-middle {
  display: flex;
  flex-direction: column;
  justify-content: center;
  flex-grow: 1;
}

.outer-row {
  margin: 2rem 0;
}

.row {
  display:flex;
  justify-content: space-evenly;
  height: 2.7rem;
}

.readings {
  width: 15%;
  font-size: 1.25rem;
  text-align: center;
  display: flex;
  justify-content: space-evenly;
  background-color: rgb(249, 249, 249);
}



.demo-off {
  color: transparent;
}

p {
  margin: 0;
  display: flex;
  flex-direction: column;
  justify-content: center;
}
```

# References

[1] B. Weedy, B. Cory, N. Jenkins, J. Ekanayake and G. Strbac, Electric Power Systems, Wiley, 2012.

[2] Entsoe, “Digital Fault Recorders,” [Online]. Available: https://www.entsoe.eu/Technopedia/techsheets/digital-fault-recorders.

[3] A. O. Pires, “Continuing evolution of fault recording,” [Online]. Available: https://www.pacw.org/continuing-evolution-of-fault-recording.

[4] H. Rich and L. Barcelos, “The Return of the Dedicated DFR,” GE Grid Solutions, LLC.

[5] Qualitrolcorp, “Why Should You Use Standalone DFRs?,” [Online]. Available: https://www.qualitrolcorp.com/resource-library/blog/why-should-you-use-standalone-dfrs/.

[6] N. Grid, “Substation Construction,” [Online]. Available: https://www.nationalgrid.co.uk/substation-construction.

[7] PiCockpit, “Everything about Raspberry Pi Zero 2 W,” [Online]. Available: https://picockpit.com/raspberry-pi/everything-about-raspberry-pi-zero-2-w/.

[8] U. Electronic, “RP2040 VS ESP32[Video+FAQ]: Which one is better?,” [Online]. Available: https://www.utmel.com/components/rp2040-vs-esp32-which-one-is-better?id=1478.

[9] Okdo, “The Differences Between Single-Board Computers vs. Microcontrollers,” [Online]. Available: https://www.okdo.com/blog/single-board-computers-vs-microcontrollers/.

[10] Wikipedia, “ESP32,” [Online]. Available: https://en.wikipedia.org/wiki/ESP32.

[11] R. N. Tutorials, “ESP32 ADC – Read Analog Values with Arduino IDE,” [Online]. Available: https://randomnerdtutorials.com/esp32-adc-analog-read-arduino-ide/.

[12] Adafruit, “Micro SD Card Breakout Board Tutorial - Arduino Wiring,” [Online]. Available: https://learn.adafruit.com/adafruit-micro-sd-breakout-board-card-tutorial/arduino-wiring.

[13] Arduino, “ADS1115 high speed data rate,” [Online]. Available: https://forum.arduino.cc/t/ads1115-high-speed-data-rate/1043434/7.

[14] Adafruit, “Micro SD Card Breakout Board Tutorial - Arduino Library,” [Online]. Available: https://learn.adafruit.com/adafruit-micro-sd-breakout-board-card-tutorial/arduino-wiring.

[15] GridWatch, “Power Grid Frequency,” [Online]. Available: https://gridwatch.co.uk/frequency.

[16] “The Nyquist–Shannon Theorem: Understanding Sampled Systems,” [Online]. Available: https://www.allaboutcircuits.com/technical-articles/nyquist-shannon-theorem-understanding-sampled-systems/.

[17] LiveSparks, “Easiest ESP32 BLE (Bluetooth Low Energy) Tutorial | Arduino,” [Online]. Available: https://www.youtube.com/watch?v=P0aqbD9umDE.

[18] “GPS Sentences | NMEA Sentences | GPGGA GPGLL GPVTG GPRMC,” [Online]. Available: https://www.rfwireless-world.com/Terminology/GPS-sentences-or-NMEA-sentences.html.

[19] “TinyGPSPlus,” [Online]. Available: https://github.com/mikalhart/TinyGPSPlus.

[20] “ESP32 Timers & Timer Interrupts,” [Online]. Available: https://circuitdigest.com/microcontroller-projects/esp32-timers-and-timer-interrupts.

[21] “DacESP32,” [Online]. Available: https://github.com/yellobyte/DacESP32.

[22] “UART vs I2C vs SPI – Communication Protocols and Uses,” [Online]. Available: https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/.

[23] ESP32_I2C_Slave. [Online]. Available: https://github.com/gutierrezps/ESP32_I2C_Slave.

[24] “Question about Esp32 using WIFI and BLUETOOTH at same time,” [Online]. Available: https://github.com/pschatzmann/arduino-audio-tools/issues/9.

[25] S. B. -. Introduction. [Online]. Available: https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm.

[26] L. Java. [Online]. Available: https://www.javatpoint.com/lombok-java.

[27] “ESP32 NTP Client-Server: Get Date and Time (Arduino IDE),” [Online]. Available: https://randomnerdtutorials.com/esp32-date-time-ntp-client-server-arduino/.

[28] RS, “Siemens SICAM P855 Power Quality Analyser,” [Online]. Available: https://uk.rs-online.com/web/p/power-quality-analysers/2363545.

[29] PJRC, “Teensy® 4.1 Development Board,” [Online]. Available: https://www.pjrc.com/store/teensy41.html.

[30] Wikipedia, “ESP8622,” [Online]. Available: https://en.wikipedia.org/wiki/ESP8266.

[31] Wikipedia, “ESP32,” [Online]. Available: https://en.wikipedia.org/wiki/ESP32.

[32] Beagleboard, “BeagleBone® AI,” [Online]. Available: https://beagleboard.org/ai.

[33] Espressif, “esp32-devkitC-v4-pinout,” [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/_images/esp32-devkitC-v4-pinout.png.

[34] Components101, “ADS1115 Module with Programmable Gain Amplifier,” [Online]. Available: https://components101.com/modules/ads1115-module-with-programmable-gain-amplifier.

[35] Adafruit, “MicroSD card breakout board+,” [Online]. Available: https://www.adafruit.com/product/254.

[36] 247geek, “U-Blox GPS Module GY-NEO6M NEO-6M + Antenna,” [Online]. Available: https://247geek.co.uk/gyneo6m?srsltid=AeTuncp8-sVH4k3BuExIKma4sYFVFdpsvK00YWKsGowhE7qwgj2B63sq5rE.